

Foundations of Applied Cryptography and Cybersecurity

Amir Herzberg
Comcast Professor of Security Innovations
Department of Computer Science and Engineering
University of Connecticut

November 26, 2022

For updated draft, presentations (lectures), lab-software and more, see:
<http://bit.ly/AI2CS>.
Comments, corrections and suggestions, are appreciated; send by email to
amir.herzberg@uconn.edu.

©Amir Herzberg

Contents

Contents	ii
List of Figures	xi
List of Tables	xvi
List of Labs	xviii
List of Principles	xix
1 Introducing Cybersecurity and Cryptography	1
1.1 Cybersecurity and Cryptography: the basics	1
1.1.1 Three Cybersecurity Functions: Prevention, Detection and Deterrence	2
1.1.2 Generic Security Goals	4
1.1.3 Attack Model	5
1.1.4 Provable Security	6
1.1.5 Risk and costs-benefit analysis	8
1.2 The basic mechanisms: encryption, signatures and hashing	9
1.2.1 Encryption: symmetric and asymmetric cryptosystems .	9
1.2.2 Kerckhoffs' principle	11
1.2.3 Digital Signature schemes	13
1.2.4 Applying Signatures for Evidences and for Public Key Infrastructure (PKI)	15
1.2.5 Cryptographic hash functions	17
1.3 Background and notations	18
1.3.1 A bit of Computational Complexity	18
1.3.2 A bit of Number Theory and Group Theory	19
1.3.3 A bit of Probability	20
1.3.4 Sequence diagrams and the dot and key notations . . .	20
1.3.5 Notations	21
1.4 Provable-Security and Definitions	22
1.4.1 Definition of a Signature Scheme	24
1.4.2 Signature attack models and the conservative design prin- ciple	26

1.4.3	Types of forgery	28
1.4.4	Game-based Security and the Oracle Notation	29
1.4.5	The Existential Unforgeability CMA Game	30
1.4.6	The unforgeability advantage function, concrete/asymptotic security and negligible functions	31
1.4.7	33
1.5	A Brief History of Cryptography and Cybersecurity	34
1.5.1	A brief history of cryptography	35
1.5.2	The History of Computing and Cybersecurity	38
1.6	Lab and Additional Exercises	41
2	Confidentiality: Encryption Schemes and Pseudo-Randomness	45
2.1	Historical Ciphers	46
2.1.1	Ancient Keyless Ciphers	47
2.1.2	Generalized-Caesar cipher: a Keyed Variant of the Caesar Cipher	51
2.1.3	The General Monoalphabetic Substitution (GMS) Cipher	52
2.1.4	Frequency analysis attacks on monoalphabetic ciphers	53
2.1.5	The Polyalphabetic Vigenére ciphers	55
2.2	Cryptanalysis Attack Models: CTO, KPA, CPA and CCA	58
2.3	Generic attacks and Effective Key-Length	63
2.3.1	The exhaustive-search generic CTO attack	63
2.3.2	The Table Look-up and the Time-Memory Tradeoff Generic CPA attacks	65
2.3.3	Effective key length	65
2.4	Unconditional security and the One Time Pad (OTP)	67
2.4.1	OTP is a Stateful Cryptosystem / Stream Cipher	69
2.5	Pseudo-Randomness, Indistinguishability and Asymptotic Security	70
2.5.1	Pseudo-Random Generators and their use for Bounded Key-length Stream Ciphers	71
2.5.2	The Turing Indistinguishability Test	73
2.5.3	PRG indistinguishability test	73
2.5.4	Defining Secure Pseudo-Random Generator (PRG)	74
2.5.5	Secure PRG Constructions	76
2.5.6	RC4: Vulnerabilities and Attacks	79
2.5.7	Random functions	82
2.5.8	pseudorandom functions (PRFs)	86
2.5.9	PRF: Constructions and Robust Combiners	93
2.5.10	The key separation principle and application of PRF	94
2.6	Block Ciphers and PRPs	95
2.6.1	Random and Pseudo-Random Permutations	96
2.6.2	Security of block ciphers	100
2.6.3	The Feistel Construction: $2n$ -bit Block Cipher from n -bit PRF	101
2.7	Defining secure encryption	103
2.7.1	Attack model	104

2.7.2	The Indistinguishability-Test for Shared-Key Cryptosystems	105
2.7.3	The Indistinguishability-Test for Public-Key Cryptosystems (PKCs)	109
2.7.4	Design of Secure Encryption: the Cryptographic Building Blocks Principle	110
2.8	Encryption Modes of Operation	113
2.8.1	The Electronic Code Book mode (ECB) mode	115
2.8.2	The CTR and PBR modes	116
2.8.3	The Output-Feedback (OFB) Mode	119
2.8.4	The Cipher Feedback (CFB) Mode	123
2.8.5	The Cipher-Block Chaining (CBC) mode	125
2.8.6	Modes of Operation: Ensuring CCA Security?	127
2.9	Padding Schemes and Padding Oracle Attacks	127
2.10	Case study: the (in)security of WEP	131
2.10.1	CRC-then-XOR does not ensure integrity	133
2.11	Encryption: Final Words	134
2.12	Lab and Additional Exercises	136
3	Integrity: Cryptographic Hash Functions and Accumulators	151
3.1	Introducing cryptographic hash functions, properties and variants	152
3.1.1	Warm-up: hashing for efficiency	153
3.1.2	Properties of cryptographic hash functions	156
3.1.3	Applications of cryptographic hash functions	159
3.1.4	Standard cryptographic hash functions	160
3.2	Collision Resistant Hash Function (CRHF)	160
3.2.1	Keyless Collision Resistant Hash Function (Keyless-CRHF)	160
3.2.2	There are no Keyless CRHFs!	162
3.2.3	Keyed Collision Resistance	163
3.2.4	Birthday and exhaustive attacks on CRHFs	167
3.2.5	CRHF Applications (1): File Integrity	168
3.2.6	CRHF Applications (2): Hash-then-Sign (HtS)	169
3.3	Second-preimage resistance (SPR) Hash Functions	172
3.3.1	The Chosen-Prefix Collisions Vulnerability	175
3.4	One-Way Functions	178
3.4.1	Using OWF for One Time Passwords (OTPw) Authentication	179
3.4.2	Hash Chain Authentication	180
3.4.3	Using OWF for One-Time Signatures	181
3.5	Randomness Extraction and Key Derivation Functions	183
3.5.1	Von Neumann's Biased-Coin Extractor	184
3.5.2	The Bitwise Randomness Extractor	185
3.5.3	Key Derivation Functions	186
3.6	The Random Oracle Model	188
3.7	Cryptographic Accumulators	190
3.8	The Merkle-Damgård Accumulator	190
3.8.1	The collision resistant digest function	191

3.8.2	The Merkle-Damgård Construction of collision resistant digest function	191
3.8.3	The Extend Function and Validation of Entries and Extensions	195
3.9	The Merkle Accumulator	197
3.9.1	The Merkle digest scheme: Definitions	199
3.9.2	Extending the sequence: Proofs of Consistency	201
3.9.3	Merkle Digest scheme and Privacy	202
3.9.4	$2IM\mathcal{T}$: the Flat Merkle Tree construction	203
3.9.5	The Merkle tree $m\mathcal{T}$ construction	206
3.10	Blockchains, PoW and Bitcoin	209
3.10.1	The blockchain digest scheme	210
3.10.2	Controlled blockchains: permissioned and permissionless	212
3.10.3	Proof-of-Work (PoW) schemes	213
3.10.4	The Bitcoin Blockchain and Cryptocurrency	215
3.11	Lab and additional exercises	217
4	Authentication: MAC, Blockchain and Signature Schemes	223
4.1	Encryption for Authentication?	224
4.2	Message Authentication Code (MAC) schemes	225
4.3	Message Authentication Code (MAC): Definitions	227
4.4	Applying MAC Schemes	230
4.5	Constructing MAC from a Block Cipher	232
4.5.1	Every PRF is a MAC	233
4.5.2	CBC-MAC: ln -bit MAC (and PRF) from n -bit PRF	234
4.5.3	Constructing Secure VIL MAC from PRF	236
4.6	Other MAC Constructions	237
4.6.1	MAC design ‘from scratch’	237
4.6.2	Robust combiners for MAC	238
4.6.3	HMAC and other constructions of a MAC from a Hash function	239
4.7	Combining Authentication, Encryption and Other Functions	242
4.7.1	Authenticated Encryption (AE) and AEAD schemes	243
4.7.2	Authentication via EDC-then-Encryption?	245
4.7.3	Generic Authenticated Encryption Constructions	245
4.7.4	Single-Key Generic Authenticated-Encryption	249
4.7.5	Authentication, encryption, compression and error detection/correction codes	250
4.8	Additional exercises	253
5	Shared-Key Protocols	259
5.1	Modeling cryptographic protocols	260
5.1.1	The execution process	260
5.1.2	Interactions and interfaces	262
5.1.3	Adversary capabilities and security requirements	265
5.1.4	A simple EtA session/record protocol	266

5.2	Entity Authentication Protocols	270
5.2.1	Interactions and requirements of entity authentication protocols	270
5.2.2	Vulnerability case study: SNA mutual-authentication protocol	274
5.2.3	Secure Mutual Entity Authentication with the 2PP protocol	277
5.3	Authenticated Request-Response Protocols	278
5.3.1	Summary of request/response protocols	280
5.3.2	The 2PP-RR Authenticated Request-Response Protocol.	282
5.3.3	2RT-2PP Authenticated Request-Response protocol	283
5.3.4	Counter-based Authenticated Request-Response protocol	284
5.3.5	Time-based Authenticated Request-Response protocol	285
5.4	Shared-key Key Exchange Protocols	287
5.4.1	The Key Exchange extension of 2PP	288
5.4.2	Deriving Per-Goal Keys	289
5.5	Key Distribution Center Protocols	290
5.5.1	The Kerberos Key Distribution Protocol	291
5.6	The GSM Key Exchange Protocol	292
5.6.1	VN-impersonation Replay attack on GSM	296
5.6.2	Crypto-agility and cipher suite negotiation in GSM	298
5.6.3	The downgrade to A5/2 attack on GSM	301
5.7	Resiliency to Exposure: Forward Secrecy and Recover Security	306
5.7.1	Forward Secrecy 2PP Key Exchange	307
5.7.2	Recover-Security Key Exchange Protocol	309
5.7.3	Stronger notions of resiliency to key exposure	310
5.8	Additional Exercises	313
6	Public Key Cryptography	319
6.1	Introduction to PKC	320
6.1.1	Public key cryptosystems	320
6.1.2	Signature schemes	321
6.1.3	Public-Key-based Key Exchange Protocols	321
6.1.4	Advantages of Public Key Cryptography (PKC)	324
6.1.5	The price of PKC: assumptions, computation costs and length of keys and outputs	325
6.1.6	Hybrid Encryption	329
6.1.7	The Factoring and Discrete Logarithm Hard Problems	331
6.1.8	The secrecy implied by the discrete logarithm assumption	333
6.2	The DH Key Exchange Protocol	335
6.2.1	Physical key exchange	335
6.2.2	Some candidate key exchange protocols	337
6.2.3	The Diffie-Hellman Key Exchange Protocol and Hardness Assumptions	341
6.2.4	Secure derivation of keys from the DH protocol	344
6.3	Using DH for Resiliency to Exposures	346

6.3.1	The Authenticated DH protocol: ensuring PFS	346
6.3.2	The DH-Ratchet protocol: Perfect Forward Secrecy (PFS) and Perfect Recover Security (PRS)	348
6.4	The DH and El-Gamal PKCs	351
6.4.1	The DH PKC and the Hashed DH PKC	351
6.4.2	The El-Gamal PKC	354
6.4.3	El-Gamal is Multiplicative-Homomorphic Encryption . .	356
6.4.4	Types and Applications of Homomorphic Encryption . .	357
6.5	The RSA Public-Key Cryptosystem	360
6.5.1	RSA key generation.	361
6.5.2	Textbook RSA: encryption, decryption, and signing. . .	361
6.5.3	Efficiency of RSA	363
6.5.4	Correctness of RSA	364
6.5.5	The RSA assumption and the vulnerability of textbook RSA	366
6.5.6	Padded RSA encryption: PKCS#1, v1.5 and OAEP . .	368
6.5.7	Bleichenbacher's Padding Side Channel Attack on PKCS#1 v1.5	372
6.6	Public key signature schemes	377
6.6.1	RSA-based signatures	379
6.7	Labs and Additional Exercises	382
7	TLS/SSL protocols: web-security and beyond	391
7.1	Introduction to TLS and SSL	395
7.1.1	TLS/SSL: High-level Overview	395
7.1.2	TLS/SSL: security goals	397
7.1.3	SSL/TLS: Engineering goals	399
7.1.4	TLS/SSL and the TCP/IP Protocol Stack	400
7.2	The SSL/TLS Record Protocol	400
7.2.1	The Authenticate-then-Encrypt (AtE) Record Protocol	402
7.2.2	The CPA-Oracle Attack Model	406
7.2.3	Padding Attacks: Poodle and Lucky13	407
7.2.4	The BEAST Attack: Exploiting CBC with Predictable-IV	411
7.2.5	Exploiting RC4 Biases to Recover Plaintext	414
7.2.6	Exploiting Compress-then-Encrypt: CRIME, TIME, BREACH	415
7.2.7	The TLS AEAD-based record protocol (TLS 1.3)	417
7.3	The SSLv2 Handshake Protocol	419
7.3.1	SSLv2: the ‘basic’ handshake	420
7.3.2	SSLv2 Key-derivation	422
7.3.3	SSLv2: <i>ID</i> -based Session Resumption	423
7.3.4	SSLv2: Client Authentication	424
7.4	The Handshake Protocol: from SSLv3 to TLSv1.2	425
7.4.1	SSLv3 to TLSv1.2: improved derivation of keys	426
7.4.2	SSLv3 to TLSv1.2: DH-based key exchange	428
7.4.3	The TLS Extensions mechanism	430
7.4.4	SSLv3 to TLSv1.2: session resumption	431

7.4.5	SSLv3 to TLSv1.2: Client authentication	435
7.5	Negotiations and Downgrade Attacks (SSL to TLS 1.2)	436
7.5.1	SSLv2 cipher suite negotiation and downgrade attack . .	436
7.5.2	Handshake Integrity Against Cipher Suite Downgrade .	438
7.5.3	Finished Fails: the Logjam and FREAK cipher suite downgrade attacks	440
7.5.4	Backward compatibility and protocol version negotiation	443
7.5.5	The TLS Downgrade Dance and the Poodle Version Downgrade Attack	445
7.5.6	Securing the TLS downgrade dance: the SCSV cipher suite and beyond	446
7.5.7	The SSL-stripping Attack and the HSTS Defense	446
7.5.8	Three Principles: Secure Extensibility, KISS and Minimize Attack Surface	449
7.6	The TLS 1.3 Handshake: Improved Security and Performance .	450
7.6.1	TLS 1.3: Negotiation and Backward Compatibility . . .	453
7.6.2	TLS 1.3 Full (1-RTT) DH Handshake	456
7.6.3	TLS 1.3 Full (1-RTT) Pre-Shared Key (PSK) Handshake	458
7.6.4	TLS 1.3 Zero-RTT Pre-Shared Key (PSK) Handshake .	461
7.6.5	TLS 1.3 Key Derivation	462
7.6.6	Cross-Protocol Attacks on TLS 1.3	463
7.7	SSL/TLS: Final Words and Further Reading	464
7.8	Additional Exercises	465
8	Public Key Infrastructure (PKI)	473
8.1	Introduction: PKI Concepts and Goals	474
8.1.1	Rogue certificates	477
8.1.2	Security goals of PKI schemes.	478
8.1.3	The Web PKI	479
8.2	The X.509 PKI	480
8.2.1	The X.500 Global Directory Standard	480
8.2.2	The X.500 Distinguished Name	481
8.2.3	X.509 Public Key Certificates	485
8.2.4	The X.509v3 Extensions Mechanism	489
8.2.5	Trust-Anchor Certificate Validation	492
8.2.6	The SubjectAltName and the IssuerAltName Extensions	493
8.2.7	Standard key-usage and policy extensions	494
8.2.8	Certificate policy (CP) and Domain/Organization/Extended Validation	495
8.3	Intermediate-CAs and Certificate Path Validation	497
8.3.1	The certificate path constraints extensions	498
8.3.2	The basic constraints extension	499
8.3.3	The name constraint extension	500
8.3.4	The policy constraints extension	504
8.4	Certificate Revocation	504
8.4.1	Certificate Revocation List (<i>CRL</i>)	508

8.4.2	Online Certificate Status Protocol (OCSP)	511
8.4.3	OCSP Stapling and the Must-Staple Extension	518
8.4.4	Reducing OCSP Computational Overhead	525
8.4.5	Optimized Periodic Revocation Status: OneCRL, CRLsets and CRVs	529
8.5	Web PKI: Vulnerabilities and Improvements	530
8.5.1	Web PKI Certificate Authority Failures	530
8.5.2	X.509/PKIX Defenses against Corrupt/Negligent CAs .	533
8.6	Certificate Transparency (CT)	536
8.6.1	CT: concepts, entities and goals	537
8.6.2	The Honest-Logger Certificate Transparency (HL-CT) design	540
8.6.3	The Audit-and-Gossip Certificate Transparency (AnG- CT) design	544
8.6.4	The NTTP-Security Certificate Transparency (NS-CT) design	550
8.7	Additional Exercises	554
9	Human-centered Cryptography	565
9.1	Login Ceremonies	567
9.1.1	Password login ceremonies and attack-vectors	569
9.1.2	Post-compromise password security: Hashing, Salting and More	571
9.1.3	Password-Authenticated Key Exchange (PAKE)	572
9.1.4	Password Managers	572
9.1.5	Something you have login ceremonies	572
9.1.6	Biometrics-based login ceremonies	573
9.1.7	Two-Factor Authentication	573
9.2	Phishing attacks and defenses	573
9.3	Challenges of Usable-Security Research and Development . .	573
9.3.1	Secure Usability: Can Johnny Encrypt?	573
9.3.2	Human Public Key Validation and Security of End-to-End Secure Messaging	573
9.4	Lab and Additional Exercises	573
10	Conclusions and few advanced topics	577
10.1	Secret sharing and its Applications	577
10.2	Side-channels	577
10.3	Elliptic Curves Cryptography	577
10.4	Quantum and post-quantum cryptography: by Walter Krawec .	577
10.4.1	Quantum cryptanalysis and post-quantum cryptography	579
10.4.2	Quantum Cryptography	581
10.5	Privacy and anonymity	583
10.6	Theory of cryptography	583
Appendix A	Background	585

A.1	Background: Computational Complexity	585
A.2	Background: Number Theory and Group Theory	590
A.2.1	The modulo operation and modular arithmetic	590
A.2.2	Multiplicative inverses	592
A.2.3	Fermat's and Euler's Theorems	594
A.2.4	Group Theory, Cyclic Groups and Generators	598
A.3	Background: Probability	600
Index		607
Bibliography		621

List of Figures

1.1	Cybersecurity defense approaches: Prevention, Detection and Deterrence.	2
1.2	Generic Security Goals: Confidentiality, Integrity, Authentication, Availability and Non-repudiation	4
1.3	Encryption: terms and typical use	9
1.4	Shared key (symmetric) cryptosystem.	10
1.5	Public key (asymmetric) cryptosystem.	10
1.6	Digital Signature Scheme.	14
1.7	Sequence diagram for the initialization and use of a signature scheme	22
2.1	Stateful shared key (symmetric) cryptosystem.	46
2.2	The At-Bash Cipher.	48
2.3	The AzBy, Caesar and ROT13 Ciphers.	49
2.4	The Masonic Cipher	50
2.5	Letter and Bigram Frequencies in English	53
2.6	The Ciphertext-Only (CTO) attack model	59
2.7	The Known-Plaintext Attack (KPA) model	60
2.8	The Chosen-Plaintext Attack (CPA) model	61
2.9	The Chosen-Ciphertext Attack (CCA) model	62
2.10	The One Time Pad (OTP) cipher	68
2.11	PRG-based Stream Cipher	72
2.12	The Turing Indistinguishability Test	73
2.13	Intuition for the PRG Indistinguishability Test	74
2.14	Feedback Shift Register	79
2.15	Bit-wise encryption based on a random function	84
2.16	Block (n -bits) encryption using a Random Function $f(\cdot)$	86
2.17	Using PRF for secure encryption	87
2.18	The PRF Indistinguishability Test	88
2.19	Standard block ciphers (AES and DES)	96
2.20	The PRP Indistinguishability Test	98
2.21	Three ‘rounds’ of the Feistel Cipher	103
2.22	The IND-CPA test for stateless shared-key encryption	107
2.23	The IND-CPA test for symmetric encryption (E, D)	107
2.24	IND-CPA-PK, indistinguishability test for public-key encryption .	110
2.25	Electronic Code Book (ECB) mode encryption	115

2.26	Electronic Code Book (ECB) mode decryption	115
2.27	Visual demonstration of the weakness of the ECB mode	116
2.28	Per-Block Random (PBR) mode encryption	117
2.29	Counter (CTR) mode encryption	119
2.30	Output Feedback (OFB) mode encryption	120
2.31	Output Feedback (OFB) mode decryption. Adapted from [181] . . .	120
2.32	Cipher Feedback (CFB) mode encryption	124
2.33	Cipher Feedback (CFB) mode decryption	124
2.34	Cipher Block Chaining (CBC) mode encryption	125
2.35	Cipher Block Chaining (CBC) mode decryption	126
2.36	The Padding Oracle Attack model	129
2.37	A single round of the ANSI X9.31 stateful PRG	143
3.1	Keyless and Keyed Hash Functions	152
3.2	Load-balancing with (keyless) hash function $h(\cdot)$	154
3.3	Algorithmic Complexity Denial-of-Service Attack	155
3.4	Load balancing with a collision-resistant hash function	156
3.5	Keyless collision resistant hash function (CRHF)	161
3.6	Keyed collision resistance hash function (CRHF)	164
3.7	Target collision resistant (TCR) hash function	165
3.8	Use of hash function h to validate integrity of file	168
3.9	Second-preimage resistance (SPR)	173
3.10	One-Way Function (aka Preimage-Resistance)	178
3.11	A one-time signature scheme, limited to a single bit	182
3.12	A one-time signature scheme, for l -bit string (denoted d)	183
3.13	A one-time signature scheme using ‘Hash-then-Sign’	183
3.14	Bitwise-Randomness Extractor (BRE) Hash Function	185
3.15	The Merkle-Damgård Construction of a collision-resistance digest function from a CRHF	192
3.16	Compression function: fixed-length input to shorter-length output	194
3.17	The keyless <i>Flat Merkle Tree</i> ($2lMT$) construction	203
3.18	The <i>keyed</i> Flat Merkle Tree ($2lMT$) construction	203
3.19	The <i>Merkle Tree</i> (MT) construction	206
3.20	Example of Proof-of-Inclusion (PoI) in the <i>Merkle Tree</i> (MT) . . .	207
3.21	A basic blockchain	211
3.22	Permissioned Blockchain	213
3.23	Bitcoin Blockchain, using Proof-of-Work (PoW) and Mining	213
4.1	Using a MAC scheme to authenticate messages	226
4.2	The CBC-MAC construction	235
4.3	Combining Encryption, Authentication, Reliability and Compression	251
5.1	The Two-Party MitM Execution Model	261
5.2	Interactions for the record/session protocols	264
5.3	Interactions for entity authentication protocols	271
5.4	The (vulnerable) SNA mutual authentication protocol.	274

5.5	Attack on SNA Mutual Entity Authentication protocol	276
5.6	The 2PP Mutual Entity Authentication protocol.	278
5.7	Interactions for Authenticated Request-Response Protocols	279
5.8	The 2PP-RR protocol: a three flow nonce-based authenticated Request-Response protocol, based on 2PP	283
5.9	The 2RT-2PP Authenticated Request-Response protocol	283
5.10	Counter-based Authenticated Request-Response protocol	284
5.11	Time-based Authenticated Request-Response protocol	286
5.12	The 2PP Key Exchange protocol	289
5.13	The Kerberos Key Distribution Center Protocol	291
5.14	The GSM Key Exchange Protocol	294
5.15	The VN-impersonation attack on GSM	297
5.16	The GSM Key Exchange Protocol with cipher suite negotiation . .	300
5.17	Simplified downgrade attack on GSM Key Exchange.	304
5.18	A ‘real’ downgrade attack on GSM Key Exchange.	305
5.19	The Forward-Secrecy 2PP Key Exchange protocol	308
5.20	Result of running the Forward-Secrecy 2PP Key Exchange for three periods	308
5.21	Running the <i>recover-security Key Exchange protocol</i> for five periods	309
5.22	Relations between notions of resiliency to key exposures	312
5.23	Simplified SSL	314
6.1	The discovery of Public-Key Cryptography	320
6.2	Operation of two-flows key exchange protocols	322
6.3	Hybrid encryption	330
6.4	Physical Key Exchange Protocol	336
6.5	The (insecure) XOR Key Exchange Protocol	337
6.6	The (insecure) Exponentiation Key Exchange Protocol	338
6.7	The Modular-Exponentiation Key Exchange Protocol	340
6.8	The Diffie-Hellman Key Exchange Protocol	341
6.9	MitM attack on the DH key-exchange protocol	343
6.10	The Generalized Diffie-Hellman Key Exchange Protocol	345
6.11	The Auth- h -DH protocol	347
6.12	The DH-Ratchet protocol	349
6.13	The DH public key cryptosystem	352
6.14	The Hashed DH public key cryptosystem	353
6.15	The El-Gamal Public-Key Cryptosystem using DDH group	355
6.16	Privacy-preserving voting using homomorphic encryption	359
6.17	RSA encryption: textbook RSA (vulnerable) vs. padded RSA . . .	362
6.18	Simplified OAEP padding	371
6.19	OAEP padding	373
6.20	The chosen-ciphertext side-channel attack (CCSCA) model	374
6.21	Bleichenbacher’s attack on RSA	375
6.22	Public key certificate issuing and usage processes.	378
6.23	How <i>not</i> to ensure resilient key exchange: illustration for Ex. 6.20	386

6.24	Insecure ‘robust-combiner’ authenticated DH protocol, studied in Exercise 6.21.	386
6.25	Insecure variant of the DH-Ratchet Protocol, for Ex. 6.26.	387
7.1	Simplified overview of the operation of SSL/TLS	396
7.2	Placement of TLS/SSL in the TCP/IP protocol stack	400
7.3	Phases of TLS/SSL connection	401
7.4	The Authenticate-then-Encrypt (AtE) SSL/TLS record protocol .	403
7.5	Padding in the SSL/TLS record protocol	405
7.6	The CPA-Oracle Attack model	407
7.7	The AEAD Record Protocol (TLS 1.3)	418
7.8	‘Basic’ SSLv2 handshake	421
7.9	SSLv2 handshake, with <i>ID</i> -based session resumption	424
7.10	The ‘basic’ RSA-based handshake, from SSLv3 till TLS 1.2 . . .	425
7.11	SSLv3 to TLSv1.2: static DH handshake	429
7.12	SSLv3 to TLSv1.2	430
7.13	SSLv3 to TLS1.2 handshake, with <i>ID</i> -based session resumption. .	432
7.14	Ticket-based session resumption	434
7.15	Client authentication in SSLv3 to TLS1.2.	435
7.16	SSLv2 handshake, with details of cipher suite negotiation	437
7.17	Example of SSLv2 cipher suite negotiation. In this example, the client offers three cipher suites, and the server supports two of these. The negotiation was changed from SSLv3.	437
7.18	Cipher suite downgrade attack on SSLv2	438
7.19	The Logjam cipher suite downgrade attack against servers supporting the exportable (weak keys) version of the DH Ephemeral (DHE) key exchange. The attack works for the (surprisingly common) case of known DH group (p, g) , allowing the attacker to precompute the most computationally challenging part of the discrete log computation and use it for the different exponents and handshakes. The MitM attack begins once this precomputation is done. The attacker forces TLS clients to use export-strength Diffie-Hellman Ephemeral (DHE) key exchange (the DHE_Export cipher suite). The attacker modifies Client Hello to request DHE_Export from the server, and modifies Server Hello to appear as if the server uses regular DHE. The client does not detect that the Diffie-Hellman values (p, g, g^b) correspond to export-version of DH, and continues the handshake, sending $g^a \bmod p$ and then Client Finished. The attacker now uses the precomputed values to compute $bb \leftarrow DiscLog(g^b \bmod p)$, allowing it to find the master key k_M and complete the handshake.	442
7.20	The SSL-stripping Attack	447
7.21	TLS 1.3 1-RTT full Diffie-Hellman handshake	456
7.22	TLS 1.3 Full (1-RTT) Pre-Shared Key (PSK) handshake	458
7.23	TLS 1.3 Zero-RTT Pre-Shared Key (PSK) Handshake	461
8.1	PKI Entities and typical application for Web PKI	476

8.2	Example of the X.500 (and X.509) Distinguished Name (DN) Hierarchy.	482
8.3	The Identifiers Trilemma: the challenge of co-ensuring unique, de-centralized and meaningful identifiers.	485
8.4	X.509 version 1 certificate. Note the fields added in later versions, mainly version 3 of X.509 (Figure 8.5), most notably the extensions field.	486
8.5	X.509 version 3 certificate	488
8.6	A single-hop (length one) certificate-path	501
8.7	A three-hop certificate-path	502
8.8	Example of the use of Name Constraint	503
8.9	Example of the use of the <i>name constraint</i> extension, with similar constraints to the ones in Figure 8.8, but here using DNS names (dNSName).	504
8.10	X.509 Certificate Revocation List (<i>CRL</i>)	508
8.11	The Online Certificate Status Protocol (OCSP)	511
8.12	OCSP used by relying party	512
8.13	The MitM soft-fail Attack on a TLS connection using OCSP	516
8.14	OCSP stapling in the TLS protocol, using the CSR TLS extension	520
8.15	MitM soft-fail attack on OCSP-stapling TLS client (browser)	522
8.16	Use of the <i>TLS-feature</i> X.509 extension [155], indicating <i>Must-Staple</i> , against the <i>MitM soft-fail</i> attack on OCSP-stapling	523
8.17	Certificates-Merkle-tree variant of OCSP	526
8.18	Signed revocations-status Merkle-tree	527
8.19	Optimizing OCSP response, using tree-of-revocations and Proof-of-Non-Inclusion	528
8.20	The issue process for both HL-CT and AnG-CT	541
8.21	Monitoring in HL-CT	542
8.22	Omitted-certificate attack by a rogue logger and rogue CA	543
8.23	Split-world attack on AnG-CT	547
8.24	Zombie-certificate attack	549
8.25	NS-CT issue process, with a (possibly corrupt) logger	551
8.26	NS-CT detection by monitor of a misleading certificate	553
8.27	NTTP-Security Certificate Transparency (NS-CT), no-faults scenario.	554
8.28	NS-CT defending against the omitted-certificate attack, by providing a Proof-of-Misbehavior of a rogue logger	555
8.29	Split-world attack by a rogue logger against NS-CT (incorrectly) deployed without gossip	556
8.30	Inter-monitor gossip	556
9.1	Attack vectors on password login ceremony.	570

List of Tables

1.1	Notations used in this manuscript	23
2.1	Cryptanalysis Attack Models	58
2.2	Table for Exercise 2.10	83
2.3	Do-it-yourself table for selecting random permutations ρ_1, ρ_2 over domain $D = \{0, 1\}^2$	97
2.4	Comparison between random function, random permutation, PRG, PRF, PRP and block-cipher	100
2.5	Encryption Modes of Operation using n -bit block cipher	113
2.6	Ciphertexts for Exercise 2.25	137
3.1	Goals and Requirements for keyless cryptographic hash functions .	157
3.2	Comparison: PRF, KDF and Extractor functions	187
4.1	Authentication schemes: MAC, Authenticated Encryption (AE, AEAD) and Signatures	244
5.1	Authenticated Request-Response (RR) protocols.	282
5.2	Notions of resiliency to key exposures of key-setup Key Exchange protocols	311
6.1	Key length and computing time for asymmetric and symmetric cryptography	327
6.2	Resiliency to key exposures of Key Exchange protocols	349
7.1	Derivation of connection keys and IVs, in SSLv3 to TLS1.2 . . .	428
7.2	Some important TLS/SSL attacks due to specifications vulnerabilities, 2011-2019.	464
7.3	Table for Exercise 7.15.	470
8.1	Standard keywords/attributes in X.500 Distinguished Names . . .	482
8.2	Standard certificate policies	496
8.3	Comparison of revocation-checking mechanisms	506
8.4	Revocation-related parameters	506
8.5	Notable Web PKI Certificate Authority Failures	531

A.1 Euler's function $\phi(n)$	595
--	-----

List of Labs

Lab 1 (Using cryptography to validate downloads)	41
Lab 2 (Ransomware and Encryption)	136
Lab 3 (CRC-hash Collisions)	217
Lab 4 (Breaking textbook and weakly-padded RSA)	382
Lab 5 (Password cracking)	573

List of Principles

Principle 1 (Attack model principle: assume capabilities, not strategy)	5
Principle 2 (Kerckhoffs' principle)	11
Principle 3 (Conservative design and usage)	27
Principle 4 (Limit usage of each key)	54
Principle 5 (Sufficient effective key length)	67
Principle 6 (Random function design)	87
Principle 7 (Key separation)	94
Principle 8 (Cryptographic Building Blocks)	111
Principle 9 (Minimize plaintext redundancy)	123
Principle 10 (Key Separation)	231
Principle 11 (Crypto-agility)	298
Principle 12 (Minimize use of public-key cryptography)	329
Principle 13 (Secure extensibility by design)	449
Principle 14 (The KISS Principle)	449
Principle 15 (Minimize the attack surface)	450
Principle 16 (The UX>Security Precedence Rule)	517
Principle 17 (Soft-fail security is insecure)	521
Principle 18 (Bellovin's principle: secure means used securely)	566

Chapter 1

Introducing Cybersecurity and Cryptography

Cybersecurity and cryptography are exciting topics, with rich history and with an extensive impact on the practical world, as well as fascinating challenges of engineering, theory and mathematics. In this textbook, our goal is to introduce cybersecurity, focusing on the area of *applied cryptography*, which is essential to most or all areas of cybersecurity. We believe that this basic knowledge in applied cryptography is desirable for any student of computer science, but especially for students which plan to focus on any area of cybersecurity or, of course, students planning to focus on cryptography.

This chapter provides several sections with important foundations to the rest of this book. The first section (Section 1.1) introduces the areas of Cybersecurity and Cryptography, and their goals, approaches and few basic principles. Section 1.2 provides an informal introduction to two basic cryptographic mechanisms, *encryption* and *signatures*. Section 1.3 provides essential *background and notation*, used throughout the text; additional background is provided in Appendix A. In section 1.4 we discuss the challenge of *defining security* of cryptographic schemes, focusing on the important definition of security for signature schemes. Finally, Section 1.5 provides a few gems from the fascinating history of *cryptography* and *cybersecurity*, which, we believe, may provide an interesting perspective.

1.1 Cybersecurity and Cryptography: the basics

The high-level goal of cybersecurity is to protect ‘good’ parties, who use computing and communicating systems legitimately, from damages due to *misbehaving* entities, to whom we usually refer as *attackers* or *adversaries*), who abuse the systems to harm the users and/or to obtain an unfair advantage or benefit. Attackers could be rogue authorized users, often referred to as *insiders*, or *outsiders*, which only have some access to the systems or the communication, and are not authorized users. Attackers may also control one or multiple de-

vices. Such devices may be legitimately owned by the attacker, or be *corrupted*, i.e., controlled by the attacker in spite of not being legitimately owned by the attackers; corrupted devices are often referred to as being *pwned*¹ by the attacker.

Unfortunately, it is not so easy to turn this high-level goal into precise *definitions of security*, allowing evaluation of the security of different systems and defenses. It is also challenging to design schemes that satisfy these definitions, and to *prove* their security. In this section, we discuss the main approaches to ensure and define security. But first, in subsection 1.1.1, we introduce the three *cybersecurity functions*: prevention, detection and deterrence.

1.1.1 Three Cybersecurity Functions: Prevention, Detection and Deterrence

Figure 1.1 illustrates the *three cybersecurity functions*, i.e., basic ways of protecting against attackers: *prevention*, *detection* and *deterrence*.

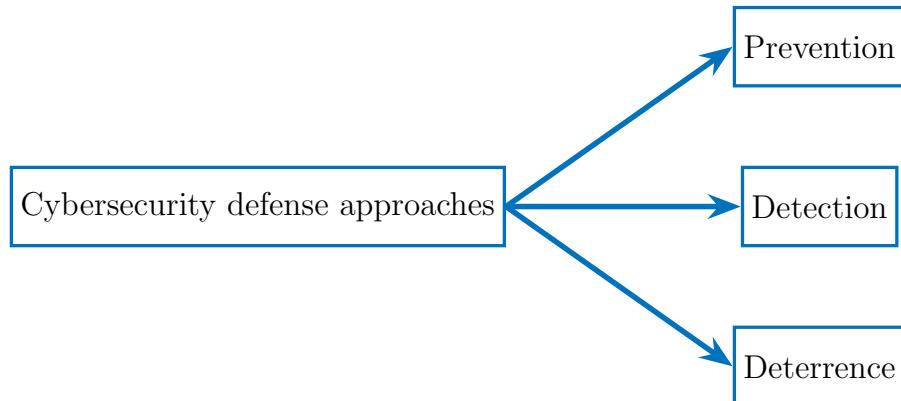


Figure 1.1: Cybersecurity defense approaches: Prevention, Detection and Deterrence.

Prevention: mechanisms that *prevent* an attacker from causing damage, or that reduce the amount of possible damage. *Encryption* is an example of a cryptographic prevention mechanism; it is usually used to *prevent* an attacker from disclosing sensitive information. When possible, prevention is obviously the best defense - ‘an ounce of prevention is worth a pound of cure’. However, sometimes it is impossible to completely prevent an attack. For example, the TLS protocol (Chapter 7) relies on *public key certificates* issued

¹The verb ‘pwn’ means to control or ‘own’ a computing system illegitimately, by exploiting a vulnerability. The verb ‘pwn’ is taken from gamers slang, where it was originally used for a player which is completely dominated by an opponent; its origin may come from chess and combine the chess ‘pawn’ with the verb ‘own’. See [257].

and signed by a trusted party, called a *Certificate Authority* (CA); attacks due to a mis-behaving CA cannot be completely prevented. Instead, such attacks are mitigated using detection mechanisms - and deterrence, with significant penalties for misbehaving CAs, as we discuss in Chapter 8. Even when a cryptographic mechanism seems to completely prevent an attack, e.g., when using encryption, it is still prudent to also deploy detection mechanisms, since hidden, subtle vulnerabilities may exist even in thoroughly reviewed designs and systems.

Detection: mechanisms that *detect* an attack, usually while the attack is ongoing. Detection allows deployment of additional defenses, which are not deployed otherwise (e.g., due to costs). Detection is a major component of important, widely deployed network-security and host-security defenses such as *Intrusion Detection Systems (IDS)*. Detection is not as much applied in cryptography, however, we present some examples, e.g., the PKI detection mechanisms discussed in Chapter 8 (and mentioned above). Another example is the combination of an *Error Detection Code* and a *Message Authentication Code*, used to detect attempt to attack authenticated communication, as illustrated in Figure 4.3.

Deterrence is effective if the adversary is *rational*, and would refrain from attacking if her expected profit (from attack) would be less than the expected penalty. In practice, most attackers *are* rational, hence, good deterrence is an effective defense. However, a challenge is that attackers do their best to *avoid being detected* and penalized; deterrence is only effective, when combined with strong penalties - and *effective detection*. Indeed, *detection* is often a key element in ensuring security. First, detection is required to penalize attackers and hence effective detection is key to deter misbehavior; second, detection allows *reaction*, such as using additional defense mechanisms and performing operations to recover security and minimize damages. Therefore, security systems often invest a lot in *detection*, while attackers usually do their best to *avoid detection*, including refraining from attacks and actions that may result in detection.

Deterrence: mechanisms designed to discourage (deter) attackers from attacking. Deterrence is achieved either by the (visible) use of strong defenses, making it futile to attack, or by *penalizing* misbehaving entities. Penalizing requires *attribution* of the attack or misbehavior to the correct entity. *digital signature* schemes are an important cryptographic deterrence mechanism, which we discuss later in this chapter (subsection 1.2.3) and, in more details, in chapters 4 and 6. A signature verified using the misbehaving party's well-known *public key*, over a given message, provides evidence that this party signed that message; we refer to such evidence of abuse as *Proof of Misbehavior (PoM)*. PoM can be used to punish or penalize the misbehaving party in different ways - an important deterrent. Note, however, that deterrence can only be effective against a *rational adversary*; a penalty may fail to deter an irrational adversary, e.g., a terrorist.

1.1.2 Generic Security Goals

Different systems and schemes often have different security goals; however, some goals are *generic*, and apply, possibly with some variations on details, to many systems and schemes. The generic goals, illustrated in Figure 1.2, include *Confidentiality*, *Integrity*, *authenticity*, and *Availability* and *Non-repudiation*. Three of them (confidentiality, integrity, and either authentication or availability) are often referred to as the *CIA triad*, an easily-memorable acronym.

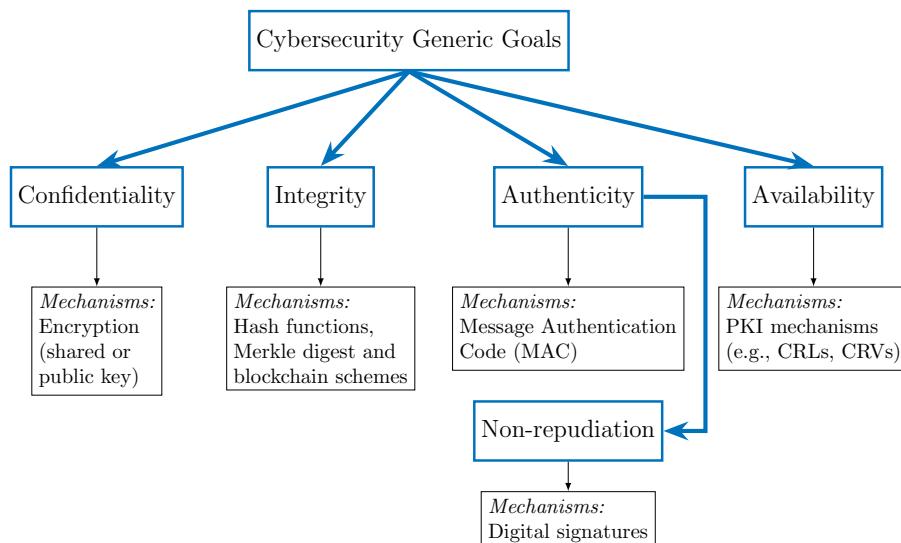


Figure 1.2: The Generic Security Goals: Confidentiality, Integrity, Authentication, Availability and Non-repudiation (which extends authentication). The first three are sometimes referred to as the *CIA triad*. Non-repudiation is an extension of authentication. For each goal, we list corresponding cryptographic mechanisms covered in this textbook.

Confidentiality. A system satisfies the *confidentiality* goal, if it prevents an attacker from disclosing some information defined as *confidential*. The ‘classical’ confidentiality mechanism is *encryption*, illustrated in Figure 1.4, with two main variants: *shared key cryptosystems*, also referred to as symmetric encryption, and *public key cryptosystems*, also referred to as asymmetric encryption.

Integrity. Ensuring integrity means prevention or detection of unauthorized operations, such as the modification of data. Integrity is applied to a wide range of situations, including the integrity of a computer system, information stored or transmitted, and more. We cover several cryptographic integrity mechanisms in Chapter 3, including *cryptographic hash functions*, *Merkle digest schemes* and *blockchains*.

Authenticity and Non-repudiation. Authentication mechanisms validate that a particular information was originated from a specific entity, or that a particular interaction involved a specific entity. A unique variant of authentication is *non-repudiation*, which provides *evidence* of the origin of information, that can be presented, later, to a third party, to ‘prove’ the identity of the origin. We cover the *Message Authentication Codes* (MAC) schemes which provides authentication, and *signature schemes*, which provides non-repudiation (and authentication).

Availability. Availability mechanisms ensure that services can be provided efficiently, even if an attacker tries to disrupt services, in what is called *Denial-of-Service (DoS)* attacks. DoS attacks have become a major concern for network and service providers, and defenses against them are a major challenge of network security; however, there is only limited use of cryptography schemes in defenses against DoS. In fact, some cryptographic protocols are *vulnerable* to DoS, i.e., a DoS attack may disrupt the security service, potentially resulting in a vulnerability. We discuss this concern for *Public Key Infrastructure (PKI)*, in Chapter 8. We explain how some PKI designs, such as CRLs and CRVs, are resistant to such DoS attacks, while others, e.g., OCSP, may be vulnerable or even abused to create a DoS attack.

Bespoke security goals. The generic goals apply to many cybersecurity systems. The security of specific systems usually include these generic goals, possibly with some adaptation, as well as addition, system-specific goals.

1.1.3 Attack Model

Security goals should be defined and evaluated with respect to the *capabilities* of the attackers, rather than by assuming a specific attacker *strategy*. This is an important principle of modern cryptology, which already appears in [101]: understand and define a clear model of the *attacker capabilities*, and then define goals/requirements for the scheme/system - against attackers with the specified capabilities, regardless of the attacker’s strategy (choices). This principle applies not only in cryptology, but in general in security. Precise articulation of the *attack model* and of the *security requirements* is fundamental to the design and analysis of security. Let us define this important principle.

Principle 1 (Attack model principle: assume capabilities, not strategy). *Security requirements should be defined and analyzed with respect to a well defined attack model, which specifies any restrictions of the capabilities of the attacker. The attack model should not restrict the attacker’s strategy.*

The attack model *principle* applies even in areas of cybersecurity where it is hard to define a rigorous attack model. Even in such scenarios, it is important to limit our assumptions to the attacker capabilities, rather than assuming a specific attacker strategy. However, where possible, a well-defined attack model,

allowing *provable security*, is better - is this is done ‘correctly’, as we discuss next.

1.1.4 Provable Security

In most areas of science and engineering, design and evaluation are based on experimental analysis, measuring the expected outcomes of the system under *typical* scenarios. In contrast, security should be ensured against an *adversary* (attacker), who is not bound to behave in some typical way; new attacks may be very different from past attacks. Observing the behavior of adversaries and designing defenses based solely on these behaviors may leave subtle or even gaping vulnerabilities that can be exploited by adversaries. That said, when a history of previous attacks is available and shows a clear pattern, it does make sense to evaluate defenses against the same type of attacks - *in addition* to evaluation against arbitrary attacker strategy, as per Principle 1.

It is challenging to ensure security against an arbitrary attacker strategy, only limiting the attacker’s capabilities. There are many subtle ways in which our intuition and imprecise arguments may fail, resulting in vulnerabilities. Cryptography and security are exceptions to the popular saying ‘*in theory there is no difference between theory and practice, while in practice there is*’ [69]. In fact, in security, and especially in cryptography, precise definitions and proofs of security, or at least extensive, clear analysis, are necessary to ensure that a system is secure against *arbitrary* attacker strategies. This approach is usually referred to as *provable security*.

That said, one has to be very careful to understand the *limitations and pitfalls of proofs*, discussed in a series of papers by Koblitz and Menezes, summarized in [196]. Proofs of security often involve simplifications and assumptions, even simplifying ‘assumptions’ known to be incorrect. An important example of such simplification is the *Random Oracle Model (ROM)*, discussed in Section 3.6. Using the ROM, a protocol using a specific cryptographic hash function, e.g., SHA-1, is analyzed as if the function used was selected at random. Clearly, this is not a correct assumption; yet, a proof using such assumption, gives *some* indication of security, or at least, limits the possible types of attacks.

Simplifications can result in a system has a proof of security - leading people to trust its security - however, in reality, this system is vulnerable and exploited. One reason this happens is that proofs are based on models of the attacker capabilities and of the system, which often do not fully capture reality; attacks often cleverly abuse and exploit exactly these aspects which were glossed-over and abstracted away. Furthermore, efforts to consider more realistic aspects tend to make analysis, proofs, and provably-secure designs, harder and more complicated. As a result, even for an expert, proofs may be challenging and require extensive effort - to write and to validate. Complex proofs may have subtle errors, which may remain undiscovered for years. *Proofs should be clean and enlightening, while reality is dirty and murky; and viruses, bugs, and vulnerabilities thrive in the dirt and the dark.*

Another challenge for provable security, is to correctly identify and rigorously define the *security requirements (goals)*. Rigorous definitions of security requirements are often challenging - to define and to understand. However, clear definitions of security requirements are important, for several reasons:

1. To prove the security of the cryptographic scheme.
2. To prove the security of an application of the scheme, e.g., as part of another cryptographic scheme or of a cryptographic protocol.
3. To allow researchers to explore attacks against the system and demonstrate that requirement are *not* met.
4. To avoid vulnerabilities in a system using the scheme, due to incorrect usage of the scheme, namely, avoid *incorrect use* of cryptographic mechanisms.

The last item is worth extra attention; incorrect usage is a common reason for cryptography-related vulnerabilities, and the exploits are often devastating. In this textbook, we will discuss several such weaknesses - in widely deployed standard and products. We believe that one of the main reasons for these failures is the fact that the system designers were not sufficiently familiar with the security properties of the cryptographic schemes they used. Our hope is that readers of this textbook will learn enough to allow them, when using cryptographic schemes, to understand their properties and how to securely use them - and to know when they need to consult with a more knowledgeable cryptographer. This will reduce the likelihood of usage errors.

The challenge of rigorously defining security requirements and proving security, also implies that protocols and mechanisms are sometimes used without a rigorous definition and/or a proof. It is sometimes necessary, for functionality, business or performance considerations, to take the risk of a subtle vulnerability, which could exist due to informal requirements or analysis. However, as explained by Koblitz and Menezes, when feasible, there are important advantages to follow the provable security approach, with definitions of requirements, models and assumptions, and proofs security. The provable security approach helps to avoid, or at least reduce, vulnerabilities and attacks. In particular, a proof of security - often, even a flawed one - rules out many possible vulnerabilities. In fact, vulnerabilities are often discovered, and circumvented, when researchers work toward a proof of security.

There are also several countermeasures which may help to address the concerns about provable security:

1. The use of *computer-aided cryptography* [18], i.e., automated tools to aid in generation of proofs and in verification of the correctness of proofs.
2. The study of *attacks* and in particular of *cryptanalysis*, i.e., attacks against cryptographic mechanisms. Researchers studying attacks on systems would often ‘think outside the box’ and exploit subtle vulnerabilities,

which may be hard to identify when we only consider the proofs of security. Attacks also provide a quantitative measure of insecurity, which may be matched against a proven quantitative measure of security, identifying possible gaps which may allow improved proofs - or improved attacks.

3. Improved *understanding* of the value, as well as the limitations, of provable security, and better *intuition*, that help practitioners make use of provable security - while avoiding subtle pitfalls. We hope that this textbook will aid in the development of such improved understanding and intuition, even to readers who will not proceed to learn the theory of cryptography.

Note that while our discussion focused on cryptographic definitions and proofs, these considerations apply to cybersecurity in general. However, rigorous definitions and proofs are much less common in other areas of cybersecurity, possibly since it is harder there to find useful simplifications. We consider this as a good motivation for studying cryptography, as a way to develop the ‘adversarial thinking’ so essential in all areas of cybersecurity.

To sum up, a proof of security is a powerful tool, but, like other power tools, *should always be used carefully and correctly*. Namely, we must fully understand the definitions, assumptions and simplifications, and never assume additional properties or applicability to scenarios where the assumptions do not hold. Therefore, cybersecurity experts must master both theory and practice. A fair amount of skepticism, paranoia and humility is also advisable.

1.1.5 Risk and costs-benefit analysis

The management of computation and communication systems involves multiple challenges, including security risks. Security managers have to consider the *costs* of deploying security mechanisms, against the *probability* of an attack and the expected *damages* from possible attacks, if the mechanisms are not deployed. *Risk analysis* is an attempt to estimate the probabilities of occurrence of different attacks, and of the attacks being successful; and cost-benefit analysis uses these values, as well as the costs of deploying different security mechanisms, to decide which security mechanisms are worth deploying.

In this textbook, we do not further discuss risk and cost-benefit analysis. The reason is that we are not aware of a sufficiently reliable and generally applicable methodology for such analysis. It seems to the author that practitioners often use crude approximations based on their experience, common sense and industry-adopted estimates. Therefore, we focus on design and analysis of secure systems, and on potential vulnerabilities and attacks; we mostly ignore the *risks* of different attacks and the *costs* of different defenses. There are a few exceptions; e.g., we focus on *computationally efficient* schemes and adversaries, where by ‘efficient’ we mean ‘whose runtime is bounded by a polynomial in the size of their inputs’; see more in Section A.1.

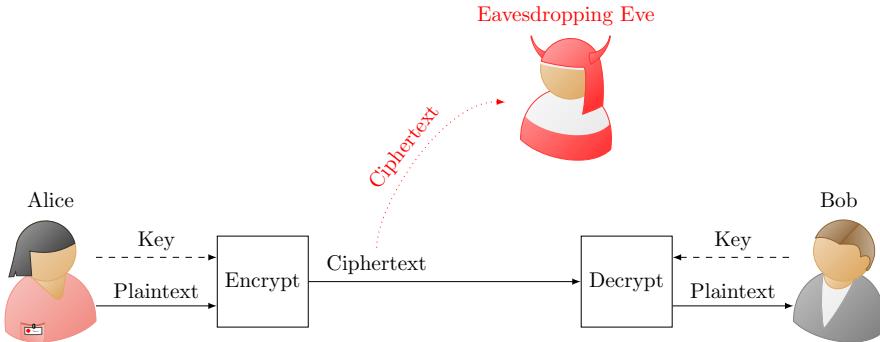


Figure 1.3: Encryption: terms and typical use. Alice needs to send sensitive information (*plaintext*) to Bob, so that the information will reach Bob - but remain confidential from the attacker, Eavesdropping Eve. To do this, Alice *encrypts* the plaintext, typically using a key; the encrypted form is called *ciphertext*. A secure encryption would prevent Eve from learning, from the ciphertext, anything about the plaintext (except how much was sent). However, by *decrypting* the ciphertext, typically using a key, Bob would recover the sent plaintext. In *symmetric cryptosystems*, encryption and decryption use the same (shared, symmetric) key, while in *asymmetric cryptosystems*, encryption uses a *public encryption key* and decryption uses a different, albeit related, *private decryption key*.

1.2 The basic mechanisms: encryption, signatures and hashing

In this section, we provide a birds-eye introduction to two basic cryptographic mechanisms: cryptosystems (symmetric and asymmetric) and signature schemes. We also introduce *Kerckhoffs' principle*, a fundamental design principle for cryptographic schemes, mostly applied also to other cybersecurity defenses.

1.2.1 Encryption: symmetric and asymmetric cryptosystems

Encryption schemes, also referred to as *cryptosystems* or *ciphers*, are the oldest and most well known cryptographic mechanism - and the main mechanism for ensuring *confidentiality*.

Encryption transforms sensitive information, referred to as *plaintext*, into a form called *ciphertext*, which allows the intended recipients to *decrypt* it back into the plaintext; the ciphertext should not expose any information to an attacker. The focus on encryption and confidentiality is evident in the term *cryptography*, i.e., ‘secret writing’, which is often used as a synonym for *cryptology*. In fact, for some reason, in the recent years, *cryptography* seems to be the more common term; so we also use it.

In all but some ancient (and completely insecure) cryptosystems, the en-

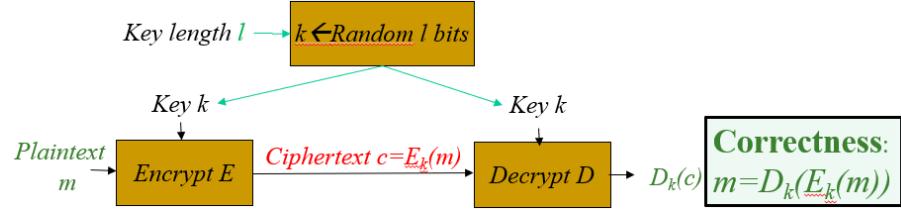


Figure 1.4: Shared key (symmetric) cryptosystem.

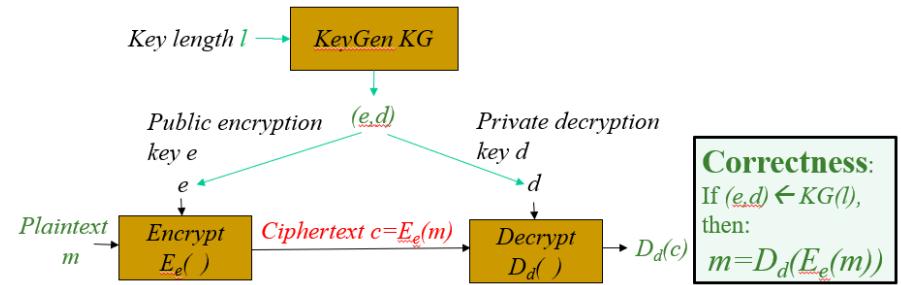


Figure 1.5: Public key (asymmetric) cryptosystem.

Encryption and decryption operations use a key. In *symmetric cryptosystems*, encryption uses the same (secret) key as used for decryption, often denoted k . In contrast, in *asymmetric cryptosystems*, only the decryption key, denoted d , must be private, and the (related by different) encryption key, denoted e , can be published, i.e., is *public*. Due to these properties, symmetric cryptosystems are also called *shared-key cryptosystems*, and asymmetric cryptosystems are also called *public key cryptosystems (PKCs)*. We study shared-key cryptosystems in Chapter 2, and public-key cryptosystems in Chapter 6.

Symmetric (shared-key) cryptosystems use the same key, e.g., k , for both encryption and decryption, as illustrated in Figure 1.4. The key k is chosen as a random bit-string. In the figure, the key length is variable, given as input (denoted l); in practice, many cryptosystems are designed for a specific key length. A symmetric cryptosystem must ensure *correctness*, i.e., $m = D_k(E_k(m))$, for every plaintext message m and every key k .

The basic security requirement from symmetric cryptosystems is to ensure *confidentiality*, i.e., an adversary should not be able to learn, from the ciphertext, any information about the plaintext. We present the definition only in subsection 2.7.2, since it involves some subtle aspects.

Asymmetric (public-key) cryptosystems are illustrated in Figure 1.5. As shown, public key cryptosystems use a *pair* of two different, but related, keys: a *public encryption key* e for encryption, and a *private decryption key* d for decryption. The *keypair* (e, d) is produced by a *key generation* algorithm, denoted KG , which is defined as part of the asymmetric cryptosystem. For public-key cryptosystems that support variable key length l , the length l is

provided as input to the KG (as illustrated). Public key cryptosystems should also satisfy *correctness*, namely: $m = D_d(E_e(m))$, for every keypair (e, d) generated by KG .

Note that a key generation algorithm is not required for shared-key cryptosystems, since the shared key k can simply be selected as a random string. However, clearly, for asymmetric cryptosystems, we cannot select randomly both the public key e and the (corresponding) private key d .

Formally, every symmetric cryptosystem can be viewed also as an asymmetric cryptosystem, simply by defining KG as random selection of l bits, which are used for both d and e . However, such construction will fail to provide the stronger security requirement from public-key cryptosystems, i.e., that an adversary should not be able to learn, from the ciphertext, any information about the plaintext, *even when given the encryption key e* .

Indeed, it is not trivial to design asymmetric encryption schemes which ensure correctness - and which are not easy to ‘break’. Even today, there is only a limited number of different designs for public key encryption schemes. In fact, while the *concept* of public-key cryptosystems was proposed in a seminal paper [101] by Whit Diffie and Martin Hellman, the a *design* was first published only later, in [276], by Rivest, Shamir and Adelman. We present two of the most well-known public key cryptosystems, *RSA* and *El-Gamal*, in Chapter 6.

Readers are encouraged to try to come up with their own design for an asymmetric cryptosystem - not necessarily a really secure one, just something which will not be trivial to break. You may find it quite a challenge - and are welcome to peer in Chapter 6 to see how it can be done.

1.2.2 Kerckhoffs’ principle

In both symmetric and asymmetric cryptosystems, the key used for decryption must be kept secret. However, what about the *algorithms* used for encryption and decryption? Knowledge of the algorithms may help an attacker, so, intuitively, it seems that the algorithms should be kept secret. Indeed, some ancient cryptosystems did not use a key at all, and relied entirely on the secrecy of the algorithms; we give few examples in Chapter 2. Even for keyed cryptosystems, it is harder to attack without knowing the design; see Exercise 2.25. This is often reflected using the expression *knowledge is power*. Therefore, traditionally, cryptosystems were kept secret, and this secrecy was considered as necessary for their security - an approach we refer to as *security by obscurity*.

However, in 1883, the Dutch cryptographer Auguste Kerckhoffs realized that ‘security by obscurity’ has serious disadvantages; in particular, once an attacker obtains a cryptosystem, the system may become completely insecure. This motivated Kerckhoffs to publish the following principle [190], which is now considered the ‘basic’ rule in applied cryptography. We extended Kerckhoffs’ principle to refer to arbitrary security mechanisms, and not just to cryptosystems or even to cryptographic systems.

Principle 2 (Kerckhoffs' principle). *When designing or evaluating the security of (cryptographic) systems, assume that the adversary knows the design, i.e., knows everything except the secret keys.*

Kerckhoffs' principle has additional advantages. One advantage is that the resulting design is likely to be more secure, as it was designed against a more powerful attacker - an attacker that knows the details of the design. An even more important advantage is the ability to *evaluate* the security of the design by experts which were not part of the design team, and challenging them to find vulnerabilities; it is often easier for an expert to find a vulnerability in a system designed by somebody else.

Note that the Kerckhoffs' principle does *not require that the design be made public*; it merely *allows* publication, since it requires that the *goal* of the designers should be for security to hold even against an attacker than knows the design. In principle, we may follow Kerckhoffs' principle, yet keep the design private, thereby making it 'even harder' to find a vulnerability. Since security does not *assume* secrecy of the design, we can continue to use the system even if the confidentiality of the design is breached.

However, there are *further advantages* in going further and relying on the security of *public, standard* designs. Published, standard designs have the obvious advantage of improving the efficiency of production and use, by allowing interoperable implementations by arbitrary vendors. A more subtle advantage of published, and esp. standard, designs, is that they facilitate evaluation and cryptanalysis by many experts, and motivate experts to *find and publish vulnerabilities*. As a results, users will be alerted to vulnerabilities earlier, reducing the risks of using a system with a vulnerability known only to the attacker. Through evaluation of security by multiple, motivated experts is the best possible guarantee for security and cryptographic designs - except, possibly, for provably-secure designs. In fact, as mentioned above (and in [196]), even 'provably-secure' designs were found to have vulnerabilities during careful review by experts, due to a mistake in the proof or due some modeling or other assumption. Therefore, Justice Brandeis' saying that 'sunlight is the best disinfectant' applies also to cryptographic and cybersecurity systems.

This is well demonstrated by two important cryptographic systems, which whose design was kept confidential, both designed in the 1990's: the GSM mobile telephony network and the *CSS* encryption for DVDs. In both cases, it did not take very long for the algorithms to be leaked, and quite soon afterwards, they were found to be insecure and vulnerable to successful, practical attacks. The GSM case is particularly interesting and important, with several glaring vulnerabilities, some of which can be considered as design errors. The GSM designers did not even plan a proper 'migration plan' for changing from the exposed ciphers and protocols to more secure alternatives, which resulted in devices remaining vulnerable years after the vulnerabilities becoming known to all experts. We discuss some of these vulnerabilities in Section 5.6.

Sometimes it is feasible to combine the security benefits of open design and of maintaining the secrecy of the design, by combining two candidate schemes.

For example, we use a cryptosystem which is a *combination* of a published, standard cryptosystems and of another cryptosystem, designed for security (following Kerckhoffs' principle) but kept confidential (to make attacks harder). We use the term *robust combiner* for such construction, which ensures security as long as *one* of the two schemes is not broken; several such robust combiners are known, for different cryptographic schemes; see [160] and subsection 2.5.9.

1.2.3 Digital Signature schemes

Cryptosystems are designed to ensure confidentiality, i.e., secrecy of information. Let us now focus on *digital signature schemes*, which are designed to ensure the *authentication* and *non-repudiation* goals (subsection 1.1.2).

Digital signature schemes, like asymmetric encryption, were proposed as a concept in [101], but their first published design was in the RSA paper [276]. Later on, we discuss two constructions of signature schemes: the RSA design in Chapter 6, and a hash-based design which is limited to signing a single message, called a *one-time signature schemes*, in Chapter 3.

Handwritten signatures: goals and reality. To provide intuition to the *digital signature*, let us first consider *handwritten signatures*. Ideally, handwritten signatures should allow everyone to verify a signed document, by comparing the signature on the document to a sample signature, known to be of that signer. The purported security of handwritten signature, is based on two (implicit) assumptions. The first assumption is that *only* the signer herself should be able to sign a document in a way which produces a signature matching her sample signature. The second assumption is that it is infeasible to *change* a signed document, without leaving marks that would invalidate the signature.

Reality is less ideal; handwritten signatures are forged, hand-signed documents are modified. Indeed, for these reasons, there are experts who are called upon to detect forged signatures and documents modified after signature - and these experts may fail to make a correct determination - indeed, different experts may disagree. A serious forger may, for example, lure the victim signer into signing a benign-looking document, such as '*I owe Mal \$1*', which may make it easier to modify into '*I owe Mal \$1,000,000*' without leaving marks. It could be quite hard to prove that the document was modified after it was signed.

Digital signatures and their security. Let us now focus on digital signatures, illustrated in Figure 1.6, and their security properties. Basically, we argue that *digital signatures provide the security that is desired, but not really provided, by handwritten signatures*.

There are some obvious differences between handwritten signatures and digital signatures. Obviously, the document and the signature are strings (files) rather than ink on paper, and the processes of signing and validating are done by applying appropriate functions (or algorithms), as illustrated in Figure 1.6.

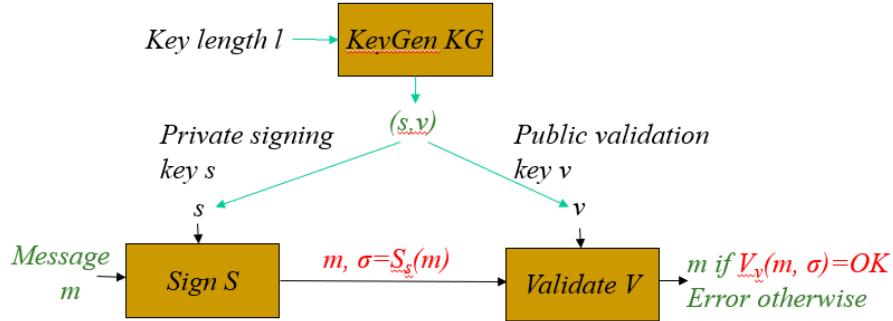


Figure 1.6: Digital Signature Scheme.

The signing and verification functions require appropriate *keys*. The *private signing key*, s , replaces the unique personal characteristics of the signer, creating the personalized signature; the *public verification key*, v , replaces the sample signature available to anybody wishing to verify signed documents. This keypair, (s, v) , is produced together, using a *key generation algorithm*, KG .

A signature scheme $\mathcal{S} = (\mathcal{KG}, \text{Sign}, \text{Verify})$, therefore, consists of these three algorithms: \mathcal{KG} , for producing the keys, Sign , for signing, and Verify , for verifying. These three algorithms are similar to these of a public key cryptosystem (Figure 1.5), and, like there, the key generation algorithm, KG , may receive as input an indication of the required key-length l .

The key generation algorithm outputs a pair of keys: a private signing key s , to be known only to the signer, and a corresponding public verification key v , to be known to anybody who wishes to verify signatures. The Sign algorithm is given the (private) signing key s and a message m ; the output of the signature algorithm, which we denote in Figure 1.6 by σ , is usually referred to as the *signature* of the message m : $\sigma = \text{Sign}_s(m)$.

Signature schemes *should* also ensure *security*. Intuitively, the security requirement is that an attacker cannot *forge* messages. Namely, given the public verification key v , an attacker cannot obtain a signature σ such that $V_v(m, \sigma) = \text{TRUE}$, unless it was also given the private signing key s or the computed value of $\sigma = S_s(m)$.

We provide a precise definition of this security requirement in subsection 1.4.1, since this definition requires some notations and background that we did not yet discuss. Note, however, that like public-key encryption, it is non-trivial to *design* a signature scheme (which is not easy to break). Therefore, we present an implementations only much later: of RSA in Chapter 6, and of one-time signatures in subsection 3.4.3.

The reader may wonder why do we define the security requirements of signature schemes already in this chapter. There are a few reasons:

- The security requirements of signature schemes are relatively simple and easy to define. By discussing the requirements and defining them, we

demonstrate the tools of probability and computational complexity, which are later used to define other cryptographic schemes.

- Signatures are one of the most important and widely used cryptographic schemes, yet - they are one of the least known and understood outside the cryptographic community.
- Signatures are the main *deterrence-based* security mechanism, at least in cryptography, and definitely in this textbook. Deterrence is not always sufficiently recognized for its potential to ensure security; we discuss this next.

Non-repudiation. Everyone can use the public validation key v to verify signatures; this key does not suffice to generate valid signatures! Therefore, a digital signature can provide not only authentication, but also *non-repudiation*: the signer cannot deny having signed the message. As we will see in Chapter 7 and Chapter 8, this property is critical to applications of public-key cryptography, including the TLS protocol, which is key to web-security and other applications.

Warning: conflicting use of the term ‘digital signature’. To finish this high-level introduction to the cryptographic mechanism of digital signatures, we need to warn the reader about a conflicting use of the same term for a very different mechanism. Specifically, the term ‘digital signature’ is often used to refer to the *visual appearance* of a ‘signature’ in a document, such as a signature included in a PDF file. This mechanism offers little or no technical defense against forgery; it is quite easy for an attacker to use the same visual ‘signature’ in a different file, i.e., forgery is often easy.

1.2.4 Applying Signatures for Evidences and for Public Key Infrastructure (PKI)

Signatures are *asymmetric*: *signing* requires the private signing key, but *validation* of a signature only requires the corresponding public verification key. This property facilitates two functions which are critical to cybersecurity: provision of *evidences* and facilitating the *Public Key Infrastructure (PKI)*. Let us briefly discuss both of these functions.

Signatures facilitate non-repudiation and evidences. The recipient of a signed message ‘knows’ that once she validated a signature, using the verification key v , she would be able to *convince other parties* that the message was, in fact, signed by the use of the private key corresponding to v . We refer to this property as *non-repudiation*, since the owner of the private key cannot claim that the ability to verify messages allowed another party to forge a signature, i.e., compute a seemingly-valid signature of a message, without access to the private signing key.

The non-repudiation property allows a digitally-signed document to provide an *evidence* for the agreement of the signer, much like the classical use of hand-written signatures. Indeed, the use of digital signatures to prove agreement, has significant advantages compared to the use of hand-written signatures:

Security. Handwritten signatures are prone to forging of the signature itself, as well as to modification of the signed document. If the signature scheme is secure (i.e., *existentially unforgeable*, see Definition 1.6), then production of a valid signature over a document m practically requires the application of the private signing key to sign *exactly* m .

Convenience. Digital signatures can be sent over a network easily, and their verification only requires running of an algorithm. Admittedly, signature verification does involve some non-negligible overhead, but this is incomparably easier than the manual process and expertise required to confirm handwritten signatures. Later on, digital signatures may be easily archived, backed-up and so on.

Non-repudiation is essential for many important applications, such as signing an agreement or a payment order, or for validation of recommendations and reviews. Non-repudiation is also applied extensively in different cryptographic systems and protocols.

Legal interpretation of signatures and digitized handwritten signature. Digital signatures are covered by legislation in some jurisdictions, however, their legal definition and implications vary significantly between jurisdictions, and often differs considerably from what you may expect based on the cryptographic definitions and properties. For example, many web services use the term ‘digital signature’ to refer to agreement by a user in a web form, sometimes accompanied by a visual representation of a handwritten signature. Other systems and organizations, consider as a ‘digital signature’ the scanned or scribbled version of a person’s signature, which may be better referred to as a *digitized handwritten signatures*. Such services may offer *convenience*, but not the *security* of digital signatures (in the sense used in this textbook and by experts). In particular, since *digitized handwritten signatures* are merely digitally-represented images, they definitely cannot prevent an attacker from modifying the ‘signed’ document in arbitrary way, or even reusing the signature to ‘sign’ a completely unrelated document. From the security point of view, these *digitized handwritten signatures* are quite insecure - not only compared to cryptographic signatures, but even compared to ‘real’ handwritten signatures, since ‘real’ handwritten signatures may be verified with some precision by careful inspection (often by experts).

Signatures facilitate public key infrastructure (PKI) and certificates. Most applied cryptographic systems involve public key cryptosystems (PKCs), e.g. RSA, and key-exchange protocols, e.g. the Diffie-Hellman (DH) protocol,

both presented in Chapter 6. In particular, PKCs and key-exchange are central to the TLS/SSL protocol (Chapter 7), which is probably the most widely-used and important cryptographic protocol, and the main cryptographic web-security mechanism. However, all of these depend on the use of *authentic public keys* for remote entities, using only public information (keys). This still leaves the question of establishing the authenticity of the public information (keys).

If the adversary is limited in its abilities to interfere with the communication between the parties, then it may be trivial to ensure the authenticity of the information received from the peer. In particular, if the adversary is passive, i.e., can only eavesdrop to messages, then it suffices to simply send the public key (or other public value).

Some designs assume that the adversary is inactive or passive during the *initial* exchange, and use this exchange information such as keys between the two parties. This is called the *trust on first use (TOFU)* adversary model.

In few scenarios, the attacker may inject fake messages, but cannot eavesdrop on messages sent between the parties; in this case, parties may easily authenticate a message from a peer, by previously sending a challenge to the peer, which the peer includes in the message. We refer to this as a *off-path* adversary. Off-path adversaries are mainly studied when focusing on non-cryptographic aspects of network security.

However, all these methods fail against the stronger Man-in-the-Middle (MitM) adversary, who can modify and inject messages as well as eavesdrop on messages. Furthermore, there are many scenarios where attackers may obtain MitM capabilities, and even when this seems harder to believe, it is always better to ensure security against such powerful attackers, following the conservative design principle (Principle 3). To ensure security against a MitM attacker, we must use strong, cryptographic authentication mechanisms.

Signature schemes provide a solution to this dilemma. Namely, a party receiving signed information from a remote peer, can validate that information, using only the public signature-validation key of the signer. Furthermore, signatures also allow the party performing the signature-validation, to first validate the public signature-validation key, even when it is delivered by an insecure channel which is subject to a MitM attack, such as email. This solution is called *public key certificates*, and we discuss it in Chapter 8.

1.2.5 Cryptographic hash functions

A cryptographic hash function h receives an input string m and outputs a short string $h(m)$. We refer to this output as the hash, fingerprint, digest or checksum of the input string m .

Several security properties are defined for cryptographic hash functions; see Chapter 3. The most well-known is *collision resistance*. Collision resistance means that given the digest $h(m)$ of some string m , it is infeasible to find a different string $m' \neq m$, which has the same digest: $h(m') = h(m)$. The collision resistance property is often used to ensure integrity, e.g., integrity of software downloads; see Lab 1.

1.3 Background and notations

Modern cryptography makes extensive use of mathematics, in particular, complexity theory, number theory, group theory, and probability. These are large, important and interesting areas, which are often part of computer science curriculum. However, we believe that it is not essential to study these areas *before* studying this textbook; the textbook only requires limited use of basic concepts and results from these areas. Instead, we provide the necessary, limited background, for a reader who did not learn these areas so far, in Appendix A. In fact, studying this textbook before studying these topics, may provide motivation and prepare the reader for in-depth study of these important and interesting areas.

In this section, we briefly introduce these topics, to allow readers to determine if they need to learn a bit from any of them, or if they know enough from prior studies. If readers find it necessary to learn a bit more about these topics, we provide the necessary background in Appendix A. Readers can read the appendixes in advance and/or ‘as needed’, i.e., when the text makes use of the relevant area.

In subsections 1.3.4 and 1.3.5, we introduce *notations* used throughout this textbook. Notations are important, therefore, we urge readers to read and refer to these sections. In subsection 1.3.4 we present *sequence diagrams*, a widely-used graphical notation for presenting interactions between entities, which we use extensively to present protocols and attacks; and the convenient *dot notation*, which identifies values associated with a particular entity. In subsection 1.3.5, we focus on additional notations, many of which may be familiar to readers; these notations are summarized in Table 1.1, for handy reference when reading the text.

1.3.1 A bit of Computational Complexity

Most cryptographic schemes assume restrictions on the computational abilities of the adversary². The focus on adversaries with restricted computational capabilities, allows us to rule out some attacks which require absurd amount of resources, such as *exhaustive search* - trying out all possible keys (see subsection 2.3.1).

How, however, can we give a well-defined meaning to a restriction on the adversary’s computational abilities? This can be quite tricky. However, the theory of *computational complexity* provides an elegant solution. In this introductory textbook, we only need to understand some very basic notions and properties from the theory of complexity, which we summarize in Section A.1. Our discussion is quite restricted to what we consider essential, and simplified, although it basically conforms with the corresponding precise definitions. Hopefully, this will suffice to allow readers who are not familiar with computational

²There are also some definitions and constructions of *unconditionally secure* cryptographic schemes. We cover two important unconditionally-secure schemes: *One Time Pad (OTP)* encryption (Section 2.4) and *Secret Sharing* (Section 10.1).

complexity to follow this textbook. It may also motivate some readers to take a course or read one of the many excellent textbooks on computational complexity, e.g., [88, 142], which will provide a more complete and precise coverage.

Some of the aspects of computational complexity which we use in the textbook, and describe in Section A.1, include:

- The big-O notation and its use for specifying and comparing the time and space complexities of algorithms.
- The definition of *probabilistic polynomial time (PPT)* algorithms, also referred to as *efficient* algorithms or *polytime* algorithms, and the corresponding notion for functions.
- The security parameter 1^l , which roughly can be viewed as the key length, encoded in unary (i.e., a string of l bits whose value is 1).
- The *non-deterministic polynomial-time (NP)* class of problems, and the $NP \stackrel{?}{=} P$ question.

1.3.2 A bit of Number Theory and Group Theory

Number theory and group theory are often used in the design and analysis of cryptographic schemes. In this textbook, we use only a tiny subset that is necessary for our study of applied cryptography.

The subset of number theory that we need is mostly focused on *modular arithmetic*, i.e., the computation of expressions involving arithmetic operations over integers, where the operations include modulo operations. Modular arithmetic shares many of the properties of regular arithmetic (over the integers and the real numbers), however, there are also important differences; a reader not familiar with these basic topics, is strongly advised to learn this topic, from Section A.2 or from one of the many good textbooks covering this topic.

One example of a difference between modular arithmetic and regular arithmetic is the subject of *multiplicative inverses*, which we cover in subsection A.2.2. The multiplicative inverse of a number x is denoted x^{-1} , and is the number satisfying $x \cdot x^{-1} = 1$, where we may use modular multiplication or regular multiplication. Over the integers, there are no multiplicative inverses (except for 1 who is its own inverse, while over the reals, every number, except zero, has an inverse). For modular arithmetic, the situation is a bit more complex: an integer a has multiplicative inverse modulo integer $m > 0$, if and only if a and m are *coprime*, namely, they do not have a common divisor (except 1).

Computing the multiplicative inverse is one of the problems which is widely believed to be *computationally hard*, i.e., it is believed that there is no efficient (PPT) algorithm to compute multiplicative inverses, but only when applied to numbers which are hard to factor, i.e., numbers chosen as a multiplication of very large random primes. Finding the factors of such numbers is referred to as the *factoring problem*, and it is considered *computationally-hard* (Section A.1); and when the factors are hard to compute, computing multiplicative inverses is

also hard. In fact, this is crucial to the security of the well-known and important RSA public key cryptosystem, which we discuss in Chapter 6. Specifically, in RSA, the private key is the multiplicative inverse of the public key, which means that if it is possible to efficiently compute multiplicative inverses, then RSA is insecure.

For details and more applications of multiplicative inverses, see subsection A.2.2.

In subsection A.2.3, we discuss two important and beautiful results of number theory, which are very important in cryptography: Fermat’s and Euler’s theorems. Among their application, is also the efficient computation of multiplicative inverses - for numbers with known, or small, prime factors. In fact, these theorems can also allow to reduce the complexity of *modular-exponentiation*. This property is key to the design of the RSA public key cryptosystem (see Chapter 6).

Finally, in subsection A.2.4, we introduce basic notions from the domain of group theory, which are used widely in applied cryptography, and a bit in this textbook. In particular, group theory is used to define the discrete logarithm problem, which is another important number-theoretic problems considered computationally hard. The discrete logarithm problem is used as the basis for several public-key schemes, including the important Diffie-Hellman key-exchange protocol (also in Chapter 6).

1.3.3 A bit of Probability

Probabilistic analysis and algorithms are very important for computer science in general and for cryptography in particular. However, luckily, only the very basics are required for our study of applied cryptography. We present this minimal background on probability in Section A.3; for more in-depth coverage, take a course and/or read one of the many excellent textbooks, e.g., [88, 149].

Probability deals with events which result in a value from some predefined set. For simplicity, we only consider a *finite* set of possible outcomes, and only *uniform distributions* and independent random variables.

We are often interested in the outcome of a *randomized (or probabilistic) algorithm* \mathcal{A} , i.e., an algorithm that can perform *bit flip* operations. We use the notation $\Pr(\pi(\mathcal{A}))$ to denote the probability that predicate π holds for the (randomized) output of \mathcal{A} , and the notation $y \leftarrow \mathcal{A}(x)$ to denote that y is assigned the random outcome of a uniformly-chosen run of \mathcal{A} with input x .

In Section A.3, we present several simple, yet useful, properties of probability, which we use in this textbook.

1.3.4 Sequence diagrams and the dot and key notations

Let us now present *sequence diagrams*, a widely-used technique for illustrating the interactions between entities over time; sequence diagrams are widely used in cybersecurity, communication protocols and other areas involving interactions between entities. We also introduce two important notations that we use

throughout this textbook: the *dot notation*, for referring to items within a tuple, and the *key notation*, for denoting the key when provided as input to a cryptographic function.

Sequence diagrams. Figure 1.7 is a *sequence diagram*, like most figures in this textbook and in most of the cybersecurity and networking literature. A sequence diagram illustrates the progression of events over time. In this figure, and in most figures in this textbook, the time proceeds from top to bottom; note that in some other schedule diagrams, time proceeds from left to right. The top of the diagram shows the different parties and processes/algorithms. For example, in Figure 1.7, we have two parties, Alice and Bob, and the three algorithms comprising the signature scheme $(\mathcal{K}\mathcal{G}, \text{Sign}, \text{Verify})$. The arrows represent communication: over the network (between parties) or within the same party (with algorithms). To show delays, the arrows may be slanted; but in this textbook, we mostly ignore the communication delays, hence the sequence diagrams mostly use horizontal arrows.

The dot notation. We use $A.s$ and $A.v$ to denote the signing and verification keys of Alice, respectively. Here, A stands for ‘Alice’, and the s and v after the dot represent specific values (keys) associated with Alice. We find dot notation a convenient way to identify *different keys* and other values associated with a particular entity. We also use dot notation, when necessary, to avoid ambiguity when referring to the *different functions* comprising a cryptographic scheme, for example, a signature scheme $\mathcal{S} = (\mathcal{K}\mathcal{G}, \text{Sign}, \text{Verify})$, e.g.: $(s, v) \xleftarrow{\$} \mathcal{S}.\mathcal{K}\mathcal{G}(1^l)$. Finally, we also use dot notation to refer to *different outputs* of a function returning multiple values. We use these conventions throughout this textbook.

The key notation. Cryptographic functions often use *keys*. While the key is, formally, an input to the function like any other input, it is often convenient, and customary, to place the key as *subscript* to the function name. For example, we use $\text{Sign}_s(m)$ to denote the signing algorithm Sign applied to the message m , using the *signing key* s . In Figure 1.7 we use this notation together with the dot notation, and write $\text{Sign}_{A.s}(m)$ to denote the result of applying signature algorithm Sign to message m , using the signing key $A.s$ of Alice.

Example: sequence diagram for signature schemes. Figure 1.7 demonstrates sequence diagrams and some of the notations we presented, by presenting a *sequence diagram* of the initialization and the typical use (signing and verifying) of a signature scheme.

1.3.5 Notations

Notations are essential for precise, efficient technical communication. However, it can be frustrating to read text which uses unfamiliar, forgotten or confusing notations. This could be a special challenge for these readers of this text

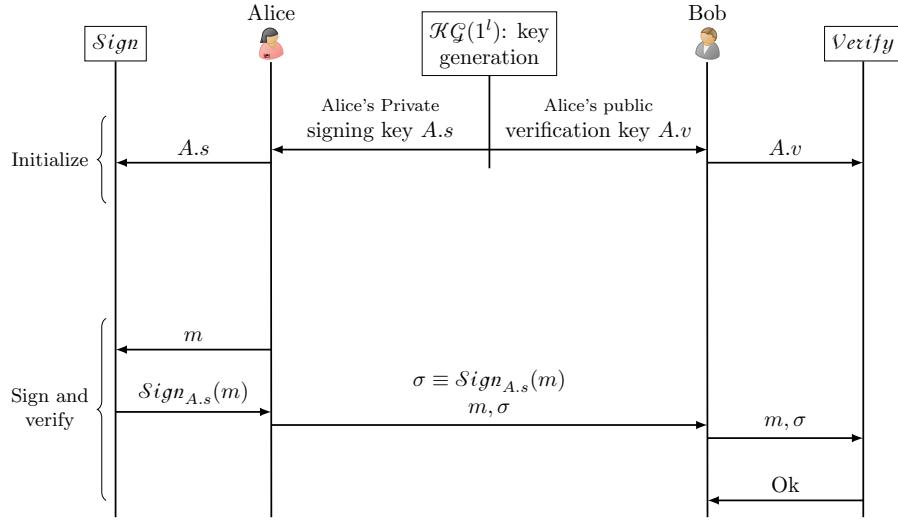


Figure 1.7: Sequence diagram for the initialization and use of a signature scheme. Alice *signs* message m with her private signing key $A.s$, resulting in $\sigma = \text{Sign}_{A.s}(m)$. Alice sends σ and m to Bob, who *verifies* the signature by computing $\text{Verify}_{A.v}(m, \sigma)$. Since in this example σ is a valid signature of m , the result is *Ok*. If σ is not a valid signature of m , the result should be *Invalid*.

who were not much exposed to notations used in mathematics and theory of computer science. Furthermore, unfortunately, there are often multiple notations for the same concept as well as multiple conflicting interpretations for the same notation.

To try to help the readers to follow the notations in this text, Table 1.1 presents notations which we use extensively. Please refer to it whenever you see some unclear notation, and let me know of any missing, incorrect or confusing notation.

We tried to choose the more widely used and least conflicting and confusing notations, but that required some difficult tradeoffs. For example, we use the symbol $\#$ to denote string concatenation, although the symbol \parallel is more commonly used to denote string concatenation in cryptographic literature. The reason for preferring $\#$ is to avoid confusing readers who are used to the use of the symbol \parallel to denote the logical-OR operator, as in several programming languages.

1.4 Provable-Security and Definitions

Ensuring security is challenging. It is tempting to identify a list of possible attacks, and evaluate security against these; but that is often misleading, resulting in vulnerability to other, unforeseen attacks. Instead, modern cryptography is mostly based on *provable security*, whose goal is to *prove* that an attacker

Table 1.1: Notations used in this manuscript.

$S = \{a, b, c\}$	A set S with three elements - a, b and c . Sets are denoted with capital letter.
$\mathbb{N}, \mathbb{Z}, \mathbb{Z}^+$	\mathbb{N} : natural numbers (integers greater than zero); \mathbb{Z} : all integers; \mathbb{Z}^+ : non-negative integers.
$\mathbb{Z}_p, \mathbb{Z}_p^*$	The sets $\{0, \dots, (p-1)\}$ and $\{1, \dots, (p-1)\}$, respectively.
$\{x \in X \mid f(x) = 0\}$	The subset of elements $x \in X$ s.t. $f(x) = 0$.
$(\forall x \in X)(f(x) > 1)$	For all elements x in the set X , holds $f(x) > 1$. Set X omitted when ‘obvious’.
$(\exists x \in X f(x) > 1)$	There is (exists) some x in X s.t. $f(x) > 1$.
$\prod_{x \in S} V_x$	Multiplication of V_x for every $x \in S$, e.g., $\prod_{x \in \{a, b, c\}} V_x = V_a \cdot V_b \cdot V_c$. Similar to use of $\Sigma_{x \in S}$ for addition.
$i!$	The factorial of i , defined as: $i! \equiv \prod_{j \in \{1, \dots, i\}} j = 1 \cdot 2 \cdot \dots \cdot i$.
$C \cup B$	Union of sets C and B .
$A \subseteq B$	Set A is a subset of set B , i.e., $a \in A \Rightarrow a \in B$.
$A \subsetneq B$	Set A is a ‘proper’ subset of set B , i.e. $A \subseteq B$ but $A \neq B$. For example, $\mathbb{N} \subsetneq \mathbb{Z}^+ \subsetneq \mathbb{Z}$.
$A \times B$	Cross-product of sets A and B , i.e., the set $\{(a, b) a \in A \text{ and } b \in B\}$.
$0x\dots$, e.g., $0xAF2$	Hexadecimal string, i.e., $0x$ is followed by a string of hexadecimal digits (from 0 to F).
$\{a, b\}^l$	The set of strings of length l over the alphabet $\{a, b\}$.
$\{a, b\}^*$	The set of strings of any length, over the alphabet $\{a, b\}$.
$\#$ (or \parallel)	Concatenation of strings; $abc \# de = abcde$.
a^n (e.g., 1^n , 0^n)	String consisting of n concatenations of a . For example, $0^4 = 0000, 1^3 = 111$, and $(01)^2 = 0101$. 1^n is also the number n in unary notation.
$ b $	The length of string b ; hence, $ a^n = n \cdot a $ and $ 0^n = n$.
$a[i]$	The i^{th} most significant character of string a . For a binary string, the i^{th} bit; for a byte-string, the i^{th} byte. E.g., if $a = 011$, then $a[1] = 0$ and $a[2] = 1$.
$a[i : j]$ or $a[i \dots j]$	Substring containing $a[i] + \dots + a[j]$.
a^R	The ‘reverse’ of string a , e.g., $abcde^R = edcba$.
$a.b$	Dot notation: element b of tuple a (subsection 1.3.4).
$x \wedge y$	bitwise logical AND; $0111 \wedge 1010 = 0010$.
$x \vee y$	bitwise logical OR; $0111 \vee 1010 = 1111$.
$x \oplus y$	Bit-wise exclusive OR (XOR); $0111 \oplus 1010 = 1101$.
\overline{x}	The bit-wise inverse of binary string or bit x .
$x \xleftarrow{\$} X$	Select element x from set X with uniform distribution.
$\Pr_{x \xleftarrow{\$} X}(F(x))$	The probability of $F(x)$ to occur, when x is selected uniformly from set X .
$\mathcal{A}^{\mathcal{B}_k(\cdot)}, \mathcal{A}^{f_k(\cdot)}$	Algorithm \mathcal{A} with <i>oracle access</i> to algorithm \mathcal{B} or to function f , with key k . Namely, \mathcal{A} can give input x and receive $\mathcal{B}_k(x)$ or $f_k(x)$. See Definition 1.3.
PPT	The set of <i>efficient (Probabilistic Polynomial Time)</i> algorithms; see Definition A.1.
$NEGL(n)$	Set of ‘negligible functions’ in input $n \in \mathbb{N}$, see Def. 1.5.
$O(f(n))$	Big-O notation, identifies the complexity of an algorithm; see Equation A.1.
$\boxtimes_k(\psi)$	String ψ protected (‘enveloped’) using key k , e.g., by the SSL/TLS record protocol.

with given *capabilities* is unable or unlikely to ‘break security’ of a given system or cryptographic scheme.

Proving security requires to clearly and precisely define the cryptographic scheme and its interactions, the attacker capabilities, the security requirements, and any additional assumptions. Only with precise definitions of the scheme, attacker capabilities, security requirements and assumptions, can we try to *prove* security. Note that there are also many opportunities for errors leading to vulnerabilities, either in the proof itself, or in the use of ‘incorrect’ definitions for attacker capabilities, security requirements or assumptions, e.g., when an attacker may have additional capabilities or when a cryptographic scheme is used incorrectly, i.e., assuming it ensures properties beyond its security requirements. Security and cryptography are rather unique in being very applied, yet requiring ‘correct’ and precise definitions and analysis.

In this section, we introduce the provable-security approach, by presenting the definition of *cryptographic* signature scheme (subsection 1.4.1), the relevant *attack models* (subsection 1.4.2), types of signature forgeries (subsection 1.4.3) and, finally, the security requirements of signature schemes (subsection 1.4.7). This subject has been introduced in a seminal paper from 1988 by Goldwasser, Micali and Rivest [145], which is highly recommended reading.

Why signatures? Our choice of illustrating provable-security on signature schemes, rather than encryption, has multiple motivations. First, cryptographic signatures are fascinating and widely-applied; in particular, they are fundamental to the TLS protocol (Chapter 7) and the public-key infrastructure (PKI, Chapter 8). Second, signatures are less known, and, furthermore, the term ‘electronic signature’ is often applied, confusingly, to mechanisms with weaker security guarantees. Third, we believe that the definition of security requirements of signature schemes is, surprisingly, simpler and more intuitive than that of encryption schemes (see subsection 2.7.2). Fourth, signatures allow us to define and compare multiple natural, widely-known security requirements. Finally, these definitions are used in Chapter 3, which covers integrity and hash functions, including the *Hash-then-Sign (HtS)* paradigm and hash-based signature schemes, as well as in Chapter 4, which covers message-authentication and in Chapter 6 which covers public key cryptography. Presenting them here allows the reader to chose the order of reading these chapters.

1.4.1 Definition of a Signature Scheme

We now present an example of the formal definition of cryptographic signature schemes and their *correctness requirements*; the same approach is used to study other cryptographic schemes, by first defining a scheme and its correctness requirements.

A (digital/cryptographic) *signature scheme* \mathcal{S} consists of three algorithms: $\mathcal{S} = (\mathcal{KG}, \text{Sign}, \text{Verify})$. The *Sign* algorithm is used to sign messages, using a secret/private key s ; the *Verify* algorithm is used to verify the purported

signature over a message, using a known, public key v ; and the *key generation* algorithm \mathcal{KG} generates the *keypair* (s, v) .

The definition, which follows, uses the concepts of efficient (PPT) algorithms and of security parameter, which we discuss in Appendix A. Intuitively, the security parameter indicates the desired tradeoff between security and performance; a longer security parameter implies more security and more overhead, e.g., longer keys. In most of the cryptographic literature, and specifically in this textbook, the *algorithms, including the adversaries, are efficient (PPT)*. Namely, the run-time of every algorithm is bounded by a polynomial in the length of its inputs, which usually includes the security parameter 1^l . In this textbook, there is only one exception: the unconditionally secure *One Time Pad* algorithm (Section 2.4).

Definition 1.1 (Signature scheme). *A signature scheme is a tuple of three efficient (PPT) algorithms, $\mathcal{S} = (\mathcal{KG}, \text{Sign}, \text{Verify})$, and a set M of messages, such that:*

\mathcal{KG} is a randomized algorithm whose input is a unary string (security parameter 1^l) and whose output is a pair of binary strings (s, v) , called the private key and the public key, respectively. To refer to only one of the two outputs of \mathcal{KG} , we use the dot notation, i.e., $s \equiv \mathcal{KG}.s(1^l)$ and $v \equiv \mathcal{KG}.v(1^l)$.

Sign is an algorithm that receives two binary strings as input, a signing key $s \in \{0, 1\}^*$ and a message $m \in M$, and outputs another binary string $\sigma \in \{0, 1\}^*$. We call σ the signature of m using signing key s .

Verify is an algorithm that receives three binary strings as input: a verification key v , a message m , and σ , a purported signature over m , and whose output is TRUE or FALSE (i.e., a predicate). Intuitively, Verify should output TRUE if and only if σ is the signature of m using s , where s is the signature key corresponding to v (generated with v).

Usually, the set of messages M is either the set of all binary strings $\{0, 1\}^*$, or the set $\{0, 1\}^n$ of binary strings of some fixed length n . When M is not explicitly mentioned, this implies the set of all binary strings, i.e., $M = \{0, 1\}^*$.

In practice, signature schemes often assume fixed input length n . To sign longer messages, the messages are *hashed* and *then* the output of the hash is *signed*. We refer to this as the *Hash-then-Sign* paradigm, and discuss it in subsection 3.2.6. For example, the DSA standard signature scheme [245] defines the application of Hash-then-Sign (using a standard cryptographic hash function, SHA). The motivation is efficiency; the fixed-length signature schemes have high overhead (e.g., see Table 6.1), which further increase, super-linearly, as a function of the message size. Hashing longer messages, and then signing them is much more efficient.

The definition allows the algorithms of the signature schemes to *randomized*. This may look unnecessary, but, in fact, some important signing algorithms are randomized, e.g., using the *PSS* encoding [37, 240].

The *correctness requirement* of a cryptographic scheme verifies that the scheme operates correctly under benign operating conditions, usually without allowing probability of error. For a signature scheme, this simply means that if the purported signature σ is indeed the output of the corresponding ‘Sign’ operation, then the verification will return TRUE, correctly indicating a valid signature. Namely, if $(s, v) \xleftarrow{\$} \mathcal{KG}(1^l)$ is a pair of signing key and corresponding validation key, then validation, using v , of a signature σ over a message m , produced by signing m using s would always return TRUE. Let us now turn this into a precise definition of the correctness requirement.

Definition 1.2 (Correctness of a signature scheme). *We say that a signature scheme $(\mathcal{KG}, \text{Sign}, \text{Verify})$, with set M of messages, is correct, if for every security parameter 1^l , every key-pair $(s, v) \xleftarrow{\$} \mathcal{KG}(1^l)$ and every message $m \in M$ holds:*

$$\text{Verify}_v(m, \text{Sign}_s(m)) = \text{TRUE} \quad (1.1)$$

Signature scheme security requirements. Intuitively, the security goal of signature schemes is *unforgeability*, i.e., to prevent an attacker from obtaining a (meaningful) forgery, where a forgery is a valid signature for a message that was not signed by the owner of the private signing key s . However, this goal is not well defined, and may be interpreted in several different ways; in particular, we can consider different *attack models* and *types of forgery*. We discuss such variants in the rest of this section, as well as notations and concepts that are relevant to provable security, in general and of signature schemes specifically, such as the *oracle notation*.

1.4.2 Signature attack models and the conservative design principle

Following the *attack model* principle (Principle 1), security should be defined and ensured with respect to *attacker capabilities*, i.e., *attack model*, rather than assuming a specific attack strategy. Let us discuss the attack models relevant to the security of signature schemes.

The *Key-Only Attack* model. The weakest attack model against a signature scheme, is the *Key-Only Attack* model, where the adversary is given (only) the public verification key v . More generally, in this attack model, the attacker is given (only) all the public keys and other public information; this attack model can be applied against any cryptographic scheme. However, the Key-Only Attack model is usually considered too weak. Specifically, for any practical applications of signatures, surely the adversary should also be able to observe at least one signed message, in addition to the public key; this motivates the use of a stronger attack models.

The *Known Message Attack (KMA)* model. In the *Known Message Attacker (KMA)* model, the attacker can receive (an arbitrary number of) pairs of a message and its signature. However, the attacker cannot control (choose) the messages. Variants of this model may require the attack to succeed for any given set of signed messages, or for signed random messages. However, this model is also usually considered too weak, since in many applications and scenarios, the attacker may be able lure the signer into signing some messages with specific format or content. For example, both parties are typically able to influence some of the text of a contract before it is signed.

The *Chosen Message Attack (CMA)* models. In this textbook, and in most of the works in modern cryptography, we adopt the stronger *Chosen Message Attack (CMA)* model, where the attacker can ask for, and receive, the signatures of arbitrary messages of its choosing. Furthermore, we use the a strong variant called *Adaptive Chosen Message Attack (Adaptive-CMA)* we allow the adversary to choose the messages *adaptively*, based on the public key and on the signatures it has previously received (but the word ‘adaptive’ is often omitted). See [145] for the weaker models: directed CMA (where attacker chooses the messages only based on the public key) and generic CMA (attacker chooses the messages without any input).

The Conservative Design Principle. One may argue that the Adaptive Chosen Message Attack model is ‘unnecessarily strong’. In many applications, the adversary’s ability to impact the contents of the signed message is very limited; and in some, the signer may phrase the contract, not allowing the adversary to have any (substantial) impact on it. It may seem that the KMA model, or the weaker-CMA models mentioned above (directed CMA or generic CMA), may suffice; requiring security against the much stronger CMA model may seem to impose unnecessary burden.

However, we want to avoid vulnerabilities in systems using cryptographic scheme, due to incorrect usage of the schemes as well as due to incorrect attack models. It is difficult to predict the actual environment in which a cryptographic scheme would be used, and a subtle difference between the real attacker capabilities and the attack model we use, may result in a vulnerability. Specifically, in subsection 3.2.6 we show how an attacker who can receive signature over a document which is mostly benign, except for a short string selected by the attacker, is able to forge a signature over a very different document. A variant of this attack was even demonstrated to circumvent the critical Web PKI mechanism (Chapter 8). This motivates a more *conservative* approach, e.g., the use of the stronger CMA model.

This is a special case of the important *conservative design* principle, which basically says that cryptographic mechanisms should be secure under minimal assumptions on the application scenarios, the underlying mechanisms and the attacker.

Principle 3 (Conservative design and usage). Cybersecurity mechanisms, and in particular cryptographic schemes, should be specified and designed with minimal assumptions, simple usage with minimal restrictions, strongest security requirements, and maximal, well-defined attack model (attacker capabilities), rather than being designed using assumptions which hold for a specific system or application. On the other hand, when using an underlying cryptographic scheme, the design should assume the minimal requirements from the scheme, and limit, as much as possible, the attacks that can be deployed against this underlying scheme.

Both parts of the conservative design principle are very important. Many systems were vulnerable due to the use of mechanisms designed with subtle assumptions or restrictions, or ensuring insufficiently strong properties, e.g., assuming limited attacker capabilities. Other systems used cryptographic mechanisms in sub-optimal way, which unnecessarily gave the attacker ability to exploit later-discovered vulnerabilities of the cryptographic mechanisms. The design of security mechanisms to minimize assumptions and restrictions on their usage, can also make their use easier.

1.4.3 Types of forgery

Which forgery is considered as a successful attack? Clearly, the forgery must consist of a message which wasn't signed by the legitimate signer (owner of the private key), and a valid signature. However, *which* message is considered a *meaningful* forgery? We consider three types of forgeries:

Existential forgery: *any* forgery is considered meaningful. Namely, the attacker is considered successful if it is able to obtain *any* pair of a message and a valid signature over it, where the message was not signed by the legitimate signer - even if the message is pure gibberish.

Selective forgery: The attacker selects some message $m \in M$ before it begins interacting with the system, and then succeeds in generating a valid signature for m (without asking the signer to sign m , of course).

Universal forgery: The attacker can produce a valid signature to *any* message m given to it.

Attackers that can perform universal forgery, can also perform selective forgery; and attackers that can perform selective forgery, can also perform existential forgery.

For some application scenarios, it may suffice to prevent universal forgery or selective forgery, for example:

- Assume there are only two (or few) pre-defined messages to be signed, e.g., an inspector signing either 'valid' or 'invalid'. In this case, security against universal forgery suffices; the attacker cannot choose the message to be forged.

- Assume the application of signing a document, e.g., a contract. The attacker may have significant flexibility in which document it forges, but the forgery must be of a legible, meaningful contract. Existential forgery, where the attacker may only forge signature over some ‘random’ gibberish document, may not be a threat.

However, following the *conservative design* principle, it is preferable to use signature schemes that prevent (even) existential forgery. Such scheme can be safely used, even in an application where the attacker may be able to exploit signatures over seemingly-meaningless or benign messages.

Our discussion of signature schemes so far has been intuitive. However, in order to *prove* security, we need precise definitions, which we present in the following subsections.

1.4.4 Game-based Security and the Oracle Notation

There are different methods of defining security requirements for cryptographic schemes; this textbook, and possibly most works on applied provable security, follows the *game-based* approach. We believe that game-based definitions are easier, more intuitive and used by more works on provable security of applied protocols, compared to other approaches such as *simulation-based* definitions.

Games. The term *game* refers to a well-defined algorithm that returns a binary outcome of one execution, where an adversary \mathcal{A} attacks the scheme: TRUE if the attack succeeded and FALSE if the attack failed. The game is often randomized; randomness may be used by the game itself, e.g., to define random challenges, by the adversary \mathcal{A} and/or by the cryptographic scheme (if it is probabilistic, i.e., uses random bit-flip operations).

Game-based security definitions define a game, often using pseudo-code, and then use the same to define the security requirements.

Oracles and the oracle notation. Many cryptographic games, allow the adversary to receive the result of specific, limited operations that use the private key - while not providing that key to the attacker. For example, a Chosen Message Attacker (CMA) can receive signatures of messages it chooses, where the signatures are computed using the private signing key, which is obviously not disclosed to the attacker.

Basically, the game provides the attacker with ‘black box’ or ‘subroutine’ access to a function, which, internally, has access to the private key. Specifically, to allow CMA, the attacker is given such access to a function computing $\text{Sign}_s(m)$, where m is a message chosen by the attacker, and s is the private signing key (not given to the attacker).

The term *oracle access* is used to refer to such ‘black box’ access, e.g., to the function computing $\text{Sign}_s(m)$ for attacker-chosen message m . Oracles are used extensively in complexity theory and in cryptography.

We use the *oracle notation* $\mathcal{A}^{\mathcal{S}.Sign_s(\cdot)}$ to denote that the adversary \mathcal{A} is given *oracle access* to $\mathcal{S}.Sign_s(\cdot)$, the signing functionality using the private signing key s , for attacker-chosen messages. This means that \mathcal{A} can provide input $x \in \{0, 1\}^*$ and receive $\mathcal{S}.Sign_s(x)$, i.e., a signature of x using the secret key s . Notice that \mathcal{A} does not receive the private signing key s , and has no access to the operation of $\mathcal{S}.Sign_s(\cdot)$.

Definition 1.3 (Oracle notation). *Let \mathcal{A} be an algorithm, f be a function, and $k \in \{0, 1\}^*$ be a string (typically, a secret such as a private key). We use the notation $\mathcal{A}^{f_k(\cdot)}$ to denote that algorithm \mathcal{A} can provide input x and receive $f_k(x)$. Similarly, we use the notation $\mathcal{A}^{\mathcal{B}_k(\cdot)}$ to denote that \mathcal{A} can provide input x and receive $\mathcal{B}_k(x)$, where \mathcal{B} is an algorithm. We refer to $f_k(\cdot)$ or $\mathcal{B}_k(\cdot)$ as an oracle.*

In the next subsection, we use the oracle notation to define the existential-unforgeability CMA game.

1.4.5 The Existential Unforgeability CMA Game

Algorithm 1 presents the pseudocode of the algorithm for the *existential unforgeability adaptive chosen-message attack (CMA)* game, $EUF_{\mathcal{A}, \mathcal{S}}^{Sign}(1^l)$. The game returns TRUE if the adversary ‘wins’, i.e., is able to output some message m and a valid signature for it σ ; otherwise, i.e., if the attack fails, then the game returns FALSE.

Algorithm 1 The existential unforgeability game $EUF_{\mathcal{A}, \mathcal{S}}^{Sign}(1^l)(1^l)$ between signature scheme $\mathcal{S} = (\mathcal{K}\mathcal{G}, \mathcal{S}.Sign, \mathcal{S}.Verify)$ and adversary \mathcal{A} .

```
( $s, v$ )  $\xleftarrow{\$}$   $\mathcal{S}.KG(1^l)$ ;
( $m, \sigma$ )  $\xleftarrow{\$}$   $\mathcal{A}^{\mathcal{S}.Sign_s(\cdot)}(v, 1^l)$ ;
return ( $\mathcal{S}.Verify_v(m, \sigma) \wedge (\mathcal{A} \text{ didn't give } m \text{ as input to } ., \mathcal{S}.Sign_s(\cdot))$ );
```

Explanation of the existential unforgeability game $EUF_{\mathcal{A}, \mathcal{S}}^{Sign}(1^l)$ (Algorithm 1). The game receives only one input, the security parameters 1^l , and has only three steps:

1. Use the key-generation algorithm of the signature scheme, to generate the signing and verification keys: $(s, v) \xleftarrow{\$} \mathcal{S}.KG(1^l)$. We use the $\xleftarrow{\$}$ symbol to emphasize that $\mathcal{S}.KG$ is a randomized algorithm, i.e., returns a random key pair.
2. Then, we let $(m, \sigma) \xleftarrow{\$} \mathcal{A}^{\mathcal{S}.Sign_s(\cdot)}(1^l)$, i.e., the adversary outputs a message m and a purported forged signature for it, σ . The adversary receives oracle access to the signing algorithm, i.e., can receive the values $\mathcal{S}.Sign_s(x)$ for any input x chosen by the adversary.

3. Finally, the game returns TRUE, i.e., the adversary ‘wins’, if σ is a valid signature on m (using the verification key v), provided that m is not one of the inputs x whose signature $\mathcal{S}.\text{Sign}_s(x)$ was received by \mathcal{A} from the oracle in the previous step.

Intuitively, an existentially-unforgeable signature scheme \mathcal{S} ensures that every efficient (PPT) adversary \mathcal{A} would ‘almost always’ lose, i.e., $\Pr(\text{EUF}_{\mathcal{A}, \mathcal{S}}^{\text{Sign}}(1^l) = \text{TRUE})$ would be tiny or negligible, provided that the security parameter 1^l is ‘sufficiently large’. We define this requirement below, in Definition 1.4 and Definition 1.6.

The following exercise shows that the adversary \mathcal{A} can always ‘win’ the $\text{EUF}_{\mathcal{A}, \mathcal{S}}^{\text{Sign}}(1^l)$ game against an arbitrary signature scheme \mathcal{S} , if either we allow \mathcal{A} to be inefficient (i.e., not in PPT), or if the keys generated by algS are of limited length. Namely, in these cases, $\Pr(\text{EUF}_{\mathcal{A}, \mathcal{S}}^{\text{Sign}}(1^l)(1^l) = \text{TRUE}) = 1$.

Exercise 1.1 (Forgery if adversary is computationally unbounded or if key length is bounded). *Let \mathcal{S} be an arbitrary efficient (PPT) signature algorithm. Present an adversary \mathcal{A} that is able to ‘win’ $\text{EUF}_{\mathcal{A}, \mathcal{S}}^{\text{Sign}}(1^l)$ every time, if we allow either of the following:*

1. \mathcal{A} does not have to be an efficient (PPT) algorithm.
2. \mathcal{S} outputs fixed, or bounded-length, keys.

Sketch of solution to first item: Since \mathcal{S} is efficient, there is some polynomial which bounds its running time. In particular, this bounds the length of the private signing key s . The adversary will try all strings s' up to that length; the adversary will apply the signature algorithm using each such potential signature key s' , and each time, verify the signature using the public key v ; eventually, the correct signing key is found.

Sketch of solution to the second item: for simplicity, assume that the private key is always of length l . Then \mathcal{A} tries to sign using any of the 2^l possible keys, verifying the signatures using the (known) public key, until it finds the correct key. Since l is fixed, the number of keys is a (large) constant rather than a function of the security parameter; hence, the adversary can test all 2^l possible keys. \square

1.4.6 The unforgeability advantage function, concrete/asymptotic security and negligible functions

The $\varepsilon_{\mathcal{S}, \mathcal{A}}^{\text{EUF-Sign}}$ advantage function. The existential unforgeability game (Algorithm 1) is a random process, which returns the outcome of a random run of the game, with the given adversary \mathcal{A} and signature scheme \mathcal{S} . The outcome is TRUE in runs where the adversary ‘wins’, i.e., outputs a forgery, and FALSE in runs where the adversary ‘loses’, i.e., does not output a forgery.

The outcome of the game may depend on the (random) keys output by the (probabilistic) \mathcal{KG} algorithm, as well as the outputs of the (randomized)

adversary \mathcal{A} . The probability that the adversary wins usually depends on the security parameter 1^l . This probability is called the *existential unforgeability advantage* of \mathcal{A} against \mathcal{S} .

Definition 1.4. *The existential unforgeability advantage function of adversary \mathcal{A} against signature scheme \mathcal{S} is defined as:*

$$\varepsilon_{\mathcal{S}, \mathcal{A}}^{EUF-Sign}(1^l) \equiv \Pr(EUF_{\mathcal{A}, \mathcal{S}}^{Sign}(1^l) = \text{TRUE}) \quad (1.2)$$

Where the probability is taken over the random coin tosses of \mathcal{A} and of \mathcal{S} during the run of $EUF_{\mathcal{A}, \mathcal{S}}^{Sign}(1^l)$ with input (security parameter) 1^l , and $EUF_{\mathcal{A}, \mathcal{S}}^{Sign}(1^l)$ is the game defined in Algorithm 1.

The advantage function gives us a measure of the security of the signature scheme; in particular, clearly, a scheme is secure only if for any efficient adversary \mathcal{A} , the advantage is small, or better yet, *negligible*³. Note, however, that for any fixed value of the security parameter 1^l , there is an adversary \mathcal{A} that *always wins* - i.e., such that $\varepsilon_{\mathcal{S}, \mathcal{A}}^{EUF-Sign}(1^l) = 1$ (Exercise 1.1). Therefore, our definition of security cannot be bounded to a specific security parameter, and must consider the advantage as a function.

Which advantage functions are sufficiently-small (or negligible)? There are two main ways in which we can deal with this question: *asymptotic security* and *concrete security*. In this textbook we will adopt the asymptotic security approach, which we explain below, since it is a bit easier to use. However, let us first briefly explain the alternative approach of concrete security, which allows more detailed analysis of security - but is a bit harder to use.

Concrete security. The concrete security approach uses the advantage function directly as the measure of security. Namely, in this approach, there is no explicit definition of a ‘secure’ scheme; each scheme is only associated with a specific advantage function. This allows the calculation of the advantage as a specific probability value - for any given, concrete values of the security parameter 1^l ; this is also the reason for the term, *concrete security*. In fact, in this approach, often the advantage function is given additional parameters. For example, the advantage function for a signature scheme may include the number of messages signed during the execution. In general, inputs to the advantage function often include the number of (different kinds of) oracle calls.

Concrete security allows precise analysis of security for specific key lengths and other parameters, and the security impacts of different cryptographic constructions. These is a significant advantage of concrete security over asymptotic security (defined next). However, we believe that concrete security may be less appropriate for this introductory textbook. Instead, we decided to adopt the simpler-to-use *asymptotic security* approach, described next, where a design is either secure or insecure - no quantitative measure.

³Unfortunately, no efficient signature scheme can ensure zero advantage; see Exercise 1.6.

Asymptotic security and negligible functions. Asymptotic security requires the advantage function, e.g., $\varepsilon_{\mathcal{S}, \mathcal{A}}^{\text{EUF-Sign}}(1^l)$, to be *negligible* in the security parameter l .

What does it mean when we say that a function $\varepsilon : \mathbb{N} \rightarrow \mathbb{R}$ is *negligible*? Clearly, we expect such a function to converge to zero for large input, i.e.: $\lim_{l \rightarrow \infty} \varepsilon(l) = 0$. Moreover, a negligible function is a function that converges to zero *faster than any (non-zero) polynomial*; the definition follows.

Definition 1.5 (Negligible function). *A function $\varepsilon : \mathbb{N} \rightarrow \mathbb{R}$ is negligible, if for every non-zero polynomial $p(l) \neq 0$ holds:*

$$\lim_{l \rightarrow \infty} \varepsilon(l) \cdot p(l) = 0 \quad (1.3)$$

We use NEGGL to denote the set of all negligible functions.

Notes:

1. An equivalent condition is to say that $\varepsilon : \mathbb{N} \rightarrow \mathbb{R}$ is *negligible* if for every c holds $\lim_{l \rightarrow \infty} \varepsilon(l) \cdot l^c = 0$.
2. Any non-zero polynomial is not negligible.
3. For any constant $x > 1$, the inverse exponential function $\varepsilon(l) = x^{-l}$, e.g., 2^{-l} , is negligible.

Here is an exercise to make sure this important concept is well understood.

Exercise 1.2. which of the following functions are negligible? Why?

- (a) $f_a(l) = 10^{-8} \cdot l^{-10}$, (b) $f_b(l) = 2^{-l/2}$, (c) $f_c(l) = \frac{1}{l!}$, (d) $f_d(l) = \frac{(-1)^l}{l}$, (e) $f_e(l) = 0.5^l$.

Working with negligible functions is a useful simplification; here is one convenient property, which shows that if an algorithm has negligible probability to ‘succeed’, then running it a polynomial number of times will not help - the probability to succeed will remain negligible.

Lemma 1.1. Consider negligible function $\epsilon : \mathbb{N} \rightarrow \mathbb{R}$, i.e., $\epsilon \in \text{NEGGL}$. Then for any polynomial $p(l)$, the function $f(l) = p(l) \cdot \epsilon(l)$ are also negligible, i.e., $f(l), g(l) \in \text{NEGGL}$.

1.4.7

Existentially-unforgeable signature scheme. Finally, let’s define an *existentially unforgeable* signature scheme.

Definition 1.6 (Existentially-unforgeable signature scheme). A signature scheme \mathcal{S} is existentially unforgeable if for all PPT algorithms \mathcal{A} , the advantage of \mathcal{A} over \mathcal{S} is negligible, i.e.: $\varepsilon_{\mathcal{S}, \mathcal{A}}^{\text{EUF-Sign}}(1^l) \in \text{NEGGL}(l)$, where $\varepsilon_{\mathcal{S}, \mathcal{A}}^{\text{EUF-Sign}}(1^l)$ is defined in Definition 1.4.

Recall that $\varepsilon_{\mathcal{S}, \mathcal{A}}^{EUF-Sign}(1^l) \equiv \Pr(EUF_{\mathcal{A}, \mathcal{S}}^{Sign}(1^l)(1^l = \text{TRUE}))$ is the probability that the adversary \mathcal{A} succeeds to forge a message, in a random run of the existential-unforgeability game, with adversary \mathcal{A} , signature scheme \mathcal{S} and security parameter 1^l .

Exercise 1.1 shows that that a signature scheme cannot be existentially-unforgeable, if the length of the keys it generates is fixed or bounded, regardless of the security parameter. In spite of this, standard signature schemes are defined for fixed key (and input, output) length.

We leave the definition of the corresponding game and notion of selective-unforgeability to the reader. Notice we do not include in this exercise universal-unforgeability, since it requires a slightly different type of definition.

- Exercise 1.3.**
1. Define the selective-unforgeability game.
 2. Define a selectively-unforgeable signature scheme.
 3. Show: if \mathcal{S} is existentially-unforgeable, then \mathcal{S} is selectively-unforgeable.

One-Time Signatures. Our definitions above focused on the ‘classical’ definitions of signature schemes and their security. However, there are many other variations considered in the cryptographic literature; let us mention one of these variants, *One-Time Signature* schemes. These are signature schemes which allow only a single of signature operation. A similar variant allows a limited number of signature operations (*Limited-use signatures*).

- Exercise 1.4.**
1. Define the existential-unforgeability game for a one-time signature (or: limited-use signature).
 2. Define a one-time existentially-unforgeable signature scheme.

One-time (and limited-use) signatures can be more computationally efficient than the ‘classical’, unlimited-use signatures; see subsection 3.4.3. They may also be convenient way to ensure security against attackers with quantum-computing capabilities, see Section 10.4. Adapting the definitions we presented above to support such variants is not difficult; see Exercise 1.7. And there are many applications, where we can use one-time signatures.

So why do we define and normally use ‘classical’, unlimited-use signatures? One reason is the conservative-design principle above. It is also simpler to use (and define) unlimited-use signatures, and it is more convenient that we can use them for many applications. Convenience, reuse and simplicity are all very important properties.

1.5 A Brief History of Cryptography and Cybersecurity

Cryptography is a surprisingly ancient art, indeed, one of the earliest sciences; and it also played a surprisingly large role in the development of computing.

And the history of computing is obviously closely linked to the history of cyberspace and cybersecurity - where cryptography plays a major role.

Therefore, we conclude this chapter with a brief review of the history of cryptology (subsection 1.5.1), and of the history of computing and cybersecurity (subsection 1.5.2).

1.5.1 A brief history of cryptography

We now present a brief history of cryptography, which we divide to three subjects: from ancient cryptography to Kerckhoffs' book ([190], 1883), cryptography in the WWII and the birth of computing, and modern cryptography. We are only able to give few important highlights from the fascinating history of cryptology, and interested readers should consult some of the excellent manuscripts such as [184, 297].

From ancient cryptography to Kerckhoffs. Cryptology, which literally means the ‘science of secrets’, has been applied to protect sensitive communication since ancient times, and is therefore much more ancient than computing devices. Originally, cryptology focused on protecting secrecy, i.e., on *confidentiality*, which is mainly provided by *encryption* schemes, also called *ciphers* and *cryptosystems*; see Figure 1.3.

One of the early evidences of encryption is from about 1500BC; it is an encryption of a formula for pottery glaze, which presumably was commercially valuable. We present few other ancient ciphers in Section 2.1, but are not able to properly cover this topic here; if interested, read some of the excellent books such as [184, 217, 297].

Kerckhoffs's publication [191], in 1883, could be viewed as the beginning of a new era in cryptology. Kerckhoffs's work may have been the first major published work in cryptography; until it, most works in cryptography, and designs of cryptosystems, were kept secret, following the *security by obscurity* approach. Kerckhoffs realized that it is better to assume that the attacker may be able to capture encryption devices and reverse-engineer them to learn the algorithm, as we paraphrase in the Kerckhoffs principle (Principle 2), see subsection 1.2.2.

Kerckhoffs's principle remains a basic principle of cryptography, and is gradually being adopted into other areas of cybersecurity. With the increased use of cryptography, adoption of standards, software implementations and advanced reverse-engineering tools, Kerckhoffs principle only became even more important.

Note that Kerckhoffs did not argue that cryptogrpthic designs *should* be published. And, indeed, for many years after Kerckhoffs book was published, research and development in cryptography remained an area mostly dealt with by intelligence and defense organizations, and mostly in secret. This was definitely true until, and during, World War II.

Cryptography in WWII and the birth of computing. The most important advances in cryptography after Kerckhoffs's publication were made as part of the efforts of the second world war (WWII), and in the period leading to it. Cryptography, and in particular cryptanalysis, i.e., 'breaking' the security of cryptographic schemes, played a key role during the second world war; an important by-product was the development of the first computer.

During the war, both sides used multiple types of encryption devices. The most well known is the *Enigma* and encryption device, used by the Germans. The early versions of the Enigma were in use already from 1924, and it was modified and improved over the years.

Details of the design of the Enigma were kept secret, however, the designers clearly tried to follow Kerckhoffs' principle, and ensure security even if the design would be known to the cryptanalysts. This fact, together with continuous improvements to Enigma, made the Germans believe that Enigma will not be broken.

Indeed, the Allies were very concerned about their lack of ability to decipher Enigma traffic; and the cryptanalysis of Enigma was a major undertaking and a huge achievement. However, there may have been also a drawback to the German cryptographic effort: they were over-confident in the security of Enigma, and continued using it long after it was broken. Possibly due to this overconfidence, they also used Enigma in ways which made it easier to break the cipher, in violation of the *conservative design and usage* principle (Principle 3); See Section 2.2.

When the Enigma was eventually broken, the allies took extraordinary measure to prevent the Germans from realizing this, so that the Germans will not change keys, change Enigma or take other precautions. Usually, this meant creating alternate explanations to the Allies response to the information in the plaintext; but sometimes , the difficult decision was made to avoid reaction to the information, since the resulting risks to people and/or property was deemed less than the risks if the Germans realize that Enigma was broken. It has been estimated that the successful cryptanalysis of Enigma, shortened the war by about two years, and saved millions of lives.

While Enigma was designed following Kerckhoffs principle, i.e., to ensure security even if its design is known, the German also made extensive efforts to maintain the secrecy of the design. Indeed, the successful attacks on Enigma were based on leakage of some information about it. In 1932, the French intelligence were able to obtain the secret manual and the settings for the Enigma from a German officer, and shared this information with the British intelligence and the Polish cryptanalysis unit, called the Cypher Bureau.

In the Polish Cypher Bureau, Captain Maksymilian Cięński led team that used mathematics for cryptanalysis, consisting of three of his cryptography students: Rejewski, Zygalski and Różycski. Using Engima's manual and setting, the team reverse-engineered the Enigma, and built Enigma replicates. Furthermore, they developed special-purpose electromechanical devices called *Bombe*, that allowed efficient testing of different Enigma keys against limited exposed information, such as known pairs of ciphertext and (likely) plaintext.

In 1939, Poland shared their Bombe devices and their cryptanalysis results with the British. Using plaintext/ciphertext pairs and the Bombe machines, the cryptanalysis center in Bletchley Park, led by Alan Turing, was able to decipher much of the intercepted Enigma traffic.

However, the Germans periodically improved the Enigma. Every change in the Enigma required extensive effort to design and create modified Bombe devices, and a period when traffic could not be deciphered.

Furthermore, the somewhat less known *Lorenz encryption devices* were introduced by the Germans later in the war, and no complete device was captured and available to cryptanalysts.

The challenges of adapting Bombe devices to changes in Enigma, and of breaking the Lorenz devices, motivated the construction of the *first electronic computer*, called *Colossus*. For more on the history of computing and cybersecurity, see subsection 1.5.2. Since Colossus was programmable, it was possible to test many possible attacks and to successfully cryptanalyze (different versions of) Lorenz, Enigma and other cryptosystems.

Modern cryptography. Until the 1970s, cryptography remained mainly a topic for intelligence and research organizations. In the 1970s, this changed, quite dramatically, with the beginning of what we now call *modern cryptology*, which involves extensive academic research, publication, products and standards, and has many important commercial applications.

Two important landmarks mark the beginning of modern cryptology. The first landmark is the development and publication of the *Data Encryption Standard (DES)* [244]. The publication of DES as an open standard, marks that cryptography began to be widely deployed in commercial products; in particular, DES was key to the security of the emerging bank networks, and to the security mechanisms of the emerging computer communication networks, which was dominated, for many years, by IBM's SNA.

The second landmark is the introduction of radical, innovative concept of *Public Key Cryptology (PKC)*, where the key to encrypt messages may be *public*, allowing easy distribution of encryption keys. The first publication was of the seminal paper *New directions in cryptography* [101], by Diffie and Hellman. In [101], Diffie and Hellman introduced the concepts of public-key cryptography, public-key encryption and digital signatures. They also presented the important *Diffie-Hellman Key Exchange* protocol; we discuss public key cryptography and the Diffie-Hellman protocol in Chapter 6.

It is notable in [101], Diffie and Hellman did not yet present a design of a public key cryptosystem. The first published public-key cryptosystem was RSA by Rivest, Shamir and Adelman in [276]. RSA and the Diffie-Hellman protocol remain widely-used public-key cryptographic mechanisms; we discuss them in Chapter 6.

In fact, the same design as RSA was already discovered a few years earlier, by the GCHQ British intelligence organization. However, the GCHQ kept this achievement secret until 1997, long after the same design was re-discovered

independently and published in [276]. See [217] for more details about the fascinating history of the discovery - and re-discovery - of public key cryptology.

These two discoveries of public-key cryptology, with very different impacts on society, illustrate the dramatic change between ‘classical’ cryptology, studied only in secrecy and with impact mostly limited to the intelligence and defense areas, and modern cryptology, with extensive published research and extensive impact on society and economy.

1.5.2 The History of Computing and Cybersecurity

Early computing: from Babbage to Colossus. The first known idea of computing was proposed by Charles Babbage in 1822. Babbage’s created two designs of mechanical computing devices: the *difference engine* and the *analytical engine*. The difference engine was a special-purpose machine, designed to tabulate logarithms and trigonometric functions; however, the analytical engine was a general purpose computer, which was designed to process input consisting of data and a program, both provided using punched cards.

Babbage was also interested in cryptography, and designed the first known practical attack against the *Vigenère cipher* (Section 2.1). The attack was based on usage of letter frequencies, and one of the applications Babbage designed for the analytical engine was to compute letter frequencies.

Babbage but never completed implementing either engine, but the designs, however, were correct; they were implemented and tested for historical purposes, from 1989 to 1991, i.e., more than a century after he died. However, Babbage did build some modules of the engines, and has demonstrated and described their designs to peers.

The demonstration and description excited Ada Lovelace, one of the few female mathematicians at the time. Ada wrote a description of the sequence of operations for solving certain mathematical problems by the analytical engine, which is considered the first computer program; Ada was also the first to suggest that computers may be used to manipulate non-numeric data, such as text or music.

It took about a century from the initial design of the engines, and until the first *working* computing device was implemented. This was a mechanical device designed and implemented in 1938, by the German inventor Konrad Zuse; Zuse named it the Z1.

The Z1 was unreliable and slow. As a result, it did not have any useful applications or use, except as proof of feasibility. In particular, special-purpose calculating devices were way better in performing applied calculations. In particular, this held for electromechanical designs including the Enigma cipher used for encryption, and the Bombe machines used in Bletchley park to break the Enigma; both were much more efficient and reliable than the Z1.

However, as we discussed above, there was a repeated struggle to adapt the Bombe devices to changes in Enigma; and the Bombe failed to break the newer Lorenz cryptosystem. This motivated the construction of the *first practical computer*, called *Colossus*.

Colossus was designed by Tommy Flowers as part of the Bletchley park WWII cryptanalysis effort. Colossus was the first fully-electric computing device, i.e., did not involve mechanical components. The Colossus was also the first *computer*, i.e., the first computing device that could be *programmed* for arbitrary tasks, rather than only perform a predefined set of tasks or computations. This was in contrast to previous devices, including the Enigma and the Lorenz cryptosystems, which were electro-mechanical and also were limited to a predefined computation.

From Colossus to Modern Computers One critical difference between the Colossus and more modern computers, as well as from Babbage's design of the analytical engine, is that *the Colossus did not read a program from storage*. Instead, setting up the program for the Colossus involved manual setting of switches and jack-panel connections.

This method of 'programming' the Colossus wasn't very convenient, but it was acceptable for the Colossus, since there were only a few such machines and only a few, simple programs, and the simplicity of design and manufacture was more important than making it easier to change programs. Even this crude form of 'programming' was incomparably easier than changing the basic functionality of the machine, as required in special-purpose devices - including the Enigma devices and the Bombe devices used for cryptanalysis of the Enigma traffic.

Designs of an electromechanical computer which supports a stored program, were proposed already in 1936/1937 - by two independent efforts. The first was by Konrad Zuse, who mentioned such design in a patent on floating-point calculations published in 1936 [329]; the second was Alan Turing, who defined and studied a formal model for stored-program computers. This *Turing machine model*, introduced in Turing's seminar paper 'On Computable Numbers' [305], is still fundamental to the theory of computing.

However, practical implementations of stored-program computers appeared only *after* WWII. Stored-program computers were much easier to use, and allowed larger and more sophisticated programs as well as the use of the same hardware for multiple purposes (and programs). Hence, stored-program computers quickly became the norm - to the extent some people argue that earlier devices were not 'real computers'.

Stored-program computers also created a vast market for programs. It now became feasible for programs to be created in one location, and then shipped to and installed in a remote computer. For many years, this was done by physically shipping the programs, stored in media such as tapes, discs and others. Now that computer networks are widely available, program distribution is often, in fact usually, done by sending the program over the network.

Easier distribution of software meant also that the same program could be used by many computers; indeed, today we have programs that run on billions of computing devices. The ability of a program to run on many computers created an incentive to develop more programs; and the availability of a growing number of programs increased the demand for computers and their impact.

Computer networks and cyberspace. The ability to develop a program and have it applied in multiple computers caused the economic ‘network effect’ that made computers and programming much more useful. This effect dramatically increased when *computer networks* began to facilitate inter-computer communication.

The introduction of personal computers (1977-1982), and the subsequent introduction of the Internet, the web and of the smartphone, caused, each, a further dramatic increases in the use and impact of computing and of computer networking.

There was also a growing interest in the potential social implications of computers and networks, and a growing number of science-fiction works focused on these aspects. One of these was the novel ‘Burning Chrome’, published by William Gibson in 1982. Actually, it seems that it was this novel that introduced the term *cyberspace*, to refer to this interconnected environment connecting networks, computers, devices and humans.

The *cyber* part of the term *cyberspace* is taken from the term *cybernetics*, introduced in [319] to describe the study of communication and control systems in machines, humans and animals.

By now, computing is used not only in ‘traditional computers’, but as a critical component of many other devices, from tiny sensors to vehicles - cyber-physical systems and IoT (Internet of Things) devices. The term *cyberspace* is now mostly used for the ubiquitous use of devices with different computing capabilities, communicating via networks and interacting with humans.

Cybersecurity and Hacking. With great power, comes great responsibility; and the increased importance of the Internet and cyberspace also increased the risks of abuse and cyber-attacks. The awareness of these risks significantly increased as attacks on computer systems and networks became widespread, especially attacks exploiting software vulnerabilities, and/or involving malicious software, i.e., *malware*. This awareness resulted in the study, research and development of threats and corresponding security mechanisms, including *computer security*, *software security*, *network security* and *data/information security*.

The awareness of security risks also resulted in important works of fiction. One of these was the 1983 novel *cyberpunk*, by Bruce Bethke. Bethke coined this term for individuals which are socially-inept yet technologically-savvy. Originally a derogatory term, cyberpunk was later adopted, with a positive interpretation, as a name of a movement with several manifestos, e.g., [193].

The reverse process happened with regards to the term *hacker*, which was originally used, already from the 1960s, to describe proficient programmers, thinking ‘out of the box’ and finding creative solutions and shortcuts (‘hacks’), and more recently, to experts of computer security. However, from the 1980s, the term hacker is often applied with a negative connotation, to a person who tries to *break* into computer systems and to circumvent defenses. The terms *black-hat hacker* (or *cracker*) and *white-hat hacker* (or just *hacker*) are often used to distinguish between the ‘attacking hacker’ and the ‘defending hacker’,

or between hackers operating illegally and hackers following legal, and hopefully also ethical, principles.

In works of fiction, cyberpunks and hackers are often presented as socially-inept yet technology-savvy, with incredible abilities to penetrate systems. These abilities are mostly presented in positive, even glamorous ways, e.g., as saving human society from oppressive, corrupt regimes, governments, agencies and rogue Artificial Intelligence systems. The focus on decentralization and personal freedom is definitely a main part of the cyberpunk manifestos [193].

Indeed, much of the success of the Internet is due to its decentralized nature, and to the use of cryptography to provide security to financial transactions and some level of privacy. Important privacy tools, such as the *Tor* anonymous communication system [102], are based on cryptography and inherently decentralized, which may be hoped to defend against potentially malicious governments. Furthermore, some of the cryptography and privacy mechanisms, such as the *PGP* encryption suite [134], were developed in spite of significant resistance by governments. Cryptography is also at the core of the extensive efforts to develop *blockchains* (see Section 3.10) and other decentralized financial tools and currencies, such as the Bitcoin cryptocurrency.

1.6 Lab and Additional Exercises

Lab 1 (Using cryptography to validate downloads). *Malware is among the most common, well-known and harmful cybersecurity threats. In this lab, we explore the use of basic cryptographic mechanisms, specifically, cryptographic hashing and signatures, to validate downloads and thereby avoid installing and using downloaded malware that is a fake version of the software that the user wanted to download.*

*As for the other labs in this textbook, we will provide Python scripts for generating and grading this lab (*LabGen.py* and *LabGrade.py*). If not yet posted online, professors may contact the author to receive the scripts. The lab-generation script generates random challenges for each student (or team), as well as solutions which will be used by the grading script. We recommend to make the scripts available to the students, as example of how to use the cryptographic functions. It is easy and permitted to modify these scripts to use other languages/libraries or to modify and customize them as desired.*

1. *Using hash for download integrity.* In this question we use a cryptographic, collision-resistant hash function (see subsection 1.2.5 and Section 3.2) to ensure the integrity of software downloads, i.e., to ensure download is of the intended, authentic software, and not of a malware impersonated as the desired software. Software is often made available via repositories, which may not be fully secure; to ensure the integrity, the publishers often provide the hash of the software. Namely, to protect the integrity of some software download, say encoded as a string m , the publisher provides in some secure channel the value of the hash $H_m \equiv h(m)$. The user then downloads the software from the (insecure) repository, obtaining

the downloaded string m' . To confirm its integrity, i.e., confirm that $m' = m$, the user uses m' if $h(m') = H_m$, i.e., $h(m') = h(m)$. Based on the collision-resistance property of h , the fact that $h(m') = h(m)$ is believed to imply that $m = m'$. Note that other applications of hash functions may rely on other properties, for example, on the one-way property (Section 3.4).

Input: a folder $Q1files$ containing several files, and a file $Q1.hash$, containing the SHA-256 hash applied to one of the files. Note that the file contains the hash in binary bytes, not encoded as text.

Goal: identify the file in $Q1files$ whose hash is given in $Q1.hash$.

Submission: the name of the matching file and a program, $A1.py$ that, given a hash file $Q1.hash$ and a folder $Q1files$, outputs the name of the matching file (or a message if there is no matching file).

2. *Using signatures to authenticate downloads.* The hash mechanism has two disadvantages. First, an attacker which controls the value of the Digest can set it to be the digest of a malware; second, the digest must be updated for any software update. In this question, we show an alternative: *authenticating* the software using *digital signatures* (subsection 1.2.3).

Practical cryptographic libraries such as PyCryptodome use the *Hash-then-Sign* paradigm (subsection 3.2.6), i.e., they apply the signing function to the *hash* of the information to be signed. Hence, you will need to specify both a signature scheme (e.g., use RSA) and a hash function (e.g., use again the SHA-256 hash function). The reason for using the Hash-then-Sign paradigm, is that it is absurdly inefficient to apply the public-key signature algorithm directly over a (usually) long message, rather than over the (short) hash.

Input: The file $Q2pk.pem$, which contains the public validation key, which can be used to validate files purportedly signed by the legitimate software, and the directory $Q2files$ which contains several files and the corresponding purported signatures. Note: the signature was created by the *LabGen.py* script, using RSA signature (with PKCS#1 v1.5 padding) and using the SHA-256 hash function.

Goal: identify the file(s) in $Q2files$ which are properly signed, as validated using the public key given in $Q2pk.pem$.

Submission: the name(s) of the properly signed file(s), and a program, $A2.py$ that, given the public validation key file $Q2pk.pem$ and a folder $Q2files$, outputs the name(s) of the matching file(s) (or a message if there is no matching file).

3. To get a feeling for the performance of public key signatures and of cryptographic hash functions, perform experiments to and presents graphs showing:

- a) The time required for hashing as a function of the input length, from 1000 bits to one million bytes.
- b) The time required for generation of signing and validation public key pairs, from keys of lengths from 1000 bits to the maximal length you find feasible (say, up to five minutes).
- c) The time required to sign inputs for inputs of length from 1000 bits to one million bits, using each of the private signing keys you generated in the previous item.
- d) The time required to validate signatures for inputs of length from 1000 bits to one million bits, using each of the private signing keys you generated in the previous item.
- e) Explain how the results received in the previous items make sense, and the implications for the time required for hashing, signing, and validating, and the relations to the *Hash-then-Sign* paradigm (subsection 3.2.6).

Note: you may need to repeat some operations many times to be able to measure the times with reasonable precision.

Exercise 1.5. Let \mathcal{S} be a signature scheme, and let \mathcal{A}^{Key} and \mathcal{A}^σ be the following two simple adversaries:

$\mathcal{A}^{Key}(v)$ randomly guesses the signing key s . After guessing s , the adversary signs the desired message using s , and ‘wins’ if the signature validates correctly using verification key v .

$\mathcal{A}^\sigma(v)$ randomly guesses the signature σ for a message m chosen by the adversary, and ‘wins’ if σ validates correctly using verification key v .

Consider a single guess by both adversaries. Let l_s denote the length of the signing key s and l_σ denote the length of the signatures produced by the signing algorithm (assume all signatures has same length l_σ).

1. Compute the exact probability that each adversary wins, after a single guess.
2. What are the relationships between the probabilities computed in the previous item, and the probabilities of winning, after a single guess, each of the three notions of unforgeability introduced in this chapter? Explain.
3. Compute the exact probability that each adversary wins, after two guesses.
4. Consider adversaries \mathcal{A}_E^{Key} and \mathcal{A}_E^σ which operate similarly to \mathcal{A}^{Key} and \mathcal{A}^σ , except that instead of guessing only one or two times, they test every possible value (of s or of σ) until winning. What is the maximal and average number of guesses required by each of \mathcal{A}_E^{Key} and \mathcal{A}_E^σ ?

5. What is the advantage of algorithms \mathcal{A}_E^{Key} and \mathcal{A}_E^σ over \mathcal{S} ? (Definition 1.4)
6. Do the replies to the previous item imply that \mathcal{S} is not existentially unforgeable?

Exercise 1.6 (There is always some probability of forgery.). Show that there is no signature scheme \mathcal{S} such that every efficient adversary \mathcal{A} has zero advantage against \mathcal{S} , i.e., such that: $\varepsilon_{\mathcal{S}, \mathcal{A}}^{\text{EUF-Sign}}(1^l) = 0$. Preferably, show this result holds for any value of 1^l .

Exercise 1.7. Based on the definitions in this chapter, define One-time signature schemes.

Exercise 1.8. Prove that if m is co-prime with n , then for every integer $l > 0$ holds:

$$m^l \mod n = (m \mod n)^l \mod \phi(n) \mod n$$

Chapter 2

Confidentiality: Encryption Schemes and Pseudo-Randomness

Encryption deals with protecting the confidentiality of sensitive information, which we refer to as *plaintext message* m , by encoding (encrypting) it into *ciphertext* c , as illustrated in Figure 1.3. The ciphertext c should hide the contents of m from the adversary, yet allow recovery of the original information by legitimate parties, using a decoding process called *decryption*. Encryption is one of the oldest applied sciences; some basic encryption techniques were already used thousands of years ago.

The most important categorization of encryption schemes is between *shared key cryptosystems*, also called *symmetric cryptosystems* (Figure 1.4), and *public key cryptosystems*, also called *asymmetric cryptosystems* (Figure 1.5). In both cases, we use the terms ‘encryption scheme’ and ‘cryptosystem’ interchangeably.

In this chapter, we mostly focus on shared-key cryptosystems; we discuss public-key cryptography in Chapter 6. Let us begin by defining (stateless) shared-key cryptosystems and their *correctness* requirement.

Definition 2.1 (Stateless shared-key cryptosystem and their correctness). *A shared-key cryptosystem is a pair of keyed algorithms, $\langle E, D \rangle$, and sets K , M and C , called the key space, plaintext space and ciphertext space, respectively.*

A shared-key cryptosystem is correct if for every input key $k \in K$ and plaintext $m \in M$, the encryption of m using k returns ciphertext $c \in C$ that decrypts, using key k , back to m . Namely,

$$(\forall k \in K, m \in M) D_k(E_k(m)) = m \quad (2.1)$$

Definition 2.1 does not allow the encryption and decryption algorithms to maintain state, i.e., are for a stateless shared-key cryptosystem. However, many practical cryptosystems use state, as illustrated in Figure 2.1; for example, the state may be used as a counter. Let us, therefore, extend Definition 2.1 to define a *stateful shared-key cryptosystem* and its correctness.

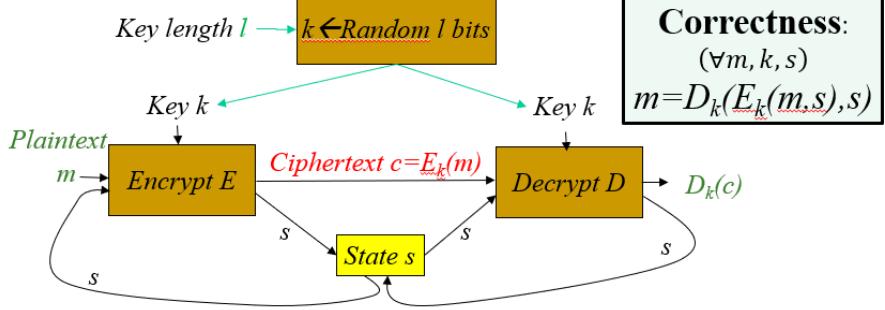


Figure 2.1: Stateful shared key (symmetric) cryptosystem.

Definition 2.2 (Stateful shared-key cryptosystem and their correctness). A stateful shared-key cryptosystem is a pair of keyed algorithms, $\langle E, D \rangle$, and sets K , M , C and S , called the key space, plaintext space, ciphertext space and state space, respectively.

A stateful shared-key cryptosystem is correct if for every input key $k \in K$, plaintext $m \in M$ and state $s \in S$, the encryption of m using k with state s returns ciphertext $c \in C$, that decrypts, using key k and state s , back to m . Namely,

$$(\forall k \in K, m \in M, s \in S) D_k(E_k(m, s), s) = m \quad (2.2)$$

Shared-key cryptosystems are also sometimes referred to as *ciphers*, but we use this term for two specific types of shared-key cryptosystems: *block ciphers*, which encrypt and decrypt *fixed-length blocks* of bits, and *stream ciphers*, which are stateful cryptosystems, which, typically, encrypt and decrypt *bit by bit*, i.e., $M = C = \{0, 1\}$.

In this chapter we will see a variety of shared-key cryptosystems. Some of these are deterministic, some randomized; some stateless, some stateful; and with different plaintext and key spaces.

Note that definitions 2.1 and 2.2 define only the *correctness* requirements; we did not yet define security requirements. We will define security later in this chapter, but it will be more complex than one may initially expect. Intuitively, the goal is clear: confidentiality, in a strong sense, against powerful adversaries. However, there are subtle issues, as well as multiple variants which differ in their exact requirements and assumptions about the adversary capabilities.

2.1 Historical Ciphers

Cryptology is one of the most historical sciences. To ‘warm up’ for our discussion of encryption schemes, let us first discuss a few historical ciphers, which were in use from ancient times till the nineteenth century. These simple, historical ciphers help us introduce some of the basic ideas and challenges of cryptography

and cryptanalysis, and provide some historical perspective, beyond the little we presented in Section 1.5. For more information, see [184, 297].

One has to keep in mind is that the *design* of these ciphers was mostly kept secret, believing that an attacker cannot break the cipher if it does not know its design; a (usually false) belief we refer to as *security by obscurity*. In fact, some of the ancient ciphers relied *only* on the secrecy of their design, and did not even use a secret key; we discuss such *keyless ciphers* in subsection 2.1.1.

Even when using a published design, users typically kept their choice secret, and often did minor changes. Indeed, it *is* harder to cryptanalyze a scheme which is not even known. Still, it is ill-advised to rely on ‘security by obscurity’. We explain this in subsection 1.2.2, where we present the *Kerckhoffs’s principle*, which essentially says that security of a cipher should not depend on the secrecy of its design.

Many historical ciphers, and in particular most ancient ciphers, were *monoalphabetic substitution ciphers*. Monoalphabetic substitution ciphers use a fixed mapping from each plaintext character to a corresponding ciphertext character (or some other symbol). Namely, these ciphers are stateless and deterministic, and defined by a permutation from the plaintext alphabet to a set of ciphertext characters or symbols. We further discuss general monoalphabetic substitution ciphers in subsection 2.1.3. We also discuss, in subsection 2.1.5, three variants of the Vigenére cipher, which is *Poly-alphabetic* substitution ciphers.

2.1.1 Ancient Keyless Ciphers

In this section, we discuss few ancient ciphers. These ciphers are all simple, *keyless* (no secret key) and *monoalphabetic*. A cipher is monoalphabetic if it is defined by a single, fixed mapping from each plaintext letter to ciphertext letter or symbol.

The At-Bash cipher The At-Bash cipher may be the earliest cipher whose use is documented; specifically, it is believed to be used, three times, in the Old Testament book of Jeremiah. The cipher maps each of the letters in the Hebrew alphabet to a different letter. Specifically, the letters are mapped in ‘reverse order’: first letter to the last letter, second letter to the second-to-last letter, and so on; this mapping is reflected in the name ‘At-Bash’¹.

The At-Bash cipher is illustrated in Fig. 2.2. Even if you are not familiar with the letters of the Hebrew alphabet, the mapping may still be identified by the visual appearance. If you still find it hard to match, that’s Ok; we next describe an adaptation of the At-Bash cipher to the Latin alphabet.

To properly define ciphers as well as more complex cryptographic scheme, we use pseudocode or a formula. For monoalphabetic ciphers, a formula usually suffices. We define the cipher as a function of the input letter, where each letter is represented by its distance from the beginning of the alphabet.

¹The name ‘At-Bash’ reflects the ‘reverse mapping’ of the Hebrew alphabet. The ‘At’ refers to the mapping of the first letter (‘Aleph’, א) to the last letter (‘Taf’, ט), and of the second letter (‘Beth’, ב) to the second-to-last letter (‘Shin’, ש).

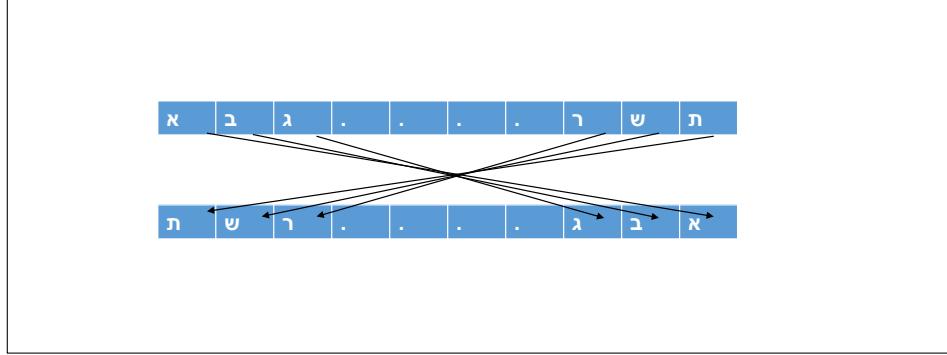


Figure 2.2: The At-Bash Cipher.

In Hebrew, there are 22 letters, so we encode them by the numbers from 0, representing the Hebrew letter Alef (\aleph), to 21, representing the Hebrew letter Taf (τ).

Let p be a *plaintext* message consisting of l Hebrew letters, where $p[i]$, for $i = 0, \dots, (l-1)^2$, is the encoding of the corresponding letter ($p[i] \in \{0, 1, \dots, 21\}$). We use c to denote the corresponding l -letters *ciphertext*, i.e., the encryption of p using the At-Bash cipher: $c = E_{At-Bash}(p)$. We compute the c using the following formula, for $i = 0, \dots, (l-1)$:

$$(\forall i = 0, \dots, (l-1)) \quad c[i] = 21 - p[i] \quad (2.3)$$

It is convenient to denote the alphabet size by n , i.e., in Hebrew, $n = 22$. With this convention we can rewrite the formula in Equation 2.3 as $c[i] = (n-1) - p[i]$.

The Az-By cipher The Az-By cipher is the same as the At-Bash cipher, except using the Latin alphabet, which has $n = 26$ letters (from A to Z).

We illustrate the Az-By cipher in the top part of Fig. 2.3; below it, we present two other keyless ancient ciphers, which we next discuss - the Caesar and the ROT13 ciphers.

Let p be a *plaintext* message consisting of the encoding of l Latin letters, where $p[i] \in \{0, 1, \dots, 25\}$, i.e., $p[i] \in \{0, 1, \dots, (n-1)\}$. The corresponding Az-By ciphertext, $c = E_{Az-By}(p)$, is given by:

$$(\forall i = 0, \dots, (l-1)) \quad c[i] = 25 - p[i] = (n-1) - p[i] \quad (2.4)$$

Obviously, this formula is the same as of the At-Bash cipher (Equation 2.3), except for adjusting for the fact that the Latin alpha has $n = 26$ letters, while the Hebrew alphabet has only 22.

²For technical reasons, it is a bit more convenient to use 0, rather than 1, as the index of the first plaintext letter ($p[0]$).

AzBy														
A	B	C	D	E	F	G	H	I	J	K	L	M		
Z	Y	X	W	V	U	T	S	R	Q	P	O	N		
N	O	P	Q	R	S	T	U	V	W	X	Y	Z		
M	L	K	J	I	H	G	F	E	D	C	B	A		

Caesar														
A	B	C	D	E	F	G	H	I	J	K	L	M		
D	E	F	G	H	I	J	K	L	M	N	O	P		
N	O	P	Q	R	S	T	U	V	W	X	Y	Z		
Q	R	S	T	U	V	W	X	Y	Z	A	B	C		

ROT13														
A	B	C	D	E	F	G	H	I	J	K	L	M		
N	O	P	Q	R	S	T	U	V	W	X	Y	Z		
N	O	P	Q	R	S	T	U	V	W	X	Y	Z		
A	B	C	D	E	F	G	H	I	J	K	L	M		

Figure 2.3: The AzBy, Caesar and ROT13 Ciphers.

The Caesar cipher. We next present the well-known Caesar cipher. The Caesar cipher has been used, as the name implies, by Julius Caesar. It is also a monoalphabetic cipher, operating on the set of the $n = 26$ Latin letters, from A to Z.

In the Caesar cipher, each plaintext letter is replaced by the letter appearing in the alphabet three places after the plaintext letter. To map the last three letters in the alphabet (X, Y and Z), we repeat the three first letters (A, B and C), i.e., X is mapped to A, Y to B and Z mapped to C. See the middle row of Figure 2.3.

As with the Az-By cipher, we represent each letter by its distance from the beginning of the Latin alphabet; i.e., we represent the letter ‘A’ by the number 0, and so on; ‘Z’ is represented by 25.

Let p be a *plaintext* message consisting of the encoding of l Latin letters, where $p[i] \in \{0, 1, \dots, 25\}$, and let c denote the l -lettered corresponding Caesar ciphertext $c = E_{\text{Caesar}}(p)$. Then c is given by:

$$(\forall i = 0, \dots, (l - 1)) \ c[i] = p[i] + 3 \pmod{26} \quad (2.5)$$

For example, consider encryption of plaintext word ‘axe’, whose encoding is $p[0] = 0$, $p[1] = 23$ and $p[2] = 4$. This gives $c[0] = 3$, $c[1] = 0$, $c[2] = 7$, i.e., the ciphertext string ‘dah’. Simple!

Exercise 2.1. Let p and c be a plaintext string and the corresponding Caesar ciphertext, as above. Show the process for decrypting a given ciphertext c , i.e., computing the corresponding plaintext p . Use an equation similar to Equation 2.5.

The ROT13 cipher ROT13 is a popular variant of the Caesar cipher, with the minor difference that ROT13 ‘rotates’ the letters by 13 positions, while Caesar rotates by 3 positions.

Let p be a *plaintext* message consisting of the encoding of l Latin letters, where $p[i] \in \{0, 1, \dots, 25\}$, and let c denote the l -lettered corresponding ROT13 ciphertext $c = E_{\text{ROT13}}(p)$. Then c is given by:

$$(\forall i = 0, \dots, l - 1) \quad c[i] = p[i] + 13 \mod 26 \quad (2.6)$$

The ROT13 cipher is illustrated by the bottom row in Figure 2.3.

We are not aware of usage of ROT13 to obtain secrecy; it is normally used only to prevent inadvertent exposure to the plaintext, such as to hide potentially offensive jokes or to obscure an answer to a puzzle or other spoiler. Because of its utter unsuitability for real secrecy, ROT13 is used to refer to weak encryption schemes (e.g., ‘about as secure as ROT13’).

A convenient feature of ROT13 is that it is a *self-inverse function*, i.e., decryption is exactly the same process as encryption.

Exercise 2.2. Show that ROT13 is a self-inverse function. Namely, show that for every plaintext message p holds: $p = E_{\text{ROT13}}(E_{\text{ROT13}}(p))$. Can you identify additional ancient ciphers which are self-inverse functions?

The Masonic cipher. A final example of a historic, keyless, monoalphabetic cipher is the Masonic cipher. The Masonic cipher is from the 18th century and is illustrated in Fig. 2.4. This cipher uses a ‘key’ to map from plaintext to ciphertext and back, but the key is only meant to assist in the mapping, since it has a regular structure and is considered part of the cipher.

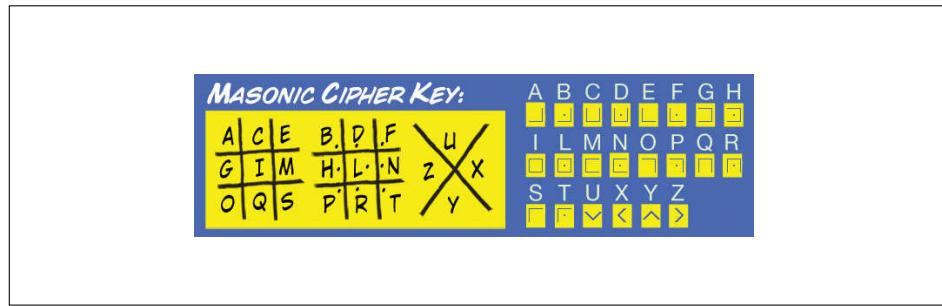


Figure 2.4: The Masonic Cipher, written graphically and as a mapping from the Latin alphabet to graphic shapes.

2.1.2 Generalized-Caesar cipher: a Keyed Variant of the Caesar Cipher

Keyless ciphers have limited utility; in particular, the design of the cipher becomes a critical secret, whose exposure completely breaks security. Therefore, every modern cipher, and even most historical ciphers, use secret keys.

Readers who are interested in these (keyed) historical ciphers should consult manuscripts on the history of cryptology, e.g. [184, 297]. We only discuss, briefly, few simple and well known keyed historical ciphers in subsection 2.1.5.

In this subsection, we present a very simple, and trivially vulnerable, keyed cipher: the *Generalized-Caesar cipher*, also referred to as the *shift cipher*. The Generalized-Caesar cipher is a simple keyed variant of the Caesar cipher. In fact, some people do not even distinguish between the Generalized-Caesar cipher and the Caesar cipher. The Generalized-Caesar cipher helps us explain the Vigenère cipher (in subsection 2.1.5), and illustrates the fact that using a key - even a long key - is not sufficient for security.

Recall that the Caesar cipher is defined by $c[i] = p[i] + 3 \pmod{n}$, where $p[i] \in \{0, 1, \dots, n-1\}$ is a single letter (with $n = 26$ for Latin). The Generalized-Caesar cipher is defined with an additional parameter: a key $k \in \{0, 1, \dots, n-1\}$.

Given an input plaintext string p consisting of l ‘letters’, $p[0], \dots, p[l-1]$, the ciphertext string $c = E_k^{\text{GC}-n}(p)$ consists of the l ‘letters’ $c[0], \dots, c[l-1]$ computed as:

$$(\forall i = 0, \dots, l-1) \quad c[i] = p[i] + k \pmod{n} \quad (2.7)$$

When using $n = 26$, the Generalized-Caesar cipher encrypts a single Latin character at a time, just like the Caesar cipher; it simply uses an arbitrary rotation k , rather than the fixed rotation as done in the original Caesar ($k = 3$) and ROT13 ($k = 13$) ciphers.

Obviously, with only $n = 26$ keys, the Generalized-Caesar cipher is also insecure; the attacker only needs to try the 26 possible key values. This attack, where the attacker tries all possible keys, is called *exhaustive search*, and can be used against any cipher.

To foil exhaustive search, we can use longer keys and blocks of plaintext. For example, if we extend our character set to include the 16-bit UCS-2 character set, then $n = 2^{16}$, making exhaustive search much harder. By using even longer keys and blocks, say, five UCS-2 characters (80 bits), we have $n = 2^{80}$ keys, making exhaustive search impractical.

However, even with very large n , the Generalized-Caesar cipher is still insecure. In particular, as the following exercise shows, even with huge n , Generalized-Caesar cipher can still be easily broken by an attacker who has access to a single pair of plaintext p and the corresponding ciphertext $c = E_k^{\text{GC}-n}(p)$. Furthermore, *any* message would do - even a single-letter message ($l = 1$). This is a very weak form of a *Known plaintext attack (KPA)*; we discuss KPA and other attack models for cryptosystems in Section 2.2.

Exercise 2.3 (Known plaintext attack (KPA) on Generalized-Caesar cipher). Let p be an arbitrary l -lettered plaintext for the Generalized-Caesar cipher

with any alphabet size n , i.e., $\forall i = 1, \dots, l : p[i] \in \{0, \dots, n - 1\}$, and let $c = E_k^{GC-n}(p)$ be the corresponding ciphertext using key $k \in \{0, \dots, n - 1\}$. Given a pair of one letter from p , say $p[1]$, and the corresponding letter from c , i.e., $c[1]$, show how the attacker can find the key k , allowing decryption of any ciphertext.

Solution: From Equation 2.7, $k = c[i] - p[i] \pmod{n}$ for any i , e.g., for $i = 0$ (the first plaintext letter and the corresponding first ciphertext letter). \square

2.1.3 The General Monoalphabetic Substitution (GMS) Cipher

(GM cipher)

Monoalphabetic substitution ciphers are deterministic, stateless mappings from plaintext characters to ciphertext characters or symbols. The use of any other set of symbols instead of letters does not substantially change in the security of such ciphers, hence we focus on permutations on a fixed alphabet.

The Mason, At-Bash, Caesar and Generalized-Caesar ciphers are all *monoalphabetic substitution ciphers*. The Mason, At-Bash and Caesar ciphers are keyless, i.e., are defined by a specific permutation. For example, the Caesar's cipher is the rotate-by-three permutation. The Generalized-Caesar cipher uses the rotate-by- k permutation, where k is the key, which is a letter in the alphabet.

The General Monoalphabetic Substitution (GMS) cipher is the simple keyed cipher obtained by applying an arbitrary permutation (mapping) from the plaintext characters to the ciphertext characters. *The key is the permutation.*

Namely, given an plaintext $p = p[0], \dots, p[l - 1]$, the ciphertext $c = E_k^{GMS}(p)$ consists of the l ‘letters’ $c[0], \dots, c[l - 1]$ computed as:

$$(\forall i = 0, \dots, l - 1) \quad c[i] = k(p[i]) \tag{2.8}$$

The key, which is a permutation over the alphabet, is often written as a table with two rows: the first containing the plaintext letters and the other containing the corresponding ciphertext letters. Some readers may recall having used sometime, maybe long ago, such ‘key tables’ to create a simple monoalphabetic cipher; many kids do.

In the Latin alphabet, there are $n = 26$ letters; each could be chosen for A , then any of the remaining 25 could be used for B , and so on. Namely, the total number of permutations (i.e., keys) is $26!$, the factorial³ of 26. The factorial grows very fast as a function of n ; for example, $26! > 2^{88}$, i.e., there are over 2^{88} permutations (keys) for 26-letters General Monoalphabetic Substitution (GMS) cipher.

Namely, the Latin alphabet, with $n = 26$ characters, suffices to make exhaustive search infeasible for General Monoalphabetic Substitution (GMS) cipher. This improves significantly compare to the Generalized-Caesar cipher,

³ $26! \equiv 26 \cdot 25 \cdots 2 \cdot 1$.

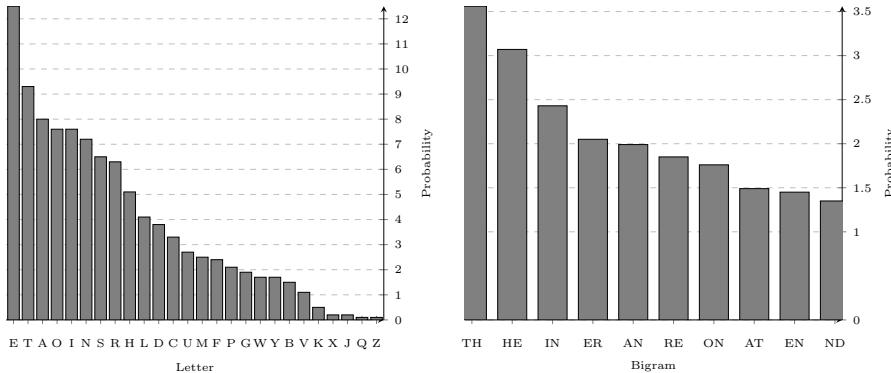


Figure 2.5: Frequencies of letters and (most common) bigrams in English, based on [250]

where to obtain the same number of keys, we need a huge alphabet of $n = 26! > 2^{88}$ letters.

However, when using a small alphabet such as of Latin, General Monoalphabetic Substitution (GMS) cipher is vulnerable to the *frequency analysis* attack, a simple attack that we describe next.

2.1.4 Frequency analysis attacks on monoalphabetic ciphers

Plaintext messages are rarely completely random strings; some messages, or part of messages, are more common than others. A *frequency analysis* attack exploits knowledge about the *plaintext distribution*, to facilitate cryptanalysis. Frequency analysis attacks often succeeded against historical ciphers, using *only* this knowledge about the plaintext distribution, and a collection of ciphertext messages; we refer to such attacks as *ciphertext only (CTO)* attacks. The frequency analysis attack is effective against any monoalphabetic cipher, including General Monoalphabetic Substitution (GMS) cipher. The only exceptions, where frequency-analysis may fail, are when using extremely large alphabets, and/or when the plaintext has uniform ('random') distribution.

Classical monoalphabetic ciphers map each letter in the alphabet of a specific natural (human) language, e.g., the Latin alphabet, to a fixed letter (or symbol). Namely, the alphabet is not very large. Furthermore, in the typical case where the plaintext is a natural language message, e.g., in English. Therefore, the plaintext is not uniformly distributed. In fact, some letters are significantly more common than others; similarly, some pairs of letters (*bigrams*) and some strings of several letters (*ngrams*), are significantly more common than others. See Figure 2.5.

Knowledge about the language, and the distributions of letters, bigrams and ngrams, is usually available to the attacker; this makes frequency analysis, and other ciphertext only (CTO) attacks, easier to launch than Known Plaintext Attacks (KPA).

Furthermore, frequency attacks work exceedingly well against classical monoalphabetic ciphers, given encryption of text in a known language. Indeed, some deductions, and often complete decryption, may be done manually; for example, in texts in English, the most common letter is almost always E (12.49%). Identification of T and H is also quite easy; T is the second-most-common (9.28%), and TH and HE are the most common bigrams (3.56% and 3.07%, respectively). Similarly, since we identified E, it is now easy to identify R too, since ER and RE are of the most common bigrams (2.02% and 1.85%). These easy deductions suffice to decrypt about a third of the letters in the ciphertext; and the reader can surely find few other easy deductions. Of course, no reason to work manually; it is easy to write a program, to efficiently cryptanalyze any monoalphabetic cipher, including General Monoalphabetic Substitution (GMS) cipher.

Exercise 2.4. *Write two programs: one that implements General Monoalphabetic Substitution (GMS) cipher, and another that cryptanalyzes the resulting ciphertexts (without being given the key). Your cryptanalysis program can assume that the encrypted plaintext is typical text in English.*

By experimenting with the cryptanalysis program of Exercise 2.4, you will find that it may fail if given short ciphertext - but is very reliable, given sufficiently long ciphertext. This phenomenon exists for other attacks too; cryptanalysis often requires a significant amount of ciphertext encrypted using the same encryption key. This motivates *refreshing (changing) the key*, thereby limiting the use of each key to a limited amount of plaintext (and ciphertext). Frequent key refresh make cryptanalysis harder or, ideally, infeasible.

Principle 4 (Limit usage of each key). *Systems deploying ciphers/cryptosystems should limit the amount of usage of each key, changing keys as necessary, to foil cryptanalysis attacks.*

An extreme example of this is the one time pad (OTP) cipher, which we discuss later (Section 2.4). The one-time pad is essentially a one-bit substitution cipher - but with a different random mapping for each bit. This turns the insecure substitution cipher into a *provably secure* cipher!

Another way to defend against letter frequency attacks, is to use a much larger alphabet, or to map sequences of letters rather than individual letters. For example, by simply mapping pairs of plaintext letters rather than single letters, we basically prevent the use of the letter-frequency table; of course, the attacker can still take advantage of the bigram distribution. However, this requires the use of an accordingly-larger table to map between plaintext and ciphertext. Such larger tables are difficult to store and share, and result in high overhead.

In Section 2.6 we present *block ciphers*, which are basically, efficient monoalphabetic substitution ciphers which use a large ‘alphabet’, e.g., 64 bits for *DES* or 128 bits for *AES*. Block ciphers use much shorter keys compared to the General Monoalphabetic Substitution (GMS) cipher. Block ciphers use

sufficiently-long keys to foil exhaustive search, typically from 56 bits (DES) to 256 bits (AES). Unlike the Generalized-Caesar cipher, however, block ciphers are designed to ensure security against frequency analysis as well as KPA and other cryptanalysis attacks.

2.1.5 The Polyalphabetic Vigenére ciphers

We conclude our discussion of historical ciphers, by discussing *polyalphabetic ciphers*, and in particular, two variants of the *Vigenére cipher*. We first describe the simpler (and weaker) (*repeating-key*) *Vigenére cipher* published, in 1553, by Giovan Battista Bellaso⁴. We then describe the stronger *Autokey*, published in 1586 by Blaise de Vigenére. We refer to these two ciphers as the Vigenére ciphers.

Both Vigenére ciphers are *polyalphabetic ciphers*, namely, they use multiple mappings from plaintext characters to ciphertext characters. The motivation to use multiple mappings is to defeat *frequency analysis* attacks, and different variants of the Vigenére cipher were used until the twenty century; in fact, even the Enigma (Section 1.5) is a polyalphabetic cipher and essentially similar to application of a cascade of few Vigenére ciphers.

The (repeating-key) Vigenére cipher. The (repeating-key) Vigenére cipher extends the Generalized-Caesar cipher, by using a string of few characters as the key, rather than a single character as in the Generalized-Caesar cipher. The (repeating-key) Vigenére cipher simply applies the letters of the key, one by one, repeating the key after using its last character.

Namely, given an input plaintext string p consisting of l characters, $p[0], \dots, p[l-1]$, and a key k consisting of λ characters, $k[0], \dots, k[\lambda]$, the ciphertext string $c = E_k^{\text{Vigenére cipher}}(p)$ consists of the l characters $c[0], \dots, c[l-1]$ computed as:

$$(\forall i = 1, \dots, l) \quad c[i] = p[i] + k[i \mod \lambda] \mod n \quad (2.9)$$

Example 2.1. Consider encryption of the string $p = 'ABCDEFG'$, whose encoding is $(i = 0, \dots, 6)p[i] = i$, using the key $k = 'BED'$, whose encoding is $k[0] = 1, k[1] = 4$ and $k[2] = 3$. The resulting ciphertext c is the string $c = 'BFFEIIH'$.

Attacking the (repeating-key) Vigenére cipher. Since polyalphabetic ciphers such as the Vigenére cipher use different keys (offsets) for different characters, direct application of frequency analysis would fail. The first attack against the (repeating-key) Vigenére cipher was published by Kasiski in 1863⁵.

⁴The (repeating-key) Vigenére cipher is usually referred to simply as the Vigenére cipher. We add the qualifier (*repeating key*) to avoid confusion with the Autokey, which is the stronger cipher published, later, by Vigenére. The Vigenére cipher was published first by Bellaso, so its name is anyway misleading.

⁵This attack was known earlier but not published; in particular, it was found in personal notes written by Babbage in 1846.

This attack proceeds in two steps: finding the length λ of the key k , and then finding the key itself $(k[0], \dots, k[\lambda - 1])$.

Once the key length λ is found, we can apply frequency analysis separately to each of the λ sequences s_i ($i = 0, \dots, \lambda - 1$) of ciphertext characters generated by the different key letters:

$$\begin{aligned} s_0 &\equiv (c[0], c[\lambda], \dots) \\ s_1 &\equiv (c[1], c[\lambda + 1], \dots) \\ &\dots \\ s_{\lambda-1} &\equiv (c[\lambda - 1], c[2\lambda - 1], \dots) \end{aligned}$$

To find λ , Kasiski looked for repeating sequences of characters (ngrams) in the ciphertext. For example, assume that for some m, m' hold $c[m : m + 3] = c[m' : m' + 3]$. By substituting $c[m : m + 3]$ and $c[m' : m' + 3]$ from Equation 2.9, we have:

$$(\forall i = 0, \dots, 3) p[m+i] + k[m+i \mod \lambda] \mod n = p[m'+i] + k[m'+i \mod \lambda] \mod n \quad (2.10)$$

Very often, the plaintext contains some repeating words or strings, ranging from relatively-common words or ngrams, to terms which are specific to that plaintext. This would be the most common reason for repeating strings in the ciphertext. Namely, when we find such m, m' , it is likely that $p[m : m + 3] = p[m' : m' + 3]$. Hence, from Equation 2.10, holds:

$$(\forall i = 0, \dots, 3) k[m+i \mod \lambda] \mod n = k[m'+i \mod \lambda] \mod n \quad (2.11)$$

Usually, when Equation 2.11 holds, then $m = m' \mod \lambda$, i.e., $m - m'$ is either λ or a multiple of λ . The attack usually proceed by finding few such repeating sequences, e.g., $m_1 = m'_1 \mod \lambda$ and $m_2 = m'_2 \mod \lambda$, and λ is the common divisor of $m_1 - m'_1$ and of $m_2 - m'_2$, or one of the (few) common divisors. We then apply frequency analysis to find each of the λ characters of the key, $k[0], \dots, k[\lambda - 1]$.

The one-time-pad. An important scenario is when we use the Vigenére cipher with a key which is (at least) as long as the plaintext. Namely, each character of the key is used to hide only one character of the plaintext. In this case, Equation 2.9 simplifies to $c[i] = p[i] + k[i] \mod n$; and in the special case of $n = 2$, we have $c[i] = p[i] \oplus k[i]$.

When the key k is chosen randomly, and especially when $n = 2$, this cipher is referred to as the *one-time pad*; we discuss it in Section 2.4. The one-time pad is *unconditionally secure*, i.e., the attacker cannot learn from the ciphertext anything about the plaintext, regardless of the attacker's computational capabilities (for any $n > 1$).

The Autokey. We now describe *Autokey*, which is the cipher that Vigenère actually designed; it is an enhancement or variant of the ‘repeating key Vigenère cipher’. Both Vigenère ciphers operate in the same way, until exhausting the characters of the key; but then they differ. Recall that the repeating-key Vigenère cipher reuses the same key string. Instead, the Autokey uses the *plaintext*.

Namely, given an input plaintext string p consisting of l characters, $p[0], \dots, p[l-1]$, and a key k consisting of λ characters, $k[0], \dots, k[\lambda]$, we compute the ciphertext string $c = E_k^{Autokey}(p)$ as follows. We first define the *autokey* k' , as $k' \equiv k||p$, i.e., the concatenation of the plaintext to the key. Next, we compute the ciphertext c , which consists of the l characters $c[0], \dots, c[l-1]$ computed as:

$$(\forall i = 1, \dots, l) \quad c[i] = p[i] + k'[i] \mod n \quad (2.12)$$

Example 2.2. Consider encryption of the string $p = 'ABCDEFG'$, whose encoding is ($i = 0, \dots, 6$) $p[i] = i$, using the key $k = 'BED'$, whose encoding is $k[0] = 1$, $k[1] = 4$ and $k[2] = 3$. The autokey would be $k' = 'BEDABCDEF'$, and the resulting ciphertext c is the string $c = 'BFFDFHK'$.

Attacking the Autokey. Let us present instructive attacks on the Autokey.

Example 2.3 (A Known plaintext attack against Autokey). Assume that the attacker captures ciphertext c , and knows that the plaintext is of the form $p = p_{Known} \# p_{Secret}$; for example, $p_{Known} = 'The password is:.'$

The attacker can find the key k and the plaintext p_{Secret} as follows, assuming that the key k is not longer than p_{Known} . Let λ denote the number of characters in the key k , i.e., $k = k[0 : \lambda]$. Hence, for $i = 0, \dots, \lambda$ holds: (1) $k'[i] = k[i]$ (by definition of k'), (2) $p[i] = p_{Known}[i]$ and (3) $c[i] = p[i] + k'[i]$ (Equation 2.12). The attacker can, therefore, find the key k , since for $i = 0, \dots, \lambda$ holds: $k[i] = k'[i] = c[i] - p_{Known}[i]$.

In the (less likely) case that k is longer than p_{Known} , this attack will expose the first $|p_{Known}|$ characters of k . This will allow exposure of the first $|p_{Known}|$ characters of the plaintext of other messages encrypted using k . To expose additional characters, we will need to use a different attack, such as the CTO attack which we describe next.

Example 2.4 (A Ciphertext Only Attack (CTO) against Autokey). CTO attacks require some knowledge about the plaintext; let us assume that the plaintext is known to be in English. Let us assume also that the attacker knows the number of characters in the key, λ .

Assume that the attacker knows some plaintext $p[i]$. Then $c[i + \lambda] = p[i + \lambda] + p[i] \mod n$; hence the attacker can find $p[i + \lambda]$. Now that $p[i + \lambda]$ is known, the attacker can find $p[i + 2\lambda]$, and so on. Similarly, the attacker can find $p[i - \lambda]$, provided, of course, that $i - \lambda \geq 0$. Namely, given a guess for $p[i]$, the attacker can find $p[j]$ for every $j = i \mod \lambda$.

This allows the attacker to decrypt the ciphertext, by using frequency analysis; let us sketch how. Since the letter E is very common (more than $\frac{1}{8}$ of the letters!),

Attack model	Cryptanalyst knowledge, capabilities	Section	Fig.
Ciphertext Only (CTO)	Plaintext distribution (possibly noisy/partial)	2.2	2.6
Known Plaintext Attack (KPA)	Set of (ciphertext, plaintext) pairs	2.2	2.7
Chosen Plaintext Attack (CPA)	Ciphertext for arbitrary plaintext chosen by attacker	2.2	2.8
Chosen Ciphertext Attack (CCA)	Plaintext for arbitrary ciphertext chosen by attacker	2.2	2.9
chosen-ciphertext side-channel attack (CCSCA)	‘Side-channel feedback’ from processing adversary-selected ciphertexts	6.5.7	6.20
CPA-Oracle Attack	Attacker receives encryptions of pre#x#post, for challenge x and chosen (pre, post), and feedback from processing selected ciphertexts	7.2.3	2.36

Table 2.1: Cryptanalysis Attack Models. In *all* attack types, the cryptanalyst knows the cipher design and a body of ciphertext.

the attacker tries different indexes i , testing if $p[i] = 4$ (indicating the letter E). To test if this is the case, the attacker uses the above process to find what would be the letters $p[j]$ for every $j = i \bmod \lambda$, and checks the distribution of the resulting sequence of letters. We conclude that the guess was correct ($p[i] = 4$, i.e., the i^{th} letter is indeed E), if and only if, the distribution is close to the letter frequency in English, in particular, if roughly $\frac{1}{8}$ of the letters in this sequence are E.

After the attacker finds, in this way, the sequence of plaintext letter $p[j]$ for every $j = i \bmod \lambda$, it can continue to decrypt additional ciphertext letters by guessing additional letters. In particular, the attacker can now use the bigram distribution, to guess letters adjacent to already-exposed letters.

2.2 Cryptanalysis Attack Models: CTO, KPA, CPA and CCA

As discussed in Section 1.1 and in particular in Principle 1, security should be defined and analyzed with respect to a clear, well defined model of the attacker capabilities, which we refer to as the *attack model*. In particular, cryptanalysis attack models define the capabilities of attackers trying to ‘break’ an encryption scheme.

In this section, we introduce the four basic *cryptanalysis attack models*: *CTO*, *KPA*, *CPA* and *CCA*; we already gave few examples of CTO and KPA attacks. Table 2.1 summarizes these four basic cryptanalysis attack models, as well as two additional cryptanalysis attack models: the chosen-ciphertext side-channel attack (CCSCA), against public-key cryptosystems, presented in Chapter 6, and the CPA-Oracle Attack, against shared-key cryptosystems, presented in Chapter 7.

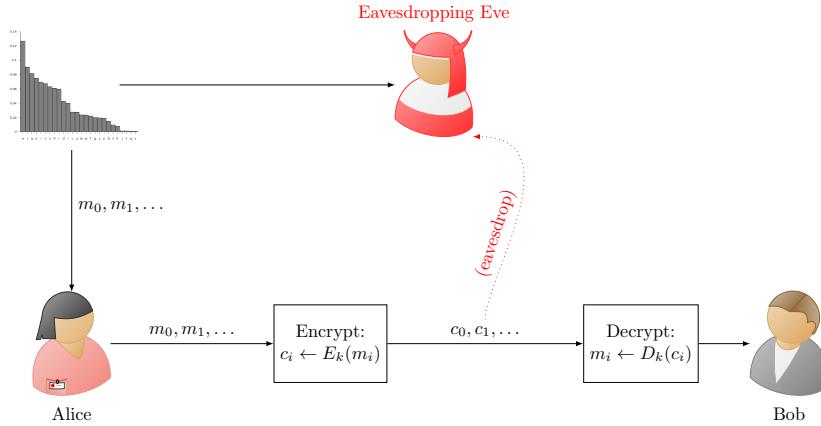


Figure 2.6: The Ciphertext-Only (CTO) attack model. Notice the small image representing the *plaintext distribution*, which is a tiny-version of the letter-distribution graph of Figure 2.5; this is used to sample the plaintext, for both the plaintext messages m_0, m_1, \dots and for sample plaintext messages given to the attacker.

The Ciphertext-Only (CTO) attack model. We discussed above the *letter-frequency attack*, which relied only on access to sufficient amount of ciphertext, and on knowing the letter-distribution of plaintext messages. In subsection 2.3.1, we present *exhaustive search*, an attack requiring the ability to identify correctly-decrypted plaintext (with significant probability). Both of these attacks requires only access to (sufficient) ciphertext, and some knowledge about the plaintext distribution (letter frequencies or ability to identify possible plaintexts). We refer to attacks which require only these ‘minimal’ attacker capabilities, as *ciphertext-only (CTO)* attacks. In particular, ciphertext-only attacks do not require a (plaintext, ciphertext) pair.

To facilitate CTO attacks, the attacker must have some knowledge about the distribution of plaintexts. In practice, such knowledge is typically implied by the specific application or scenario. For example, when it is known that the message is in English, then the attacker can apply known statistics such as the letter-distribution histogram Figure 2.5. For a formal, precise definition, we normally allow the adversary to *pick* the plaintext distribution. Note that this requires defining security carefully, to prevent absurd ‘attacks’, which clearly fail in practice, to seem to fall under the definition.

The Known-Plaintext Attack (KPA) model. In the *known-plaintext attack (KPA) model*, the attacker receives one or multiple pairs of plaintext and the corresponding ciphertext. However, the attacker cannot choose the plaintext; one way to model this is to assume that the plaintext is chosen randomly.

In the historical attacks on cryptographic systems, known plaintext attacks

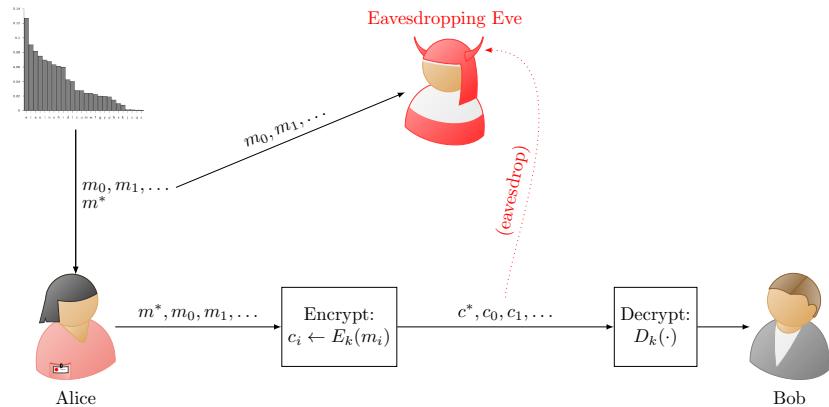


Figure 2.7: The *Known-Plaintext Attack (KPA)* model. As in the CTO model, all plaintext messages, m^* and m_0, m_1, \dots , are chosen from a known *plaintext distribution*, and the eavesdropping-adversary receives their encryptions, c^* and c_0, c_1, \dots , and tried to learn m^* - or something about m^* . In the KPA model, the adversary receives, in addition, the plaintext messages m_0, m_1, \dots , except for the ‘challenge message’ m^* . The plaintext messages m_0, m_1, \dots are *not* given to the CTO attacker.

were sometimes possible, such as when some text was available both in plaintext and in ciphertext. One interesting example is the ‘deciphering’ of the Rosetta stone, which contained the same inscription, engraved in three different ways: one in Greek and two in Egyptian, once using hieroglyphic script and once using Demotic script. This is how archaeologists learned to read hieroglyphic. Previous attempts to decipher hieroglyphic, using ‘ciphertext only’, i.e., without known plaintext, were in vain.

Another example of historical known-plaintext attack was the cryptanalysis of the *Enigma* cipher by the Allies during WW II (Section 1.5). The Germans were often sending encryptions of plaintext which would either be known or could be guessed with reasonable probability. Sometimes the same message was sent to some recipients encrypted, and to others in plaintext; and often the message began with a predictable greeting or otherwise contained some predictable content. The (plaintext m , ciphertext $E_k(m)$) pairs were fed to the *Bombe* devices, which tried all possible keys, until finding the correct key (*exhaustive search*). Note that this was incorrect use of Enigma; following the *conservative design and usage* principle, the exposure of (plaintext, ciphertext) pairs should have been avoided.

In modern applied cryptography, it is very common for the attacker to obtain KPA capabilities. For example, consider the common use of the TLS protocol (Chapter 7) to protect web communication, referred as *https* (for ‘http-secure’). In such usage, the entire traffic between browser and web-server is encrypted by TLS. Typically, this includes images and web-pages (encoded in *HTML*) which

are sent to *all* clients. Namely, the attacker can obtain this plaintext, simply by requesting the same page from the web-server.

The Chosen-Plaintext Attack (CPA) model. In subsection 2.3.2, we discuss the table look-up and time-memory tradeoff attacks; in both of these generic attacks, the adversary must be able to obtain the encryption of one or few specific plaintext messages - the messages used to create the precomputed table. Therefore, these attacks cannot be launched under the Known-Plaintext Attack (CPA) model. Instead, these attacks are facilitated by the *chosen-plaintext attack (CPA)* model.

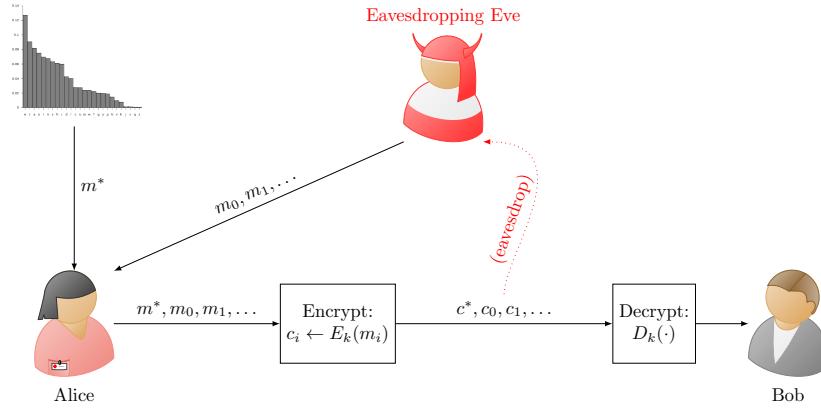


Figure 2.8: The *Chosen-Plaintext Attack (CPA)* model. Here, the attacker can *choose* the plaintext messages m_0, m_1, \dots , in addition to knowing the distribution from which m^* is sampled. Note: in the CPA attack, the attacker controls the plaintext messages given to Alice; however, we usually still consider this attacker to be an ‘eavesdropper’, since the attacker cannot modify or inject messages to the communication between Alice and Bob.

Security against CPA has been studied extensively in modern cryptography, but was not even considered in historical cryptanalysis. For example, in the second world war, cryptanalysts in Bletchley Park made extensive use of some known plaintexts, i.e., used KPA. However, they had no hope to choose the plaintext, i.e., to use CPA. Therefore, they were not able to use CPA attack techniques, e.g., the attacks in subsection 2.3.2.

However, in modern applied cryptography, it is not that unusual for the attacker to obtain CPA capabilities. For example, consider the, as for KPA, the common use of the TLS protocol (Chapter 7) to protect web communication. TLS applies encryption to the traffic from the browser to the server; however, a basic premise of the *http* protocol, is that every web-page (and script) can send requests to any other website, including websites from different domains. Namely, when the user is browsing to the attacker’s web-page, or when the browser is running a script written by the attacker, the attacker can cause the

browser to send arbitrary requests to any website. These requests would often also contain sensitive information, typically a *cookie* attached by the browser to the request. The browser will often also use the same key to encrypt other requests sent by the browser to the same website.

Every cipher vulnerable to CTO attack, is also vulnerable to KPA attack; and every cipher vulnerable to KPA, is also vulnerable to CPA attack. We say that the CPA attack model is *stronger* than the KPA model, and that KPA model is *stronger* than CTO model, and denote this by: $CPA > KPA > CTO$.

Exercise 2.5 ($CPA > KPA > CTO$). *Explain (informally) why every cryptosystem vulnerable to CTO attack, is also vulnerable to KPA, and every cryptosystem vulnerable to KPA, is also vulnerable to CPA.*

The Chosen-Ciphertext Attack (CCA) model. Finally, in the *chosen-ciphertext attack* (CCA) attack model, the attacker has the ability to receive the decryption of arbitrary ciphertexts, chosen by the attacker. Note that there are a few variants to the CCA model, including, regarding whether the adversary also has the ability to receive *encryptions* of random (sampled), known or chosen plaintexts. Another variant is in the *timing* of the choice of ciphertext messages to be decrypted; some CCA models require the adversary to perform these *before* receiving the challenge ciphertext c^* , and other CCA models allow the adversary to select c_i *after* receiving c^* (of course forbidding the use of c^* as one of these ciphertexts).

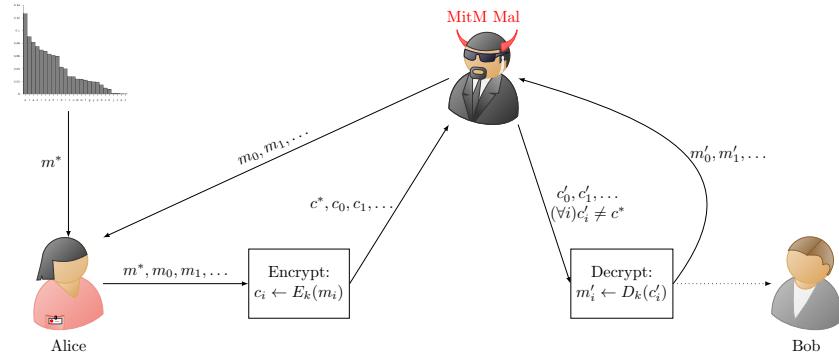


Figure 2.9: The Chosen-Ciphertext Attack (CCA) model. In the CCA model, the adversary can select *ciphertext messages* c'_0, c'_1, \dots , all different from the ‘challenge ciphertext’ c^* , and receive their decryptions $m'_i = D_k(c'_i)$.

We adopt the common definition, where CCA-attackers are also allowed to perform CPA attacks, i.e., the attacker can obtain the encryptions of attacker-chosen plaintext messages. With this definition, trivially, $CCA > CPA$. Combining this with the previous exercise, we have the complete ordering: $CCA > CPA > KPA > CTO$.

2.3 Generic attacks and Effective Key-Length

We discussed several ciphers and attack models; how can we *evaluate the security* of different ciphers, under a given attack model? This is a fundamental challenge in cryptography; we will discuss this challenge in this section, as well as later on, esp. when we introduce our first definition of a cryptographic mechanism - the Pseudo-Random Generator (PRG), in subsection 2.5.2.

One way in which non-experts often compare the security of different ciphers, is using their *key length*. Indeed, as we already mentioned, ciphers using short keys are insecure. In subsection 2.3.1, we present *exhaustive search*, an attack which essentially tries out all the keys until finding the right one.

Exhaustive search is a *generic attack*. Generic attacks work for all (or many) schemes, without depending on the specific design of the attacked scheme, but only on general properties, such as key length and attack model. Exhaustive search works on *most* ciphers and scenarios; it requires the ability to *test candidate keys*. Such ability exists usually, but not always, as we explain.

In subsection 2.3.2 we discuss two other generic attacks: *table look-up* and *time-memory tradeoff*. These attacks further demonstrate, that the attacker's success does not depend only on the key-length - it also depends on the attack model and attacker capabilities, e.g., storage capacity.

Finally, in subsection 2.3.3 we discuss additional challenges in evaluating security of cryptographic mechanisms, and introduce the *effective key length* concept and principle.

2.3.1 The exhaustive-search generic CTO attack

Following Kerckhoffs's principle, we assume henceforth that all designs are known to the attacker; defense is only provided via the secret keys. Therefore, one simple attack strategy is to try to decrypt ciphertext messages using all possible keys, detect the key - or (hopefully few) keys - where the decryption seems to result in possible plaintext, and discard keys which result in clearly-incorrect plaintext. For example, if we know that the plaintext is text in English, we can test candidate plaintexts, e.g., by checking whether their letter distributions are reasonably-close to the distribution of letters in English (Figure 2.5). This attack is called *exhaustive key search*, *exhaustive cryptanalysis* or *brute force attack*; we will use the term *exhaustive search*.

Requirements of Exhaustive Search. Exhaustive search may not work for stateful ciphers, since decryption depends on the state and not only on the key. Furthermore, it should be possible to efficiently identify correct decryption by checking the output of the decryption of the ciphertext. Namely, the decoding of an arbitrary plaintext with an incorrect key should result in clearly-invalid plaintext, with significant probability. Exhaustive search is applicable for any stateless encryption provided that such validation of plaintext is possible; this is why we refer to it as a *generic CTO attack*.

Exhaustive search is *practical* only when the key space is not too large. For now we focus on symmetric ciphers, where the key is usually an arbitrary binary string of given length, namely the key space is 2^l where l is the key length, i.e., the problem is the use of insufficiently-long keys. Surprisingly, designers have repeatedly underestimated the risk of exhaustive search and used ciphers with insufficiently long keys, i.e., insufficiently large key spaces.

Let T_S be the *sensitivity period*, i.e., the duration required for maintaining secrecy, and T_D be the time it takes to test each potential key, by performing one or more decryptions. Hence, the attacker can test T_S/T_D keys out of the key-space containing 2^l keys. If $T_S/T_D > 2^l$, then the attacker can test all keys, and find the key for certain (with probability 1); otherwise, the attacker succeeds with probability $\frac{T_S}{T_D \cdot 2^l}$. By selecting a sufficient key length, we can ensure that the success probability is as low as desired.

For example, consider the conservative assumption of testing a billion keys per second, i.e., $T_D = 10^{-9}$, and requiring the security for three thousand years, i.e., $T_S = 10^{11}$, with probability of attack succeeding at most 0.1%. We find that to ensure security with these parameters against brute force attack, we need keys of length $l \geq \log_2\left(\frac{T_S}{T_D}\right) = \log_2(10^{20}) < 74$ bits.

The above calculation assumed a minimal time to test each key. Of course, attackers will often be able to test many keys in parallel, by using multiple computers and/or parallel processing, possibly with hardware acceleration. Such methods were used during 1994-1999 in multiple demonstrations of the vulnerability of the *Data Encryption Standard (DES)* to different attacks. The final demonstration was exhaustive search completing in 22 hours, testing many keys in parallel using a \$250,000 dedicated-hardware machine ('deep crack') together with distributed.net, a network of computers contributing their idle time.

However, the impact of such parallel testing, as well as improvements in processing time, is easily addressed by reasonable extension of key length. Assume that an attacker is able to test 100 million keys in parallel during the same 10^{-9} second, i.e., $T_D = 10^{-17}$. With the same goals and calculation as above we find that we need keys of length $l \geq \log_2\left(\frac{T_S}{T_D}\right) = \log_2(10^{26}) < 100$. This is far below even the minimal key length of 128 bits supported by the Advanced Encryption Standard (AES). Therefore, exhaustive search is not a viable attack against AES or other ciphers with over 100 bits.

Testing candidate keys. Recall that we assume that decrypting arbitrary ciphertext with an incorrect key should usually result in clearly-invalid plaintext. Notice our use of the term 'usually'; surely there is *some* probability that decryption with the wrong key will result in seemingly-valid plaintext. Hence, exhaustive search may often not return only the correct secret key. Instead, quite often, exhaustive search may return *multiple* candidate keys, which all resulted in seemingly-valid decryption. In such cases, the attacker must now eliminate some of these candidate keys by trying to decrypt additional ciphertexts and

discarding a key when its decryption of some ciphertext appears to result in invalid plaintext.

2.3.2 The Table Look-up and the Time-Memory Tradeoff

Generic CPA attacks

Exhaustive search is very computation-intensive; it finds the key, on the average, after testing half of the keyspace. On the other hand, its storage requirements are very modest, and almost⁶ independent of the key space.

In contrast, the *table look-up attack*, which we next explain, uses $O(2^l)$ ⁷ memory, where l is the key length, but only table-lookup time. However, this requires ciphertext of some pre-defined plaintext message, which we denote p^* . This can be achieved by an attacker with *Chosen Plaintext Attack (CPA)* capabilities, or whenever the attacker can obtain encryptions of some well known message p^* . Many communication protocols use predictable, well-known messages at specific times, often upon connection initialization, which provides the attacker with encryptions of this *predictable* known plaintext message p^* - and suffice for this attack.

In the table look-up attack, the attacker first *precomputes* $T(k) = E_k(p^*)$ for every key k . Later, the attacker asks for the encryption of the same plaintext p , using the unknown secret key k^* ; let $c^* = E_{k^*}(p^*)$ denote the received ciphertext. The attacker now looks up c^* in the table T and identifies all the keys k such that $c^* = T(k)$. The number of matching keys is usually one or very small, allowing the attacker to quickly rule out the incorrect keys, usually by decrypting some additional ciphertext messages.

The table look-up attack requires $O(2^l)$ storage to ensure $O(1)$ computation, while the exhaustive search attack uses $O(1)$ storage and $O(2^l)$ computations. Several more advanced generic attacks allow different *tradeoffs* between the computing time and the amount of storage (memory) required for the attack. The first and most well known *time-memory tradeoff* attack was presented by Martin Hellman [159]. Later works presented other tradeoff attacks, such as the *time/memory/data tradeoff* of [55] and the *rainbow tables* technique of [251]. Unfortunately, we will not be able to cover these interesting attacks, and the readers are encouraged to read these (and other) papers presenting them. Note that these tradeoffs use *cryptographic hash* functions, which we discuss in Chapter 3.

2.3.3 Effective key length

Cryptanalysis, i.e., developing attacks on cryptographic mechanisms, is a large part of the research in applied cryptography; it includes generic attacks such as these presented earlier in this section, as well as numerous attacks which are tailored to a specific cryptographic mechanism. This may look surprising; why

⁶Exhaustive search needs storage for the key guesses.

⁷See Section A.1 for the big-O notation, $O(2^l)$.

publish attacks? Surely the goal is not to help attacks against cryptographic systems?

Cryptanalysis facilitates two critical decisions facing designers of security systems which use cryptography: *which cryptographic mechanism* to use, and *what parameters* to use, in particular, *which key length* to use.

Let us focus first on the key length. All too often, when cryptographic products and protocols are mentioned in the popular press, the key-length in use is mentioned as an indicator of their security. Furthermore, this is sometimes used to argue for the security of the cryptographic mechanism, typically by presenting the number of different key values possible with a given key length. The number of different keys, is the time required for the exhaustive search attack, and has direct impact on the resources required by the other generic attacks we discussed. Hence, clearly, *keys must be sufficiently long* to ensure security. But how long?

It is *incorrect* to compare the security of two different cryptographic systems, which use different cryptographic mechanisms (e.g., ciphers), by comparing the key length used in the two systems. Let us give two examples:

1. We saw that the general monoalphabetic substitution cipher (subsection 2.1.3) is insecure, although its key space is relatively large. We could easily increase the key length, e.g., by adding more symbols, e.g., use different symbols for lowercase and uppercase letters; but this will not significantly improve security.
2. The key length used by symmetric cryptosystems, as discussed in this chapter, rarely exceed 300 bits, and is usually much smaller - 128 bits is common; more bits are simply considered unnecessary. In contrast, asymmetric, public-key cryptography is usually used with longer keys - often much longer, depending on the specific public key cryptosystem; see Table 6.1.

It is useful to compare the security of different cryptosystems, when each is used with a specific key-length - e.g., with comparable efficiency. As explained above, using the key-length alone would be misleading. One convenient, widely used measure for the security of a given cryptosystem, used with a specific key length, is called the *effective key length*; essentially, this uses exhaustive search as a measure to compare against.

We say that a cipher using k -bit keys has *effective key length* l if the most effective attack known against it takes about 2^l operations, where $k \geq l$. We expect the effective key length of good symmetric ciphers to be close to their real key length, i.e., l should not be ‘much smaller’ compared to k . For important symmetric ciphers, any attack which increases the gap between k and l would be of great interest, and as the gap grows, there will be increasing concern with using the cipher. The use of key lengths which are 128 bits or more leaves a ‘safety margin’ against potential better future attacks, and gives time to change to a new cipher when a stronger, more effective attack is found.

Note that, as shown in Table 6.1, for asymmetric cryptosystems, there is often a large gap between the real key length l and the effective key length k . This is considered acceptable, since the design of asymmetric cryptosystems is challenging, and it seems reasonable to expect attacks with performance much better than exhaustive search. In particular, in most public key systems, the secret key is not an arbitrary, random binary string.

Note that the evaluation of the effective key length, depends on the *attack model*; there are often attackers with much smaller effective-key length, when assuming a stronger attack model, e.g., CPA compared to KPA or CTO. One should therefore also take into account the expected attack model.

Also, notice that the effective key length measure compares based on the *time* required for the attack; it does not allow for comparing different resources, for example, time-memory tradeoff.

Normally, we select sufficient key length to ensure security against any conceivable adversary, e.g., leaving a reasonable margin above effective key length of say 100 bits; a larger margin is required when the *sensitivity period* of the plaintext is longer. The cost of using longer keys is often justified, considering the damages of loss of security and of having to change in a hurry to a cipher with longer effective key length, or even of having to use longer keys.

In some scenarios, however, the use of longer keys may have significant costs; for example, doubling the key length in the RSA cryptosystem increases the computational costs by about six. We therefore may also consider the risk from exposure, as well as the resources that a (rational) attacker may deploy to break the system. This is summarized by the following principle.

Principle 5 (Sufficient effective key length). *Deployed cryptosystems should have sufficient effective key length to foil feasible attacks, considering the maximal expected adversary resources and most effective yet feasible attack model, as well as cryptanalysis and speed improvements expected over the sensitivity period of the plaintext.*

Experts, as well as standardization and security organizations, publish estimates of the required key length of different cryptosystems (and other cryptographic schemes); we present a few estimates in Table 6.1.

2.4 Unconditional security and the One Time Pad (OTP)

The exhaustive search and table look-up attacks are generic - they do not depend on the specific design of the cipher: their complexity is merely a function of key length. This raises the natural question: is every cipher breakable, given enough resources? Or, can encryption be secure *unconditionally* - even against an attacker with unbounded resources (time, computation speed, storage)?

We next present such an unconditionally secure cipher, the *one time pad (OTP)*. The one time pad is often attributed to a 1919 patent by Gilbert Vernam [314], although some of the critical aspects may have been due to Mauborgne [40], and in fact, the idea was already proposed by Frank Miller in

1882 [39]; we again refer readers to the many excellent references on history of encryption, e.g., [184, 297].

The one time pad is not just unconditionally secure - it is also an exceedingly simple and computationally efficient cipher. Specifically:

Encryption: To encrypt a message, compute its bitwise XOR with the key.

Namely, the encryption of each plaintext bit, say m_i , is one ciphertext bit, c_i , computed as: $c_i = m_i \oplus k_i$, where k_i is the i^{th} bit of the key.

Decryption: Decryption simply reverses the encryption, i.e., the i^{th} decrypted bit would be $c_i \oplus k_i$.

Key: The key $k = \{k_1, k_2, \dots\}$ should consist of independently drawn fair coins, and its length must be at least as long as that of the plaintext. Notice, that the key should be - somehow - shared between the parties, which may be a challenge in many scenarios.

See illustration in Figure 2.10.

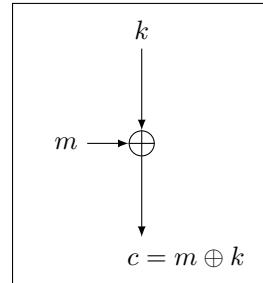


Figure 2.10: The one time pad (OTP) cipher - an unconditionally-secure stream cipher: $c = m \oplus k$ (bit-wise XOR).

The correctness of OTP, i.e., the fact that decryption recovers the plaintext correctly, follows from the properties of exclusive OR. Namely, given $c_i = m_i \oplus k_i$, the corresponding decrypted bit is $c_i \oplus k_i = (m_i \oplus k_i) \oplus k_i = m_i$, as required.

The unconditional security of OTP also follows from properties of XOR; let us explain it, albeit informally. First, OTP handles each bit completely independently of others, so we can focus on the security of a particular bit m_i , encrypted as $c_i = m_i \oplus k_i$. Recall that each key bit k_i is selected randomly, and suppose, for simplicity, that the message bit m_i was also selected randomly. Namely, both bits can be either 0 or 1 with probability half. Given c_i , there are two equally likely conclusions: either $k_i = 0$ and therefore $m_i = c_i$, or $k_i = 1$ and therefore $m_i = 1 - c_i$. Namely, seeing c_i does not change our knowledge about m_i , simply since k_i is a random bit. The precise argument is similar, mainly avoiding the assumption that the message bit is fair, i.e., allowing some prior knowledge about the probability that $m_i = 1$; the argument would show that there is no extra knowledge about m_i from observing c_i .

The unconditional secrecy of OTP was recognized early on, and established rigorously in a seminal paper published in 1949 by Claude Shannon [292]. In that paper, Shannon also proved the more challenging fact that *every unconditionally-secure cipher must have keys as long as the plaintext*; namely, as long as unconditional secrecy is required, this aspect cannot be improved. Interested readers can find this proof in textbooks on cryptography, e.g., [303].

Interestingly, OTP is actually a very special case of the Generalized-Caesar cipher cipher (subsection 2.1.2), defined by: $E_k^{\text{GC}-n}(p) = p + k \pmod n$, for $n = 2$. Specifically:

$$c_i = m_i \oplus k_i = m_i + k_i \pmod 2 = E_{k_i}^{\text{GC}-2}(m_i) \quad (2.13)$$

Indeed, the Generalized-Caesar cipher cipher is unconditionally secure - if each key k_i is chosen randomly from $\{0, 1, \dots, n - 1\}$ and used to encrypt a single plaintext ‘letter’ $m_i \in \{0, 1, \dots, n - 1\}$!

The cryptographic literature has many beautiful results on unconditional security. However, it is rarely practical to use such long keys, and in practice, adversaries - like everyone else - have limited computational abilities. Therefore, in this textbook, we focus on computationally-bounded adversaries.

While the key required by OTP makes its use rarely practical, we next show a computationally-secure variant of OTP, where the key can be much smaller than the plaintext, and which can be used in practical schemes. Of course, this variant is - at best - secure only against computationally-bounded attackers.

2.4.1 OTP is a Stateful Cryptosystem / Stream Cipher

The attentive reader may have noticed that OTP does not conform to our definition of shared-key cryptosystem in Definition 2.1. Specifically, that definition did not allow the encryption process to maintain any synchronized state between the parties. However, the OTP cipher requires sender and recipient to maintain a (synchronized) counter i , identifying the bits already used in the OTP design; namely, it requires such persistent state, i.e., is a *stateful* design. We therefore extend the definition so that the encryption and decryption operations may have another input and another output. The additional input is the current state, and the additional output is the next state; both are from a set S of possible states.

Definition 2.3 (Stateful shared-key cryptosystem). *A stateful shared-key cryptosystem is a pair of keyed algorithms, $\langle E, D \rangle$. Algorithm E (and D) receives as input plaintext (ciphertext for D), and input state $s \in S$, where S is the set of possible states. Algorithm E (and D) outputs ciphertext (plaintext for D), and output state $s' \in S$. We say that $\langle E, D \rangle$ ensures correctness if for every message $m \in M$, key $k \in K$ and input state $s \in S$ holds that if $(c, s') = E_k(m, s)$ then $(m, s') = D_k(c, s)$.*

Exercise 2.6. Define the stateful encryption and decryption functions $\langle E, D \rangle$ for the OTP cipher.

Solution: We use the index i of the next bit to be encrypted as the state, initialized with $i = 1$. Namely, encryption $E_k(m_i, i)$ returns $(m_i \oplus k_i, i + 1)$, and decryption $D_k(c_i, i)$ returns $(c_i \oplus k_i, i + 1)$. \square

Stream ciphers. The one-time pad (OTP) is often referred to as a *stream cipher*. We use the term stream ciphers⁸ to refer to stateful cryptosystems that use bit-by-bit encryption process, i.e., a stream cipher is a process of mapping each plaintext bit m_i to a corresponding ciphertext bit c_i .

Since stream ciphers map each plaintext bit to a ciphertext bit, and we require decryption to be correct (recover the plaintext), then the mapping from plaintext to ciphertext cannot be randomized; but obviously, it also cannot be the same for all bits, or decryption would be trivial. It follows that *stream ciphers must be stateful*. For example, with one time pad, not only the parties must share a key as long as all plaintext bits, the parties must also maintain an exact, *synchronized count* of the number of key bits used so far, to ensure correct decryption.

Reuse of key bits with the one-time-pad is also *insecure*. In particular, suppose the design uses the same key bit k_i to encrypt both m_i and m_{i+1} , i.e., $c_i = m_i \oplus k_i$ but also $c_{i+1} = m_{i+1} \oplus k_i$, reusing k_i . Then an attacker knowing one known plaintext, e.g., m_i and c_i , and eavesdropping on c_{i+1} , can find m_{i+1} . Specifically, in this case, $m_{i+1} = c_{i+1} \oplus (m_i \oplus c_i)$. See Example 2.5 for a more realistic example of this vulnerability.

Stream ciphers are often used in applied cryptography, and esp. in hardware implementations, mainly due to their simple and efficient hardware implementation. Rarely, we use the OTP (or another unconditionally-secure cipher), but much more commonly, a stream cipher with a bounded-length key, providing ‘only’ computational security. In the following section we introduce *pseudo-random generators (PRG)* and *pseudorandom functions (PRF)*, and show how to use either of them to design a *bounded key length stream cipher*.

2.5 Pseudo-Randomness, Indistinguishability and Asymptotic Security

Randomness is widely used in cryptography - for example, the one time pad cipher (Section 2.4) uses random keys to ensure unconditional secrecy. In this section, we introduce *pseudo-randomness*, a central concept in cryptography, and three types of pseudo-random schemes: *pseudo-random generator (PRG)*, *pseudorandom function (PRF)* and *pseudo-random permutation (PRP)*. We also introduce the central technique of *indistinguishability test*, which is central to the definitions of these three pseudo-random schemes - as well as to the definition of secure encryption, which we present later.

⁸Other authors use the term stream cipher also for cryptosystems that use byte-by-byte or block-by-block encryption, essentially as a synonym to stateful encryption.

2.5.1 Pseudo-Random Generators and their use for Bounded Key-length Stream Ciphers

In this subsection we introduce the *Pseudo-Random Generator (PRG)*. A PRG is one of the simpler cryptographic definitions, and hence we consider it a good choice for the first definition; however, it is still not *that* easy. Hence, in this subsection we introduce PRGs but define them only informally. We focus on the classical application of PRGs, which is to construct a *stream cipher*.

We have already seen a stream cipher: the one time pad (OTP). The OTP has the advantage of being unconditionally-secure, but also the disadvantage of requiring the parties to share a secret key which is as long as all the plaintext bits they may need to encrypt, i.e., to share a key of unbounded length. . In contrast, stream ciphers constructed from PRGs only require the parties to share a *bounded-length key*; this is a critical advantage over the OTP. On the other hand, stream ciphers with bounded key-length, such as these constructed from PRGs, *cannot* be unconditionally-secure; instead, they can only be *computationally secure* - i.e., secure only assuming that the attacker has limited computational capabilities.

In this section, we will see one method to implement a bounded-key-length stream cipher from a *pseudo-random generator (PRG)*. PRGs have other important applications, and are one of the cryptographic mechanism whose definition is least-complex; therefore, they are a good way to introduce the more complex - and even more important - cryptographic mechanisms of *pseudorandom functions (PRF)*, *pseudo-random permutations (PRP)* and *block ciphers*, which we present later.

PRG: intuitive definition. In this subsection, we only introduce PRGs informally, focusing on their use in the construction of bounded-key-length stream ciphers. We focus on the following simple definition,

Definition 2.4 (Informal definition of a (stateless) PRG). *Given any random input string s , usually referred to as the seed, a PRG f_{PRG} outputs a longer string, i.e., $(\forall s \in \{0, 1\}^*) (|f_{PRG}(s)| > |s|)$. Furthermore, if s is a random string (of $|s|$ bits), then $f_{PRG}(s)$ is pseudo-random. Intuitively, this means that $f_{PRG}(s)$ cannot be efficiently distinguished from a true random string of the same length $|f_{PRG}(s)|$.*

We define these concepts of efficient distinguishing and PRG precisely quite soon, in subsection 2.5.2. However, first, let us see that there are some other ways to define PRG. Let us mention one of these variants, which is widely used; to avoid confusion, we refer to this variant as a *stateful PRG*, and, where relevant, refer to PRG defined as in Definition 2.4.

Variant: stateful PRG. A stateful PRG f_{SPRG} is a function that receives two inputs, a *current state* (or seed) s and a ‘timestamp’ t , and outputs a pseudo-random string r and a *new state* s' . We use the *dot notation* to refer

to the two outputs, $f_{SPRG}.r$ and $f_{SPRG}.s'$. The timestamp input t is optional; if used, we require the outputs for a given timestamp t_1 to be pseudorandom, even if the adversary is given the outputs $f_{SPRG}(s, t_2)$ for a different timestamp $t_2 \neq t_1$.

Building stream cipher from a PRG. To obtain a stream cipher, we require a PRG which produces a pseudo-random string as long as the plaintext⁹. We then XOR each plaintext bit with the corresponding pseudo-random bit, as shown in Figure 2.11.

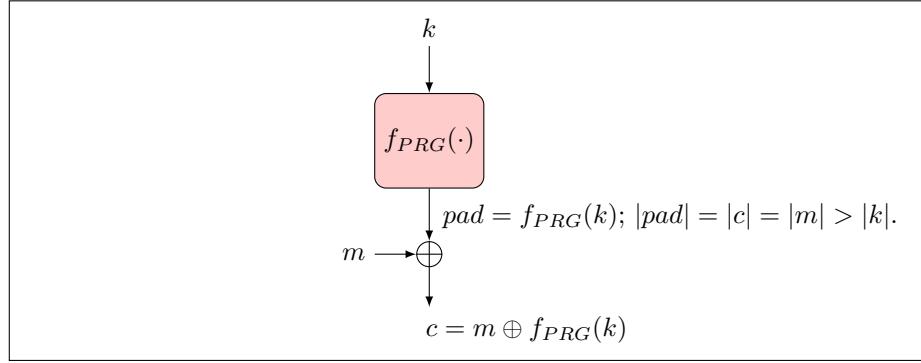


Figure 2.11: PRG-based Stream Cipher. The input to the PRG is usually called either key or seed; if the input is random, or pseudo-random, then the (longer) output string is pseudo-random. The state includes the current bit index i .

The pseudo-random generator stream cipher is very similar to the OTP; the only difference is that instead of using a truly random sequence of bits to XOR the plaintext bits, we use the output of a *Pseudo-Random Generator (PRG)*. If we denote the i^{th} output bit of $f_{PRG}(k)$ by $f_{PRG}(k)_i$, we have that the i^{th} ciphertext bit c_i , is defined as: $c_i = m_i \oplus f_{PRG}(k)_i$. This is a shared-key stream cipher in which k is the shared key, quite similar to the OTP. Specifically, the state is the index of the bit i , encryption $E_k(m_i, i)$ returns $(m_i \oplus f_{PRG}(k)_i, i+1)$, and decryption $D_s(c_i, i)$ returns $(c_i \oplus f_{PRG}(k)_i, i+1)$. Note that this may require us to compute the value of $f_{PRG}(k)$ each time we need a specific bit, or to store (all or parts of it).

In subsection 2.5.4 below, we properly define pseudo-random generators (PRGs), using the *PRG indistinguishability test*, which we present in subsection 2.5.3. But let us first introduce the ingenious *concept* of indistinguishability test, which was introduced by Alan Turing.

⁹The output of some PRGs may be only slightly longer than their input, e.g., one bit longer. However, we can use such a PRG to construct another PRG, whose output length is longer (as a function of its input length). The details and proof are beyond our scope; see, e.g., [140]).

2.5.2 The Turing Indistinguishability Test

Intuitively, a PRG is an *efficient* algorithm, whose input is a binary string s , called *seed* (or sometimes *key*); if the input is either *random* or *pseudo-random*, then the (longer) output string is *pseudo-random*. In order to turn this intuitive description into a definition, we first define clearly the notions of ‘efficient’ and ‘pseudorandom’. We discuss these notions in the following subsections; in this subsection, we first present the ingenious but non-trivial concept of *indistinguishability test*, which is key to the notion of pseudorandomness - and to many definitions in cryptography.

The first indistinguishability test was the Turing Indistinguishability test, proposed by Alan Turing in 1950, in a seminal paper [306], which lay the foundations for artificial intelligence. Turing proposed the test, illustrated in Figure 2.12, as a possible definition of an *intelligent machine*. Turing referred to this test as the *imitation test*; another name often used for this test is simply the *Turing test*.

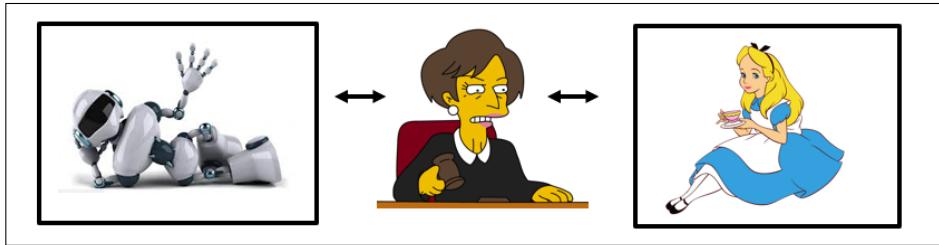


Figure 2.12: The Turing Indistinguishability Test. A machine is considered *intelligent*, if a *distinguisher* (judge) cannot determine in which box is the machine and in which is a human. Turing stipulated that communication between the distinguisher and the boxes would only be in printed form, to avoid what he considered ‘technical’ challenges such as voice recognition.

Many cryptographic mechanisms are defined using indistinguishability tests. These tests are similar, in their basic concept, to the Turing indistinguishability test. The following subsection presents the first such test, which is testing for the important property of *pseudorandomness*.

2.5.3 PRG indistinguishability test

We now return to the discussion of pseudorandomness, and define the *PRG indistinguishability test*, illustrated in Figure 2.13. The pseudorandom test is similar to the Turing indistinguishability test in Figure 2.12, in the sense that a distinguisher is asked to identify which is the ‘true’ (intelligent person in Turing test, and random sequences here) and who is the ‘imitation’ (machine in Turing test, and sequences output by function f here).

Intuitively, a pseudo-random generator is a function f whose input is a ‘short’ random bit string $x \xleftarrow{\$} \{0,1\}^n$, and whose output a *longer* string $f(x) \in \{0,1\}_n^l$

s.t. $l_n > n$, which is *pseudo-random* - i.e., *indistinguishable from a random string* (of the same length l_n).

But what does it mean for the output to be indistinguishable from random? This is defined by the *PRG indistinguishability test*, which we next define - and which, in concept, quite resembles the Turing indistinguishability test, although the details are different. The similarity can be seen from comparing the illustration of the PRG indistinguishability test in Figure 2.13, to that of the Turing test in Figure 2.12.

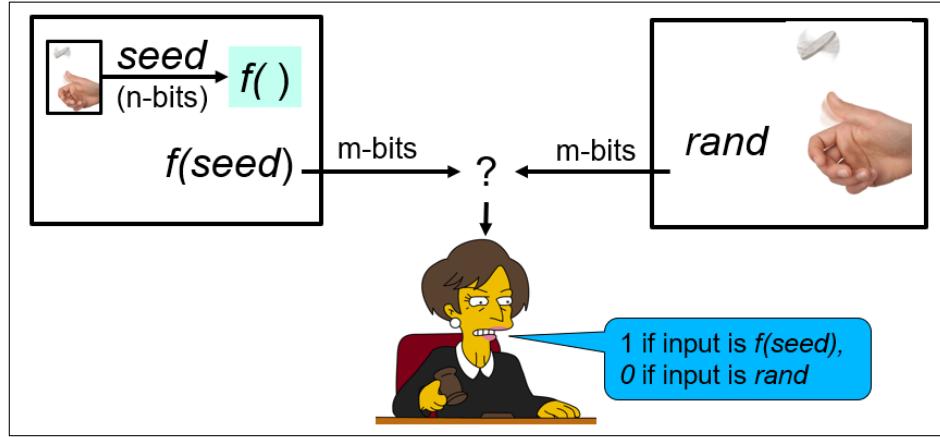


Figure 2.13: Intuition for the Pseudo-Random Generator (PRG) Indistinguishability Test. Intuitively, $f : \{0, 1\}^* \rightarrow \{0, 1\}^*$ is a (secure) pseudo-random generator (PRG), if an efficient distinguisher D can't effectively distinguish between $f(x)$, for a random input x , and a random string of the same length $|f(x)|$.

In order to turn this intuition into a definition of a (secure) *Pseudo-Random Generator (PRG)*, we must specify precisely the *capabilities* of the distinguisher and *criteria* for the outcome of the experiment, i.e., when would we say that f is indeed a (good/secure) *PRG*. We next discuss these two aspects in the following subsection, where we finally present definitions for (secure) PRG.

2.5.4 Defining Secure Pseudo-Random Generator (PRG)

We now finally define (secure) pseudo-random generator (PRG). We first define the distinguisher *capabilities*; next, we define the *advantage* $\varepsilon_{D,f}^{PRG}(n)$ of D for function f and inputs of length n ; and finally we define a (secure) PRG.

Distinguisher capabilities. We model the distinguisher as an *algorithm*, denoted D , which receives the a binary string - either a random string or the ‘pseudorandom’ output of the PRG f - and outputs its evaluation, which should be 0 if given truly random string, and 1 otherwise, i.e., if the input is not truly random. The distinguisher algorithm D has to be *efficient* (or *PPT*). The terms

efficient algorithm and *PPT* (*Probabilistic Polynomial Time*) algorithm are crucial to definitions of *asymptotic security*, which we use in this textbook; see Section A.1.

PRG: the advantage of D for f . Before we define the criteria for a function f to be considered a (secure) *Pseudo-Random Generator (PRG)*, we notice that by simply randomly guessing, the distinguisher may succeed with probability $\frac{1}{2}$. Namely, succeeding with probability $\frac{1}{2}$ does not imply a vulnerability, and should not result in any advantage for D .

We therefore define the *advantage* $\varepsilon_{D,f}^{PRG}(n)$ of D for function f as the probability that D outputs 1 (correctly) when given the output of the pseudorandom function $f(x)$, for random input x , *minus* the probability that D outputs 1 (incorrectly) when given a truly random string r . As required, this gives no advantage to a distinguisher which simply guesses the bit. However, this introduces a challenge: how should we choose x and r ? We solve this challenge with the following assumption on f .

Length-uniform assumption. We simplify the definitions by assuming that f is a *length-uniform* function, i.e., for every input of length n , the output would be of the same length l_n .

We can now present the definition of the *advantage* $\varepsilon_{D,f}^{PRG}(n)$: the probability that D *correctly* outputs 1 when given $f(x)$ for random n -bit input $x \xleftarrow{\$} \{0,1\}^n$, *minus* the probability that D *incorrectly* outputs 1 when given a random l_n -bit string r .

Definition 2.5. Let $f : \{0,1\}^* \rightarrow \{0,1\}^*$ be a length-uniform function, i.e., if $|x| = n$ then $|f(x)| = l_n$, and let D be an algorithm. The PRG-advantage of D for f is denoted $\varepsilon_{D,f}^{PRG}(n)$ and defined as:

$$\varepsilon_{D,f}^{PRG}(n) \equiv \Pr_{x \xleftarrow{\$} \{0,1\}^n} [D(f(x)) = 1] - \Pr_{r \xleftarrow{\$} \{0,1\}^{l_n}} [D(r) = 1] \quad (2.14)$$

The probabilities in Equation 2.14 are computed over uniformly-random n -bit binary string s (seed), uniformly-random $l_n = |f(1^n)|$ -bit binary string r , and uniformly-random coin tosses of the distinguisher D , if D uses random bits. Note that the length of the output of f depends only on the length of the input, hence, our use of l_n .

Definition of Secure PRG. Finally, let's define a secure PRG. The definition assumes both the PRG and the distinguisher D are *efficient (PPT)*, i.e., their running time is bounded by a polynomial in the input length. Note that the PRG must be deterministic, but the distinguisher D may be probabilistic (randomized).

Definition 2.6 (Secure Pseudo-Random Generator (PRG)). A length uniform function $f : \{0,1\}^* \rightarrow \{0,1\}^*$, s.t. $(\forall x \in \{0,1\}^n) l_n = |f(x)|$, is a secure

Pseudo-Random Generator (PRG), if it is efficiently-computable ($f \in PPT$), length-increasing ($l_n > n$) and ensures indistinguishability, i.e., for every distinguisher $D \in PPT$, the advantage of D for f is negligible, i.e., $\varepsilon_{D,f}^{PRG}(n) \in NEGL$, where $\varepsilon_{D,f}^{PRG}(n)$ is defined as in Equation 2.14.

The term ‘secure’ is often omitted; i.e., when we simply say that algorithm f is a pseudo-random generator (PRG), this implies that it is a secure PRG.

Exercise 2.7. Let $x \in \{0,1\}^n$. Show that the following are not PRGs: (a) $f_a(x) = 3x \bmod 2^{n+2}$ (using standard binary encoding), (b) $f_b(x) = 3x \bmod 2^{n+1}$ (similarly), and (c) $f_c(x) = x + \text{parity}(x)$, where $\text{parity}(x)$ returns 1 if the number of 1 bits in x is odd and 0 if it is even.

Solution for part (a): Notice that here we view x as a number encoded in binary, whose value can be between 0 and $2^n - 1$.

A simple distinguisher D_a for f_a is: $D_a(y)$ outputs 1 (i.e., pseudo-random) if $y \bmod 3 = 0$, otherwise, it outputs 0 (i.e., random). Let us show why this distinguisher has significant advantage.

First notice that if $y = f_a(x)$, then $D_a(y)$ outputs 1 (correctly), for every $x \in \{0,1\}^n$. This holds since $3x < 2^{n+2}$, and hence, $y = f_a(x) = 3x \bmod 2^{n+2} = 3x$. Namely, $D_a(y) = D_a(3x) = 1$, by definition of D_a .

It remains to show that the probability that $D_a(r) = 1$, for $r \xleftarrow{\$} \{0,1\}^{n+2}$, is significantly less than 1. If $2^{n+2} \bmod 3 = 1$, this probability is exactly third; otherwise, the probability is only $2^{-(n+2)}$ higher. In either case, the probability is definitely much less than 1!

Therefore, $\varepsilon_{D_a,f_a}^{PRG}(n) \geq 1 - (\frac{1}{3} + 2^{-(n+2)}) > \frac{1}{2}$, i.e., is clearly non-negligible. \square

2.5.5 Secure PRG Constructions

Note that we did not present a *construction* of a secure PRG. In fact, if we could have presented a provably-secure construction of a secure PRG, satisfying Def. 2.6, this would have immediately proven that $P \neq NP$, solving the most well-known open problems in the theory of complexity. Put differently, if $P = NP$, then there cannot be any secure PRG algorithm (satisfying Def. 2.6).

Since $P \stackrel{?}{=} NP$ is believed to be a very hard problem, proving that a given construction is a (secure) PRG must also be a very hard problem, and unlikely to be done as a side-product of proving that some function is a PRG. Similar arguments apply to most of the cryptographic mechanisms we will learn in this book, including secure encryption, when messages may be longer than the key. (The one time pad (OTP) *is* secure encryption, but its key is as long as the plaintext.)

What *is* possible is to present a *reduction-based* construction of a PRG, namely, construction of PRG from some other cryptographic mechanism, along with a *proof* that the PRG is secure *if* that other mechanism is ‘secure’. For example, see [140] for a construction of PRG f from a different cryptographic

mechanism called *one-way function* (which we discuss in Section 3.4), and a reduction proof, showing if the construction of f uses a OWF f_{OWF} , then the resulting function f would be a PRG. We will also present few reduction proofs, for example, later in this section we prove reductions which construct a PRG from other cryptographic mechanisms such as a *pseudorandom function (PRF)*, see Exercise 2.13, and a *block-cipher*. Courses, books and papers dealing with cryptography, are full of reduction proofs, e.g., see [140, 141, 303].

Unfortunately, there is no proof of the *existence* of any of these - one-way function, PRF, block-cipher or most other cryptographic schemes. Indeed, such proofs would imply $P \neq NP$. Still, reduction proofs are the main method of ensuring the security of most cryptographic mechanisms - by showing that they are ‘at least as secure’ as another cryptographic mechanism, typically a mechanism whose security is well established (e.g., by failure of extensive cryptanalysis efforts).

For example, it seems ‘easier’ to design a one-way function than a PRG. If so, then we could obtain a PRG using a given one-way function, and a construction of a PRG from a one-way function. As a more practical example, block-ciphers are standardized, with lots of cryptanalysis efforts; therefore, block ciphers are a good basis to use for building other cryptographic functions.

Let us give an important example of reduction-based proof which is specific to PRGs. This is a construction of a PRG whose output is significantly larger than its input, from a PRG whose output is only one-bit longer than its input. Unfortunately, the construction and proof are beyond our scope; see [140]. However, the following exercise (Exercise 2.8) proves a related - albeit much simpler - reduction, showing that a PRG G from n bits to $n + 1$ -bits, gives also a PRG G' from $n + 1$ bits to $n + 2$ -bits, simply by exposing one bit. In other words, this shows that a PRG may expose one (or more) bits - but remain a PRG.

Exercise 2.8. Let $f : \{0, 1\}^n \rightarrow \{0, 1\}^{n+1}$ be a secure PRG. Is $f' : \{0, 1\}^{n+1} \rightarrow \{0, 1\}^{n+2}$, defined as $f'(b \# x) = b \# f(x)$, where $b \in \{0, 1\}$, also a secure PRG?

Solution: Yes, if f is a PRG then f' is also a PRG. First, recall the PRG-advantage (Equation 2.14) for distinguisher D , using $l_n = n + 1$:

$$\varepsilon_{D,f}^{PRG}(n) \equiv \Pr_{x \xleftarrow{\$} \{0,1\}^n} [D(f(x))] - \Pr_{r \xleftarrow{\$} \{0,1\}^{n+1}} [D(r)] \quad (2.15)$$

Next, rewrite Equation 2.14 for f' and distinguisher D' , by substituting f by f' , x by x' , n by $n + 1$ and l_n by $n + 2$:

$$\varepsilon_{D',f'}^{PRG}(n+1) \equiv \Pr_{x' \xleftarrow{\$} \{0,1\}^{n+1}} [D'(f'(x'))] - \Pr_{r' \xleftarrow{\$} \{0,1\}^{n+2}} [D'(r')] \notin NEGL(n) \quad (2.16)$$

We next present a simple construction of a distinguisher D (for f), using, as a subroutine (oracle), a given distinguisher D' (for f'):

$$D(y) \equiv \left\{ \text{Return } D'(b + y) \text{ where } b \xleftarrow{\$} \{0, 1\} \right\} \quad (2.17)$$

Clearly D is efficient (PPT) if and only if D' is efficient (PPT).

We prove that $\varepsilon_{D',f'}^{PRG}(n+1) = \varepsilon_{D,f}^{PRG}(n)$, therefore, f' is a PRF if and only if f is a PRG. We begin by developing the first component of Equation 2.15:

$$\Pr_{x \xleftarrow{\$} \{0,1\}^n} [D(f(x))] = \Pr_{x \xleftarrow{\$} \{0,1\}^n} [D'(b \# f(x)) | b \xleftarrow{\$} \{0,1\}] \quad (2.18)$$

$$= \Pr_{x \xleftarrow{\$} \{0,1\}^n} [D'(f'(b+x)) | b \xleftarrow{\$} \{0,1\}] \quad (2.19)$$

$$= \Pr_{x' \xleftarrow{\$} \{0,1\}^{n+1}} D'(f'(x')) \quad (2.20)$$

We now develop the other component of Equation 2.15:

$$\Pr_{r \xleftarrow{\$} \{0,1\}^{n+1}} [D(r)] = \Pr_{r \xleftarrow{\$} \{0,1\}^{n+1}} [D'(b+r) | b \xleftarrow{\$} \{0,1\}] \quad (2.21)$$

$$= \Pr_{r' \xleftarrow{\$} \{0,1\}^{n+2}} D'(r') \quad (2.22)$$

Now substitute the two components in Equation 2.15:

$$\varepsilon_{D,f}^{PRG}(n) \equiv \Pr_{x \xleftarrow{\$} \{0,1\}^n} [D(f(x))] - \Pr_{r \xleftarrow{\$} \{0,1\}^{n+1}} [D(r)] \quad (2.23)$$

$$= \Pr_{x' \xleftarrow{\$} \{0,1\}^{n+1}} D'(f'(x')) - \Pr_{r' \xleftarrow{\$} \{0,1\}^{n+2}} D'(r') \quad (2.24)$$

$$\equiv \varepsilon_{D',f'}^{PRG}(n+1) \quad (2.25)$$

Hence, $\varepsilon_{D,f}^{PRG}(n) = \varepsilon_{D',f'}^{PRG}(n+1)$, namely, f is a PRG if and only if f' is a PRG. \square

Feedback Shift Registers (FSR). There are many proposed designs for PRGs. Many of these are based on Feedback Shift Registers (FSRs), with a known linear or non-linear feedback function f , as illustrated in Fig. 2.14. For Linear Feedback Shift Registers (LFSR), the feedback function f is simply the XOR of some of the bits of the register. Given the value of the initial bits r_1, r_2, \dots, r_l of an FSR, the value of the next bit r_{l+1} is defined as: $r_{l+1} = f(r_1, \dots, r_l)$; and following bits are defined similarly: $(\forall i > l) r_i = f(r_{i-l}, \dots, r_{i-1})$.

FSRs are well-studied with many desirable properties. However, by definition, their state is part of their output. Hence, they cannot directly be used as cryptographic PRGs. One solution is to define another function g over the bits of the register, which outputs one or more bits which should hopefully be pseudorandom. The following exercise gives some examples.

Exercise 2.9. For each of the following pairs of functions, show why they are not a secure PRG:

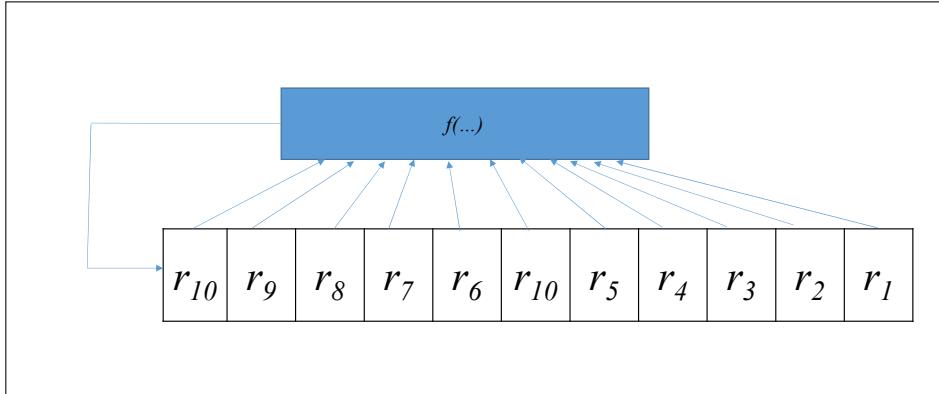


Figure 2.14: Feedback Shift Register, with (linear or non-linear) feedback function $f()$.

1. $f(r_1, \dots, r_l) = \left(\sum_{i=1}^l r_i \right) \bmod 2$, $g(r_1, \dots, r_l) = r_1$. Note: this is an LFSR.
2. $f(r_1, \dots, r_l) = \prod_{i=1}^l r_i$; and any g .

There are also many other designs of PRGs based on Feedback Shift Registers (FSRs), often combining multiple FSRs (often LFSRs) in different ways; one reason is that FSRs are convenient for efficient hardware implementations. For example, the A5/1 and A5/2 PRGs, defined in the GSM standard, combine three LFSRs. Other PRG designs exist, e.g., the *RC4* design, which is designed for convenient software implementation.

There are many cryptanalysis attacks on different PRGs, including the three mentioned above (RC4 and the two GSM ciphers, A5/1 and A5/2). Presenting details of these ‘classical’ ciphers and attacks on them is beyond our scope, see, e.g., [20] for GSM’s A5/1 and A5/2. However, let us briefly discuss the vulnerabilities found in RC4, and their potential impact.

2.5.6 RC4: Vulnerabilities and Attacks

The RC4 PRG design features simplicity and good efficiency (including for software implementation). This design is publicly available since its anonymous, unofficial disclosure in September 1994 [324]. It was therefore adopted by several standards, including the WEP and WPA wireless-LAN standards (Section 2.10) and the SSL/TLS protocol (Chapter 7), which further increased the cryptanalytical efforts to identify exploitable weaknesses. Several works have, in fact, found vulnerabilities in RC4.

Describing the details of RC4’s design and the cause of the vulnerabilities is beyond our scope. However, let us briefly discuss the impact of the first major reported vulnerability, the Mantin-Shamir attack on RC4 [224], as we find this vulnerability and its history to be instructive in several ways. We also

discuss another important vulnerability due to *incorrect usage* of RC4, which is not due to a vulnerability of RC4 at all - the same problem would hold when incorrectly using *any* PRG; such incorrect-usage vulnerabilities are even more common than vulnerabilities due to cryptanalytical attacks, which is one reason that it is so important to understand the exact definition and security goals of cryptographic mechanisms.

The Mantin-Shamir and other RC4 vulnerabilities. Since the output of a PRG should be indistinguishable from random, then any detectable difference between the output of the PRG and the uniformly random distribution, is a failure of the PRG. For the theoretical definition, this holds for any computable difference, regardless of the ability to exploit it for a specific practical attack; for example, it suffices that some efficiently computable function f has a different distribution when applied to specific output bits of RC4, compared to its distribution for random bits. However, many PRG weaknesses, and in particular, RC4 weaknesses, are simpler: a *bias* of a particular bit or byte of the RC4 output.

In particular, for a random byte sequence, the probability of any byte in the sequence to have a particular value should be exactly 2^{-8} , since all 2^8 byte values should be equally likely. However, in 2001, Mantin and Shamir found that given a random seed/key, the second output byte of RC4, denoted Z_2 , has *observable bias* from random. Specifically, they found that:

$$Pr(Z_2 = 0^8) \approx 2^{-7} \quad (2.26)$$

Clearly, this shows that RC4 does not fulfill our definition of a secure PRG: its output *can* be efficiently distinguished from a random bit sequence. Based on the conservative design principle (Principle 3), this should have caused the use of alternative mechanisms, at least significantly enhancing the RC4 security mechanism, or a completely different design. However, that did not happen; in spite of this and additional vulnerabilities found, the use of RC4 continued and even increased over more than a decade, with adoption by new standards such as TLS and WPA, in addition to its use by older standards such as SSL and WEP.

Apparently, this vulnerability appeared too minor to be a major concern and to stop using RC4. Which shows the difficulty of adopting the conservative design principle in practice. Really, can we justify the principle? Can such apparently-minor vulnerability be exploited in a realistic attack? The answer to both questions is a resounding *yes!*

First, when we find one vulnerability, we should assume more are probably lurking, possibly already known and exploited by some powerful attacker; we better change to a more secure design. In the case of RC4, this proved to be the case. Multiple additional vulnerabilities were discovered over the years, beginning with another important vulnerability in the same year [123]; see some of them in [194]. Even these did not stop the use of RC4, until the effective attack on TLS of [9] (see subsection 7.2.5). It is widely believed that some attackers

already exploited attacks against RC4 for years before it was finally abandoned. Some products still use RC4, at least for ‘backward compatibility’, which is often vulnerable to *downgrade attacks*, see subsection 5.6.3 and Section 7.5.

Second, the Mantin-Shamir attack *can be abused* in some applications and scenarios. Specifically, in some applications, the same secret, x , is encrypted using RC4 many times, using different seed values, s_1, s_2, \dots . This scenario can occur in practice, e.g., when the SSL/TLS protocol is used to secure communication between browser and website; see subsection 7.2.5. Let us show how an attacker can, in such cases, find the second byte $x[2]$ of the secret x , if x is encrypted using RC4, with seeds s_1, s_2, \dots .

Let $Z_2(s_i)$ denote the value of the second byte of the output of *RC4*, when initiated with seeds s_1, s_2, \dots . Let us focus on the (vulnerable) second byte. The encryption of $x[2]$ using seed s_i is the ciphertext $c_i[2] = x[2] \oplus Z_2(s_i)$. Hence, while the probability of most bytes of the ciphertext is about 2^{-8} , from Equation 2.26 we have $\Pr(c_i[2] = x[2]) \approx 2^{-7}$.

Therefore, the second byte of the secret x would simply be the most common second byte of the ciphertext - by a large margin. Namely, the second byte of the plaintext encrypted by RC4, can be exposed by a simple, efficient and effective Ciphertext Only (CTO) attack, if the attacker can obtain a reasonable number of ciphertexts (few hundreds at most). And if this is not sufficiently convincing, see the improved, practical attack on TLS in subsection 7.2.5.

Several variants of RC4 were proposed to defend against this and other attacks. The simplest variant simply discards some initial portion of the output; this is known as *RC4-dropN*, where N is the number of initial output bytes dropped discarded, e.g., 256 or 1024. RC4-dropN clearly avoids the specific Mantin-Shamir attack, but may fail against other attacks, most notable the attack against the use of RC4 by TLS [9], see subsection 7.2.5.

Vulnerabilities due to incorrect use of RC4 (or any PRG). While PRGs, and other cryptographic schemes, can be vulnerable, many security failures are due to *incorrect, vulnerable usage* of cryptographic schemes. Let us give an example of an attack against vulnerable *deployment* of a PRG in a system. Specifically, we present an attack against MS-Word 2002, which exploits a vulnerability in the *usage* of the RC4 PRG, rather than in its design. Namely, this attack could have been carried out if *any* PRG was used in the same (incorrect and vulnerable) way, as RC4 was used by MS-Word 2002.

Example 2.5. *MS-Word 2002 used RC4 for document encryption, in the following way. The user provided password for the document; that password was used as a key to the RC4 PRG, producing a long pseudo-random string which is referred to as Pad, i.e., Pad = RC4(password). When the document is saved or restored from storage, it is XORed with Pad. This design is vulnerable; can you spot why?*

The vulnerability is not specific in any way to the choice of RC4; the problem is in how it is used. Namely, this design re-uses the same pad whenever the document is modified - a ‘multi-times pad’ rather than OTP. For example,

suppose that the document is changed by adding one letter x , say in position i . Clearly, it is possible to find i , given only the two ciphertexts.

Furthermore, let $c[i]$ be the original ciphertext in position i , and $c'[i]$ be the ciphertext in position i after the insertion of x ; and let y be the i^{th} plaintext character before the insertion, which then moves to be the $(i+1)^{\text{th}}$ character. Then $c[i] = y \oplus \text{pad}[i]$ and $c'[i] = x \oplus \text{pad}[i]$. By XORing the two equations, we have: $c[i] \oplus c'[i] = (y \oplus \text{pad}[i]) \oplus (x \oplus \text{pad}[i]) = y \oplus x$. Hence, knowing x gives y and vice versa; and if neither is known, the attacker can still learn $y \oplus x$, which still gives some information on the plaintext.

If these exposures do not yet convince the reader of the insecurity of this design, then see details of a complete, practical plaintext-recovery attack in [226].

□

The fact that the vulnerability is due to the use of RC4 and not to cryptanalysis of RC4, is very typical of vulnerabilities in systems involving cryptography. In fact, cryptanalysis is rarely the cause of vulnerabilities - system, configuration and software vulnerabilities are more common.

2.5.7 Random functions

One practical drawback of stream ciphers is the fact that they require state, to remember how many bits (or bytes) were already output. What happens if state is lost? Can we eliminate or reduce the use of state? It would be great to allow recovery from loss of state, or to avoid the need to preserve state when encryption is not used, e.g., between one message and the next. In the next section, we introduce another pseudo-random cryptographic mechanism, called a *pseudorandom function (PRF)*, which has many applications in cryptography - including *stateless, randomized shared-key cryptosystems*. However, before we introduce *pseudo*-random functions, let us first discuss the ‘real’ *random functions*.

Process for selecting a random function. Let $F_{D,R}$ denote the set of all functions from a finite domain D to a finite range R (i.e., $F_{D,R} \equiv \{D \rightarrow R\}$). A function $f \in F_{D,R}$ maps each element in D , to an element in R ; hence, the $F_{D,R}$ is a finite set. More specifically, we can map each element in D , to any element in R . The total number of functions is, therefore, $|R|^{|D|}$, and each specific function is selected with probability $|R|^{-|D|}$.

But how can we *select* a random function? One way is as follows: for each input $x \in D$, select a random element in R to be $f(x)$, namely:

$$(\forall x \in D)f(x) \xleftarrow{\$} R \quad (2.27)$$

In a typical case, both D and R are binary strings, i.e., for some integers n, m , we have $R = \{0, 1\}^n$, $D = \{0, 1\}^m$. In this case, there are 2^m elements in the domain $D = \{0, 1\}^m$, i.e., $|D| = 2^m$, and each random selection of an element in the range $R = \{0, 1\}^n$ required n random bits; in total, to select a random function we need $n \cdot 2^m$ coin flips.

This process can be manually done for small domain and range, by randomly choosing the mapping and writing it in a table, e.g., as in Table 2.2 and Exercise 2.10.

Function	Domain	Range	00	01	10	11	coin-flips
f_1	$\{0, 1\}^2$	$\{0, 1\}$					
f_2	$\{0, 1\}^2$	$\{0, 1\}^3$					

Table 2.2: Do-it-yourself table for selecting f_1, f_2 randomly, in Exercise 2.10.

Exercise 2.10. *Using a coin, select randomly the functions below; count your coin flips.*

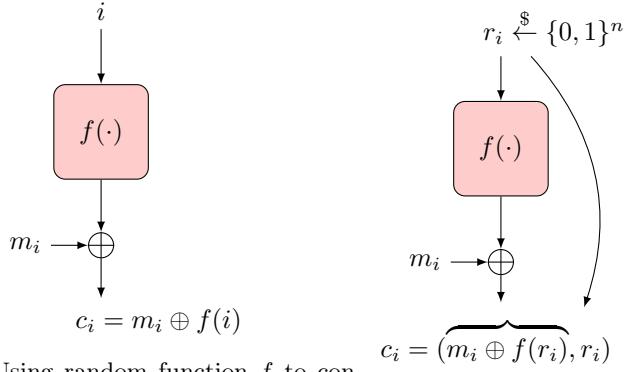
1. $f_1 : \{0, 1\}^2 \rightarrow \{0, 1\}$ (use a copy of Table 2.2)
2. $f_2 : \{0, 1\}^2 \rightarrow \{0, 1\}^3$ (use a copy of Table 2.2)
3. $f_3 : \{0, 1\}^3 \rightarrow \{0, 1\}^2$ (create your own table)

How many coin flips required were required for each function? For each of the functions, what is the probability that all its output bits are zero? And of all outputs bits being 1?

In the typical case where the domain is large, the choice of a random function requires excessive number of random bits (coin-flip operations). Selecting and storing such a function is difficult, as would be *sending* it - which is required if we want multiple parties to use the same random function, which is actually very useful for cryptographic applications - such as a stateless stream cipher. This motivates the use of *pseudorandom functions*, which we discuss in the next subsection. However, let us first discuss random functions a bit more, to improve our understanding of this important but subtle concept.

Stream cipher using a random function. Figure 2.15a presents the design of a stream cipher using a randomly-chosen function f which is shared by the two parties and kept secret. The design could be used either for bit-by-bit encryption, with the random function mapping each input i to a single bit $f(i)$, which is then XORed with the corresponding message bit m_i to form the ciphertext $c_i = m_i \oplus f(i)$. Alternatively, both input messages m_i and the output of the random function could be strings of some length, e.g., n , and then each invocation of the random function will produce n pad bits, XORed with n pad bits to produce n cipher bits.

One drawback of the use of stream ciphers is the need to maintain *synchronized state* between sender and recipient. This refers to the (typical) case where the input is broken into multiple messages, each provided in a separate call to the encryption device. To encrypt all of these messages using a stream cipher - OTP or the design in Figure 2.15a - the two parties must maintain the number of calls i (bits or strings of fixed length). To avoid this requirement, we can use *randomized encryption*, as we next explain.



- (a) Using random function f to construct a stream-cipher for stateful encryption, with limited storage (a struct stateless, randomized encryption-counter). Does not require randomization. This construction has high computation and communication-optimal munication overhead: n (random) bits ($|ciphertext| = |plaintext|$). (b) Using random function f to construct a randomized encryption. This construction has high computation and communication-optimal munication overhead: n (random) bits per plaintext bit.

Figure 2.15: Bit-wise encryption using random function $f(\cdot)$. We later improve: replace random function by *pseudorandom function (PRF)*, and use block-wise operations to reduce overhead; e.g., *counter-mode* (stateful) and *OFB-mode* (randomized).

Stateless, randomized encryption using a random function f . An even more interesting application of a random function, is to avoid the need for the two parties to maintain state (of the message/bit counter i). To do this, we use the random function to construct *randomized encryption*, as shown in Figure 2.15b.

To encrypt each plaintext message m_i , we choose a string r_i of n random bits, i.e., $r_i \xleftarrow{\$} \{0, 1\}^n$. The ciphertext is the pair $E_f(m_i) \equiv (m_i \oplus f(r_i), r_i)$. Note our use of the function f as the key to the encryption; indeed, we can think of the table containing our random mapping of each of the 2^n strings in the domain $\{0, 1\}^n$ to a bit, as the function - and as the key to the encryption process.

Security. As with every cryptographic mechanism, we ask: are the designs in Figure 2.15b and Figure 2.15a secure? Intuitively, the design of Figure 2.15a is secure as long as we never re-use the same counter value. Similarly, the design of Figure 2.15b is secure as long as we use a sufficient number of random bits. Both statements are correct; but it isn't trivial to understand why. Let us focus on the slightly more complex case of randomized encryption (the design of Figure 2.15b); the argument for the counter-based, stateful stream cipher design (Figure 2.15a) follows similarly.

An obvious concern is that an attacker may try to predict the value of $f(r_i)$ used to encrypt a message (or bit) m_i , from previously-observed ciphertexts

$\{c_j\}_{j < i}$. Let us assume, to be safe, that the attacker knows *all* the corresponding plaintexts m_j , allowing the attacker to find all the corresponding mappings $\{f(r_j)\}_{j < i}$. Using this information, can the attacker guess $f(r_i)$?

It is possible that r_i is the same as one of the previously-used random values, i.e., $r_i = r_j$ for some $j < i$. In this case, the attacker has received already c_j ; and since we assumed that the attacker knows m_j , it follows that the attacker can expose m_i , by computing:

$$m_i = c_i \oplus f(r_i) = c_i \oplus f(r_j) = c_i \oplus (m_j \oplus c_j) \quad (2.28)$$

To address this, we clearly must use sufficiently-long random strings. But what if $r_i \notin \{(r_j)\}_{j < i}$?

To answer this, reconsider the process of selecting a random function, as you did in Exercise 2.10. What we did was to select the entire table - mapping from every element in the domain to a random element in the range - *before* we applied the random function. However, notice that it does not matter if, instead, we choose the mapping for each element $r_i \in D$ in the domain *only on the first time we need to compute $f(r_i)$* . Think it over!

This means, that if $r_i \notin \{(r_j)\}_{j < i}$, then the attacker *does not learn anything about $f(r_i)$* , even if it is given all of the $\{f(r_j)\}_{j < i}$ values. Until we select (randomly) the value of $f(r_i)$, the attacker cannot know anything about it. Therefore, the only concern we have is with the case that $r_i \in \{(r_j)\}_{j < i}$. Let us return to this issue; what is the probability of that happening? Well since each mapping is selected randomly, simply $\frac{i-1}{|D|}$. Focusing on the typical case where the input domain is $\{0, 1\}^n$, this is $\frac{i-1}{2^n}$.

Therefore, if n is ‘sufficiently large’, then the maximal number of observations by the attacker would still be negligible compared to 2^n - and $\frac{i}{2^n}$ would be negligible. For example, if the attacker can observe a million encryptions, we ‘just’ need 2^n to be way larger than one million ; and considering that a million is less than 2^{20} , using n significantly larger than 20 - definitely for $n = 120$ or even less - seems safe enough.

Efficiency. So the scheme is secure - provided n is ‘sufficiently large’, e.g., 80 or more. However, is it also *efficient*? To implement the scheme, we need to compute the random function f ; since we want a recipient to decipher our messages, we need to compute and send all of f before we begin sending ciphertexts. However, this requires us to flip and (securely) share 2^n bits - for $n = 80$ (or more). Unfortunately, that’s clearly impossible. Fortunately, on the other hand, we can use a *pseudorandom function (PRF)* instead of the random function, providing an efficient solution which is still secure against computationally-limited adversaries.

Note that there is another efficiency concern with the scheme: is it really necessary to send a new random string for each bit? Of course not. We can address this concern in two ways:

Large range $R = \{0, 1\}^l$: this allows us to use the same random string r or counter i , to encrypt a *block* of l plaintext bits, by bitwise XOR of the l

message bits with the corresponding l -bit output of $f(r)$ (or $f(i)$). In this way, the n bits of r allow encryption of l bits of plaintext. See Figure 2.16.

Use $f(r)$ as seed of a PRG: if we use a sufficiently large range, a PRG could ‘expand’ $f(r)$ into as many bits as required to bit-wise XOR with the plaintext: $E_f(m) = (r, PRG(f(r)) \oplus m)$. In this way, the n bits of r allow encryption of *arbitrarily long* plaintext m - requiring new n random bits only to encrypt new plaintext, and only if the state (of the PRG) was not retained. This is essentially what is done by the *Output Feedback (OFB) mode of operation*, which we see later on, except that the OFB mode also *implements the PRG* using the PRF, instead of using two separate functions (a PRF and a PRG). Figure 2.17 shows this design, using a PRF F_k .

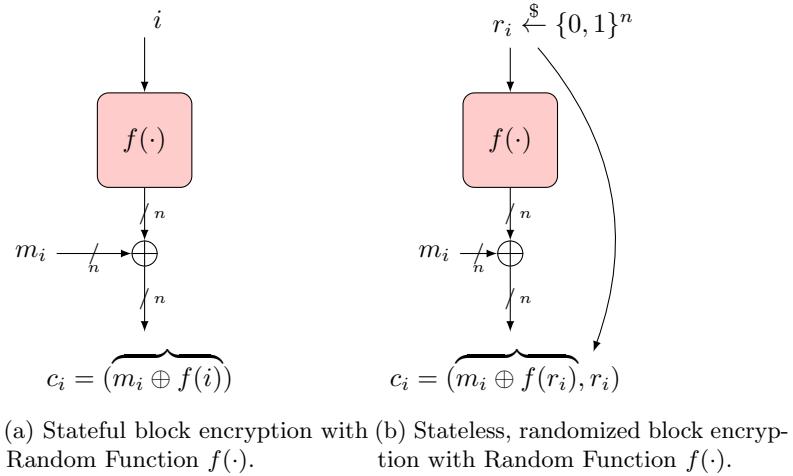


Figure 2.16: Block (n -bits) encryption using a Random Function $f(\cdot)$. Use only one function application for n plaintext bits.

2.5.8 pseudorandom functions (PRFs)

A *pseudorandom function (PRF)* is an efficient substitute to the use of a random function, which ensures similar properties, while requiring the generation and sharing of only a short key. The main limitation is that PRFs are secure only against computationally bounded adversaries.

A PRF scheme has two inputs: a *secret key* k and a ‘message’ m ; we denote it as $PRF_k(m)$. Once k is fixed, the PRF becomes only a function of the message. The basic property of PRF is that this function ($PRF_k(\cdot)$) is *indistinguishable* from a truly random function. Intuitively, this means that a PPT adversary cannot tell if it is interacting with $PRF_k(\cdot)$ with domain D and range R , or with a random function f from D to R . Hence, PRFs can be used

in many applications, providing an efficient, easily-deployable alternative to the impractical truly random functions.

For example, PRFs can be used to construct shared-key cryptosystems, as illustrated in Figure 2.17. The figure presents two designs of a cryptosystem from a PRF: a stateful encryption, as a stream cipher, in Figure 2.17a, and a stateless randomized encryption, in Figure 2.17b.

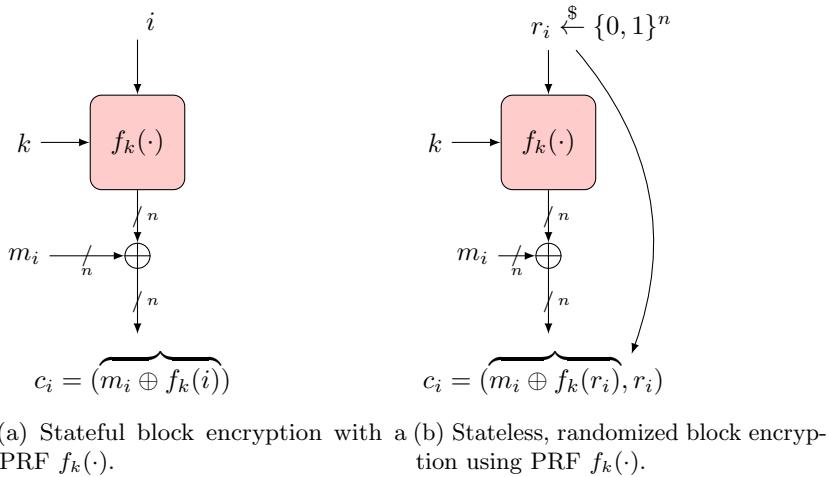


Figure 2.17: Block (n -bits) encryption using a pseudorandom function (PRF) $f_k(\cdot)$. Use only one PRF application for n plaintext bits.

Both designs simply use a PRF instead of a random function, used in the corresponding designs in Figure 2.15. The security of the PRF designs follows from the security of the corresponding random-function-based designs - and from the indistinguishability of a PRF and a random function. Indeed, this is one case of a very useful technique, which we refer to as the *random function design principle*.

Principle 6 (Random function design). *Design cryptographic protocols and mechanisms using a random function, to make the security analysis easier. Once secure, implement using a pseudorandom function; security would follow since a PRF is indistinguishable from a random function.*

We now need to finally properly define a secure *pseudorandom function (PRF)*; this definition reuses the oracle notation and other concepts introduced in Section A.1. In this definition, the adversary \mathcal{A} has *oracle access* to one of two functions: a random function from domain D to range R , i.e., $f \xleftarrow{\$} \{D \rightarrow R\}$, or the PRF keyed with a random n -bit key, F_k , with k being a random n -bit string, i.e., $k \xleftarrow{\$} \{0, 1\}^n$. We denote these two cases by \mathcal{A}^f and \mathcal{A}^{F_k} , respectively. The adversary should try to distinguish between these two cases, e.g., by outputting 0 (or ‘false’) if given oracle access to the random function f , and outputting 1

(or ‘true’) if given access to the PRF F_k . The idea of the definition is illustrated in Fig. 2.18.

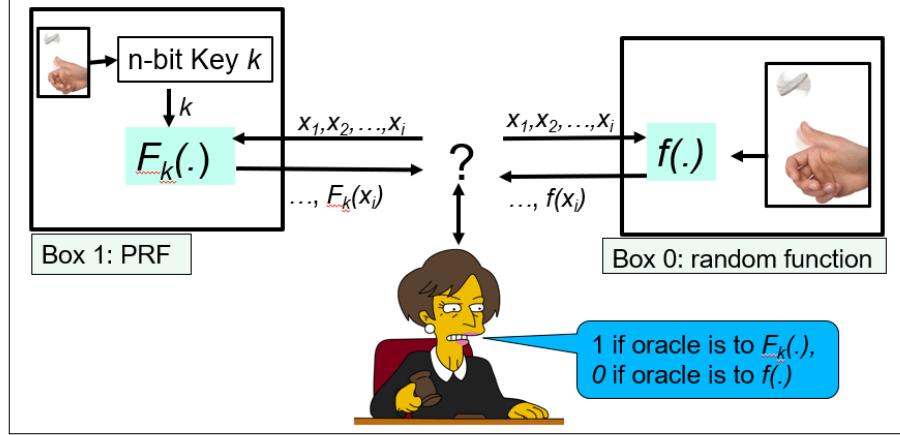


Figure 2.18: The pseudorandom function (PRF) Indistinguishability Test. We say that function $F_k(x) : \{0, 1\}^* \times D \rightarrow R$ is a (secure) pseudo-random generator (PRG), if no distinguisher D can efficiently distinguish between $F_k(\cdot)$ and a random function f from the same domain D to the same range R , when the key k is a randomly-chosen sufficiently-long binary string.

We now finally define a *pseudorandom function (PRF)*, $F_k(x) : \{0, 1\}^* \times D \rightarrow R$. The domain¹⁰ consists of the key, which we assume to be an (arbitrary long) binary string, i.e., from the set $\{0, 1\}^*$; and of an input from an arbitrary set D . The scheme must allow for arbitrary length for the key, since security requirements - in this case, indistinguishability - are defined asymptotically, i.e., for sufficiently long keys; see Chapter 1.

Definition 2.7. A pseudorandom function (PRF) is a polynomial-time computable function $F_k(x) : \{0, 1\}^* \times D \rightarrow R$ s.t. for all PPT algorithms \mathcal{A} , $\varepsilon_{\mathcal{A}, F}^{PRF}(n) \in NEGL$, i.e., is negligible, where the advantage $\varepsilon_{\mathcal{A}, F}^{PRF}(n)$ of the PRF F against adversary \mathcal{A} is defined as:

$$\varepsilon_{\mathcal{A}, F}^{PRF}(n) \equiv \Pr_{k \xleftarrow{\$} \{0, 1\}^n} [\mathcal{A}^{F_k}(1^n)] - \Pr_{f \xleftarrow{\$} \{D \rightarrow R\}} [\mathcal{A}^f(1^n)] \quad (2.29)$$

The probabilities are taken over random coin tosses of \mathcal{A} , and random choices of the key $k \xleftarrow{\$} \{0, 1\}^n$ and of the function $f \xleftarrow{\$} \{D \rightarrow R\}$.

Overview of the PRF indistinguishability test. The basic idea of this definition is the use of *indistinguishability test*, much like in the definition of a secure PRG (Definition 2.6), and even the Turing indistinguishability test

¹⁰The notation $\{0, 1\}^* \times D$ simply means a pair: a key from $\{0, 1\}^*$ and an element from set D .

(Figure 2.12). Namely, a PRF (F_k) is secure if every *PPT* algorithm \mathcal{A} cannot have significant advantage in identifying the pseudorandom function. We define the advantage as the probability that \mathcal{A} outputs 1 ('true', i.e., pseudorandom) when given oracle access to the pseudorandom function F_k , minus the probability that \mathcal{A} outputs 1 ('true', i.e., pseudorandom) when given oracle access to the random function F_k , where both functions are over the same domain D and range R . 'Significant' here means at least some positive polynomial in n , the length of the key k , often referred to as the *security parameter*.

The oracle notation. Both \mathcal{A}^{F_k} and \mathcal{A}^f use the *oracle* notation introduced in Definition 1.3. Namely, they mean that \mathcal{A} is given 'oracle' access to the respective function ($F_k()$ and $f()$). Oracle access means that the adversary can give any input x and get back that function applied to x , i.e., $F_k(x)$ or $f(x)$, respectively

Why allow arbitrary key length (security parameter)? The definition allows arbitrarily-long keys, although in practice, cryptographic standards often have a fixed key length, or only a few options. The reason that the definition allows arbitrary length is that it requires the success probability to be negligible - smaller than any polynomial in the key length - which is meaningless if the key length is bounded.

Why is 1^n given as input to the adversary? A subtle, yet important, aspect of the definition is the fact that in the two calls to the adversary \mathcal{A} , we provide the adversary with the value 1^n as input, where n is the key length (security parameter). The value 1^n simply signifies a string of n consecutive bits whose value is 1, i.e., it is the value of n encoded in *unary*. But why provide 1^n as input? It makes sense that the adversary should be informed of the key length n , but why use unary encoding? Why not provide n using the 'standard' binary encoding?

To understand the reason, first recall that we focus on *efficient* (*PPT*) algorithms; namely, the running time of both the pseudorandom function F and the adversary \mathcal{A} is bounded by a polynomial in the *size of their inputs*. The inputs to the PRF include the key, and hence, consists of at least n bits; hence, the running time of the PRF is (at least) polynomial in n . It is therefore 'only fair' that the running time of the adversary \mathcal{A} is also allowed to be polynomial in n ; to ensure this, we provide to it 1^n as input, similarly to our provision of the security parameter 1^l to other algorithms, see Section A.1.

Examples of secure and insecure PRFs. To clarify Definition 2.7, and to demonstrate how to show if a given function is a PRF or not, we give and solve two exercises. The first exercise shows two insecure PRF designs; the following exercise proves that a given construction is a secure PRF.

Exercise 2.11. *Examples of an insecure PRF constructions:*

1. Show that $F_k(m) = k \oplus m$ is not a secure PRF.
2. Let p be an n -bits prime number, and $F_k(m) \equiv k \cdot (m + k) \pmod{p}$. Show that F is not a secure PRF (to the domain $\{0, \dots, p - 1\}$).
3. Show that $F_k(m) \equiv k \vee (m^k + k^{m+k} \pmod{2^n})$ is not a secure PRF.
4. Assume that secure PRF functions exist. Given a function $F_k(m)$, define $\hat{F}_k(m) \equiv F_m(k)$, i.e., \hat{F} switches between the key and the input of F . Show that even if we know that F is secure, it is possible that \hat{F} is not a secure PRF.

Solutions:

1. *Question:* show that $F_k(m) = k \oplus m$ is not a secure PRF.
Solution: The adversary \mathcal{A}^g is given an oracle to a function g , and needs to output ‘True’ if $g(\cdot) = F_k(\cdot)$ for a random key k and ‘False’ if $g(\cdot)$ is random function. A simple way to do this is for A^g first to make a query for $g(0)$; if $g(\cdot) = F_k(\cdot)$ then \mathcal{A} receives back $g(0^n) = F_k(0^n) = k \oplus 0^n = k$. So in this case ($g(\cdot) = F_k(\cdot)$) A ‘knows’ k ; it can check if indeed $g(\cdot) = F_k(\cdot)$ (or g is a random function) by giving any other input $m \neq 0^n$. If $g(m) = k \oplus m$, then (with very high probability) the function is indeed $g(\cdot) = F_k(\cdot)$, i.e., not a random function, and \mathcal{A} returns ‘True’; if $g(m) \neq k \oplus m$ then the function g is definitely *not* $g(\cdot) = F_k(\cdot)$, which means, in this case, it must be a random function, and \mathcal{A} returns ‘False’.
2. *Question:* let p be an n -bits prime number, and $F_k(m) \equiv k \cdot (m + k) \pmod{p}$. Show that F is not a secure PRF.
Solution: Similarly to the previous item, the adversary first gives to the oracle the input 0. If the oracle is to $F_k(m) \equiv k \cdot (m + k) \pmod{p}$, then the adversary receives $k^2 \pmod{p}$. Now, every number whose value is $k^2 \pmod{p}$ for some integer k is called a *quadratic residue*. We discuss quadratic residues in subsection 6.1.8, in particular, explaining that there is an efficient algorithm to determine if a number is quadratic residue or not (Claim 6.1). The adversary may use this test, and if the oracle returns a quadratic residue, the adversary assumes that the oracle is to $F_k(m) \equiv k \cdot (m + k) \pmod{p}$ and returns ‘True’, since, with high probability, a random function would not return a quadratic residue.

Let us see another solution, that does not require testing the output for being a quadratic residue. Observe that $m + k$ is even, if and only if either both m and k are even, or both m and k are odd. This motivates the following distinguishing adversary $\mathcal{A}^g(m)$, that outputs ‘True’, with high probability, if $g(m) = F_k(m)$, and with low probability, if g is a random function. The adversary ask the oracle to compute *both* $g(0)$ and $g(2)$. If the two results, $g(0)$ and $g(2)$, have the same parity (either are both even, or are both odd), then the adversary concludes that g is more likely to be pseudo-random, and outputs ‘True’. Otherwise, if one of $\{g(0), g(2)\}$ is

even and the other is odd, then g cannot be $F_k(m) \equiv k \cdot (m + k) \pmod{p}$, and hence must be a random function, and \mathcal{A} outputs ‘False’.

Notice that when this adversary outputs ‘True’, it would be wrong in the cases where the oracle function g was selected at random, but happens to return an even number for both inputs (zero and 2). However, since g was selected at random, the values $g(0)$ and $g(2)$ were also selected at random, and the probability of both $g(0)$ and $g(2)$ being even would be only $\frac{1}{4}$. Hence, while the adversary could be wrong, it still has a significant (non-negligible) advantage.

3. *Question:* show that $F_k(m) \equiv k \vee (m^k + k^{m+k} \pmod{2^n})$ is not a secure PRF.

Solution: it seems quite hard to predict much about a specific value of $F_k(m)$ for a random key k . However, since F includes an ‘or’ operation with k , any bit which is set in k , is always output in $F_k(m)$ (for every m). Since almost always k has *some* non-zero bits, then the adversary can detect these bits by computing the ‘and’ operation to multiple outputs of the oracle (for different inputs). Notice that if the oracle is to a random function, then the probability of any given bit to be set in the outputs of l different inputs is only 2^{-l} . This allows the adversary to efficiently distinguish between being given an oracle to $F_k(m) \equiv k \vee (m^k + k^{m+k} \pmod{2^n})$ - vs. being given an oracle to a random function.

4. *Question:* assume that secure PRF functions exist. Given a function $F_k(m)$, define $\hat{F}_k(m) \equiv F_m(k)$, i.e., \hat{F} switches between the key and the input of F . Show that even if we know that F is secure, it is possible that \hat{F} is not a secure PRF.

Solution: Let F' be a secure PRF; define:

$$F_k(m) \equiv \{F'_k(m) \text{ if } k \neq 0^n, \text{ otherwise } (k = 0^n): 0^n\}. \quad (2.30)$$

As long as the key chosen is not the special all-zeros key, F is the same as F' , hence, F is also a PRF, since keys are selected randomly so $k = 0^n$ is selected only with probability 2^{-n} . It remains to show that with this F function, \hat{F} is not a secure PRF.

However, an adversary given an oracle to either \hat{F} or a random oracle, can simply give the input 0^n . If the oracle is to \hat{F} , then this returns $\hat{F}_k(0^n) = F_{0^n}(k)$. However, by definition (Equation 2.30), $\hat{F}_k(0^n) = F_{0^n}(k) = 0^n$. Hence, an adversary can easily distinguish between $\hat{F}_k(m)$ and a random function. \square

Exercise 2.12 (Example of a secure PRF construction). *Assume that $F_k(m) : \{0,1\}^n \times \{0,1\}^n \rightarrow \{0,1\}^n$ be a secure PRF. Prove that $\hat{F}_k(m) = F_k(m \oplus 1)$ is also a secure PRF.*

Solution: Assume, to the contrary, that \hat{F} is not a PRF. We show it follows that F is also not a PRF - a contradiction; this shows that \hat{F} must be a PRF.

From Definition 2.7, if \hat{F} is not a PRF, then there is a PPT algorithm $\hat{\mathcal{A}}^{\hat{g}}$, with an oracle denoted \hat{g} , such that:

$$\varepsilon_{\hat{\mathcal{A}}, \hat{F}}^{PRF}(n) \equiv \Pr_{k \xleftarrow{\$} \{0,1\}^n} \left[\hat{\mathcal{A}}^{\hat{F}_k(\cdot)}(1^n) \right] - \Pr_{\hat{f} \xleftarrow{\$} \{\{0,1\}^n \rightarrow \{0,1\}^n\}} \left[\hat{\mathcal{A}}^{\hat{f}(\cdot)}(1^n) \right] \notin NEGL(n)$$

In the left-hand expression the oracle $\hat{g}(\cdot)$ is $\hat{F}_k(\cdot)$, for random key k , while in the right-hand expression the oracle $\hat{g}(\cdot)$ is a random function denoted $\hat{f}(\cdot)$.

Given $\hat{\mathcal{A}}^{\hat{g}}$, we define adversary \mathcal{A}^g as follows:

$$\mathcal{A}^{g(\cdot)}(1^n) \equiv \left\{ \text{Let } \hat{g}(m) \equiv g(m \oplus 1); \text{ Return } \hat{\mathcal{A}}^{\hat{g}(\cdot)}(1^n) \right\} \quad (2.31)$$

The notation $\hat{\mathcal{A}}^{\hat{g}(\cdot)}(1^n)$ refers to the result of the $\hat{\mathcal{A}}$ adversary, when run with input (security parameter) 1^n , and with an oracle to the \hat{g} function, where $\hat{g}(m) \equiv g(m \oplus 1)$. Namely, whenever $\hat{\mathcal{A}}$ provides input m to the \hat{g} oracle, then the adversary \mathcal{A}^g computes $g(m \oplus 1)$ and returns it to $\hat{\mathcal{A}}$, as the response from $\hat{g}(\cdot)$.

To complete the proof, we show that $\varepsilon_{\hat{\mathcal{A}}, \hat{F}}^{PRF}(n) = \varepsilon_{\mathcal{A}, F}^{PRF}(n)$, or more specifically that the following two equations hold:

$$\Pr_{k \xleftarrow{\$} \{0,1\}^n} \left[\hat{\mathcal{A}}^{\hat{F}_k}(1^n) \right] = \Pr_{k \xleftarrow{\$} \{0,1\}^n} \left[\mathcal{A}^{\text{FALSE}_k}(1^n) \right] \quad (2.32)$$

$$\Pr_{\hat{f} \xleftarrow{\$} \{\{0,1\}^n \rightarrow \{0,1\}^n\}} \left[\hat{\mathcal{A}}^{\hat{f}}(1^n) \right] = \Pr_{\hat{f} \xleftarrow{\$} \{\{0,1\}^n \rightarrow \{0,1\}^n\}} \left[\mathcal{A}^f(1^n) \right] \quad (2.33)$$

Equation 2.32 follows immediately from substituting Equation 2.31 and then $\hat{F}_k(m) = F_k(m \oplus 1)$:

$$\Pr_{k \xleftarrow{\$} \{0,1\}^n} \left[\mathcal{A}^{\text{FALSE}_k(\cdot)}(1^n) \right] = \Pr_{k \xleftarrow{\$} \{0,1\}^n} \left[\hat{\mathcal{A}}^{\text{FALSE}_k(\cdot \oplus 1)}(1^n) \right] \quad (2.34)$$

$$= \Pr_{k \xleftarrow{\$} \{0,1\}^n} \left[\hat{\mathcal{A}}^{\hat{F}_k}(1^n) \right] \quad (2.35)$$

Let us similarly prove Equation 2.33. First substitute Equation 2.31:

$$\Pr_{f \xleftarrow{\$} \{\{0,1\}^n \rightarrow \{0,1\}^n\}} \left[\mathcal{A}^{f(\cdot)}(1^n) \right] = \Pr_{f \xleftarrow{\$} \{\{0,1\}^n \rightarrow \{0,1\}^n\}} \left[\hat{\mathcal{A}}^{f(\cdot \oplus 1)}(1^n) \right] \quad (2.36)$$

$$= \Pr_{f \xleftarrow{\$} \{\{0,1\}^n \rightarrow \{0,1\}^n\}} \left[\hat{\mathcal{A}}^{\hat{f}}(1^n) \right] \quad (2.37)$$

$$\left[\hat{\mathcal{A}}^{\hat{f}(\cdot)}(1^n) | \hat{f}(\cdot) \equiv f(\cdot \oplus 1) \right] = \Pr_{\hat{f} \xleftarrow{\$} \{\{0,1\}^n \rightarrow \{0,1\}^n\}} \left[\hat{\mathcal{A}}^{\hat{f}(\cdot)}(1^n) \right] \quad (2.38)$$

The last step is justified since we choose $f \xleftarrow{\$} \{\{0,1\}^n \rightarrow \{0,1\}^n\}$ with uniform distribution; i.e., all functions are equally likely. Hence, the probability does not change when we directly choose \hat{f} , instead of choosing f and defining $\hat{f}(\cdot) \equiv f(\cdot \oplus 1)$. \square

Additional PRF applications. PRFs have many additional applications:

Message Authentication. In Chapter 4, we show how a PRF may be used for message authentication.

Derive independently-random keys/values. In many scenarios, two parties share only one key k , but need to use multiple shared keys which are ‘independently-random’. This is easily achieved using PRF f ; if g_1, g_2, \dots are distinct identifiers, one of each required value, then we can derive the keys as $k_1 = f_k(g_1), k_2 = f_k(g_2)$, and so on. As a concrete example, to derive separate keys for each day d from the same k , we can use $k_d = f_k(d)$; exposure of k_2 and k_4 will not expose any other key, e.g., k_3 !

Pseudo-random permutation or block cipher. We discuss the use of PRF to construct a pseudo-random permutation in the following subsection; later, in Section 2.6, we show how to extend this to construct a block cipher.

2.5.9 PRF: Constructions and Robust Combiners

The concept of PRFs was proposed in a seminal paper by Goldreich, Goldwasser and Micali [143]; the paper also presents a provably-secure construction of PRF, given a PRG. That is, if there is a successful attack on the constructed PRF, this attack can be used as a ‘subroutine’ to construct a successful attack on the underlying PRG. However, the construction of [143] is inefficient: it requires many applications of the PRG for a single application of the PRF. Therefore, this construction is not applied in practice.

Instead, practical systems mostly implement PRFs using standard *block ciphers*. We model block ciphers as *invertible Pseudo-Random Permutation (PRP)* and discuss them in Section 2.6. In fact, PRGs are also often implemented from a block cipher. However, for now, let us show a simple construction of a PRG from a PRF.

Exercise 2.13. Let F be a PRF over $\{0, 1\}^n$ bits, and let $k, r \in \{0, 1\}^n$. Prove that $f(k) = (F_k(1) \oplus F_k(2))$ is a PRG.

Another option is to construct candidate pseudorandom functions *directly*, without assuming and using any other ‘secure’ cryptographic function, basing their security on failure to ‘break’ them using known techniques and efforts by expert cryptanalysts. In fact, pseudorandom functions are among the cryptographic functions that seem good candidates for such ‘ad-hoc’ constructions; it is relatively easy to come up with a reasonable candidate PRF, which will not be trivial to attack.

Finally, it is not difficult to *combine* two *candidate PRFs* F', F'' , over the same domain and range, into a combined PRF F which is secure as long as either F' or F'' is a secure PRF. We refer to such a construction as a *robust combiner*. Constructions of robust combiners are known for many cryptographic

primitives. The following lemma, from [160], presents a trivial yet efficient robust combiner for PRFs.

Lemma 2.1 (Robust combiner for PRFs). *Let $F', F'' : \{0, 1\}^* \times D \rightarrow R$ be two polynomial-time computable functions, and let:*

$$F_{(k', k'')}(x) \equiv F'_{k'}(x) \oplus F''_{k''}(x) \quad (2.39)$$

If either F' or F'' is a PRF, then F is a PRF. Namely, this construction is a robust combiner for PRFs.

Proof: see [160]. \square

2.5.10 The key separation principle and application of PRF

In the PRF robust combiner (Eq. 2.39), we used *separate* keys for the two candidate-PRF functions F', F'' . In fact, this is *necessary*, as the following exercise shows.

Exercise 2.14 (Independent keys are required for PRF robust combiners). *Let $F', F'' : \{0, 1\}^* \times D \rightarrow \{0, 1\}^*$ be two polynomial-time computable functions, and let $F_k(x) = F'_k(x) \oplus F''_k(x)$. Demonstrate that the fact that one of F', F'' is a PRF may not suffice to ensure that F would be a PRF.*

Solution: Suppose $F' = F''$. Then for every k, x holds: $F_k(x) = F'_k(x) \oplus F''_k(x) = F'_k(x) \oplus F'_k(x) = 0^{|F'_k(x)|}$. Namely, for any input x and any key k the output of $F_k(x)$ is an all-zeros string ($F_k(x) \in 0^*$). Hence F is clearly not a PRF. \square

This is an example of the general *Key separation* principle, which we present below. In fact, the study of robust combiners often helps to better understand the properties of cryptographic schemes and to learn how to write cryptographic proofs.

Principle 7 (Key separation). *Use separate, independently-pseudorandom keys for each different cryptographic scheme, as well as for different types/sources of plaintext and different periods.*

The principle combines three main motivations for the use of separate, independently-pseudorandom keys:

Per-goal keys: use separate keys for different cryptographic schemes. A system may use multiple different cryptographic functions or schemes, often for different goals, e.g., encryption vs. authentication. In this case, security may fail if the same or related keys are used for multiple different functions. Exercise 2.14 above is an example.

Limit information for cryptanalysis. By using separate, independently-pseudorandom keys, we reduce the amount of information available to the attacker (ciphertext, for example).

Limit the impact of key exposure. Namely, by using separate keys, we ensure that exposure of some of the keys will not jeopardize the secrecy of communication encrypted with the other keys.

pseudorandom functions (PRFs) have many applications in cryptography, including encryption, authentication, key management and more. One important application is *derivation of multiple separate keys* from a single shared secret key k . Namely, a PRF, say f , is handy whenever two parties share one secret key k and need to derive multiple *separate, independently pseudorandom* keys k_1, k_2, \dots from k . A common way to achieve this, is for the two parties to use some set of identifiers $\gamma_1, \gamma_2, \dots$, a distinct identifier for each derived key, and compute each key k_i as: $k_i = f_k(\gamma_i)$.

As another example, system designers often want to limit the impact of key exposure due to cryptanalysis or to system attacks. One way to reduce the damage from key exposures is to change the keys periodically, e.g., use key k_d for day number d :

Example 2.6 (Using a PRF for independent per-period keys). *Assume that Alice and Bob share one master key k_M . They may derive a shared secret key for day d as $k_d = \text{PRF}_{k_M}(d)$. Even if all the daily keys are exposed, except the key for one day \hat{d} , the key for day \hat{d} remains secure as long as k_M is kept secret.*

2.6 Block Ciphers and PRPs

Modern symmetric encryption schemes are built in modular fashion, using a basic building block - the *block cipher*. A block cipher is defined by a pair of keyed functions, E_k, D_k , such that the domain and the range of both E_k and D_k are $\{0, 1\}^n$, i.e., binary strings of fixed length n ; for simplicity, we (mostly) use n for the length of both keys and blocks, as well as the security parameter, although in some ciphers, these are different numbers.

Block ciphers should satisfy the *correctness* requirement: $m = D_k(E_k(m))$ for every $k, m \in \{0, 1\}^n$. Notice that the correctness requirement should hold *always*, i.e., not only with high probability. This is in contrast to security requirements, which are typically defined to hold only against efficient (PPT) adversaries - and allow a negligible failure probability, i.e., the adversary may win with negligible probability. But there is no reason to allow any probability for incorrect decryption.

Block ciphers may be the most important basic cryptographic building blocks. Block ciphers are in wide use in many practical systems and constructions, and two of them were standardized by NIST - the *Data Encryption Standard (DES)* (1977-2001) [244], the first standardized cryptographic scheme, and its successor, the *Advanced Encryption Standard (AES)* (2002-present) [91] (Fig. 2.19). DES was replaced, since it was no longer considered secure; the main reason was simply that improvements in hardware made exhaustive-search feasible, due to the relatively short, 56-bit key. Another reason for reduced confidence in DES - even in longer-key variants - was the presentation of *differential cryptanalysis*

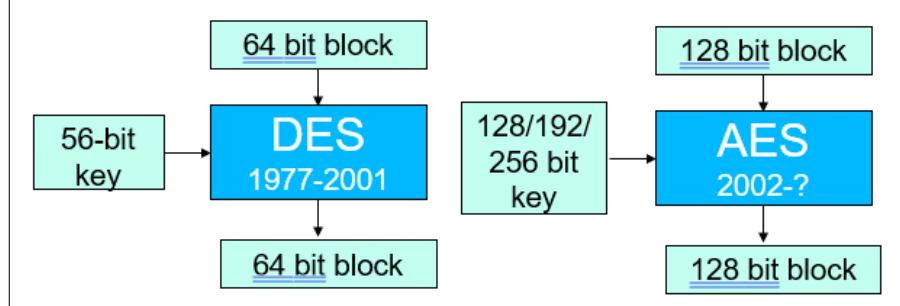


Figure 2.19: High-Level view of the NIST standard block ciphers: AES (current) and DES (obsolete).

and *linear cryptanalysis*, two strong cryptanalysis attacks, which are quite generic, namely, effective against many cryptographic designs - including DES. Indeed, AES was designed for resiliency against these and other known attacks, and so far, no published attack against AES appear to justify concerns.

We present a simplified explanation and example of differential cryptanalysis below, and encourage interested readers to follow up in the extensive literature on cryptanalysis in general and these attacks, e.g., [109, 183, 195]; [195] also gives excellent overview of block ciphers.

While the definition of correctness for block ciphers (above) is simple and widely-accepted, there is not yet universal agreement on the security requirements. We adopt the common approach, which requires block ciphers to be a pair of *Pseudo-Random Permutations (PRP)*. In the following subsection, we define pseudo-random permutations; then, in subsection 2.6.2, we discuss the security of block ciphers.

2.6.1 Random and Pseudo-Random Permutations

After discussing random functions and PRFs, we now introduce two related concepts: a *random permutation* and a *pseudo-random permutation (PRP)*.

Random permutations. A *permutation* is a function $\pi : D \rightarrow D$ mapping a domain D onto itself, where every element is mapped to a distinct element, namely:

$$(\pi : D \rightarrow D) \text{ is a permutation} \iff (\forall x, x' \in D) (\pi(x) \neq \pi(x')) \quad (2.40)$$

Note that a permutation may map an element onto itself, i.e., $\pi(x) = x$ is perfectly legitimate.

We use $\text{Perm}(D)$ to denote the set of all permutations over domain D . Selection of a random permutation over D , i.e., selecting $\rho \xleftarrow{\$} \text{Perm}(D)$, is similar to selection of a random function (Equation 2.27) - except for the need

to avoid *collisions*. A collision is a pair of elements (x, x') , both mapped to the same element: $y = \rho(x) = \rho(x')$.

One natural way to think about this selection, is as being done *incrementally*, mapping one input at a time. Let $D' \subseteq D$ be the set of elements we didn't map yet, and $R \subseteq D$ be the set of elements to which we didn't map any element yet; initially, $R = D' = D$. Given any ‘new’ element $x \in D'$, select: $\rho(x) \xleftarrow{\$} R$, and then remove x from D' and $\rho(x)$ from R .

Using this process, for a small domain, e.g., $D = \{0, 1\}^n$ for small n , the selection of a random permutation ρ is easy and can be done manually - similarly to the process for selecting a random function (over small domain and range). The process requires $O(2^n)$ coin tosses, time and storage. For example, use Table 2.3 to select two random permutations over domain $D = \{0, 1\}^2$, and notice the number of coin-flips required.

Function	Domain	00	01	10	11	coin-flips
ρ_1	$\{0, 1\}^2$					
ρ_2	$\{0, 1\}^2$					

Table 2.3: Do-it-yourself table for selecting random permutations ρ_1, ρ_2 over domain $D = \{0, 1\}^2$.

Pseudo-Random Permutation (PRP). Similarly to a PRF, a *Pseudo-Random Permutation (PRP)* over domain D , denoted $E_k(\cdot)$, is an efficient algorithm which cannot be distinguished efficiently from a random permutation $\rho \xleftarrow{\$} \text{Perm}(D)$, provided that the key k provided is ‘sufficiently long’ and chosen randomly.

In the definition, the adversary \mathcal{A} has *oracle access* to one of two functions: either $E_k(\cdot)$, keyed with a random n -bit key k , or a random permutation over domain D , i.e., $\rho \xleftarrow{\$} \text{Perm}(D)$. We denote these two cases by $\mathcal{A}^{E_k(\cdot)}$ and $\mathcal{A}^{\rho(\cdot)}$, respectively. The adversary should try to distinguish between these two cases, e.g., by outputting the string ‘Rand’ if given access to the random permutation $\rho(\cdot)$, and outputting, say, ‘not random’, if given access to the PRP $E_k(\cdot)$. The idea of the definition is illustrated in Fig. 2.20.

Note that the definition allows arbitrary length of the key (n), since indistinguishability is only defined asymptotically - for sufficiently long keys.

Definition 2.8. A pseudo-random Permutation (PRP) is a polynomial-time computable function $E_k(x) : \{0, 1\}^* \times D \rightarrow D \in \text{PPT}$ s.t. for all PPT algorithms \mathcal{A} , $\varepsilon_{\mathcal{A}, E}^{\text{PRP}}(n) \in \text{NEGL}(n)$, i.e., is negligible, where the advantage $\varepsilon_{\mathcal{A}, E}^{\text{PRP}}(n)$ of the PRP E against adversary \mathcal{A} is defined as:

$$\varepsilon_{\mathcal{A}, E}^{\text{PRP}}(n) \equiv \Pr_{k \xleftarrow{\$} \{0, 1\}^n} [\mathcal{A}^{E_k}(1^n)] - \Pr_{\rho \xleftarrow{\$} \text{Perm}(D)} [\mathcal{A}^\rho(1^n)] \quad (2.41)$$

The probabilities are taken over random coin tosses of \mathcal{A} , and random choices of the key $k \xleftarrow{\$} \{0, 1\}^n$ and of the function $\rho \xleftarrow{\$} \text{Perm}(D)$.

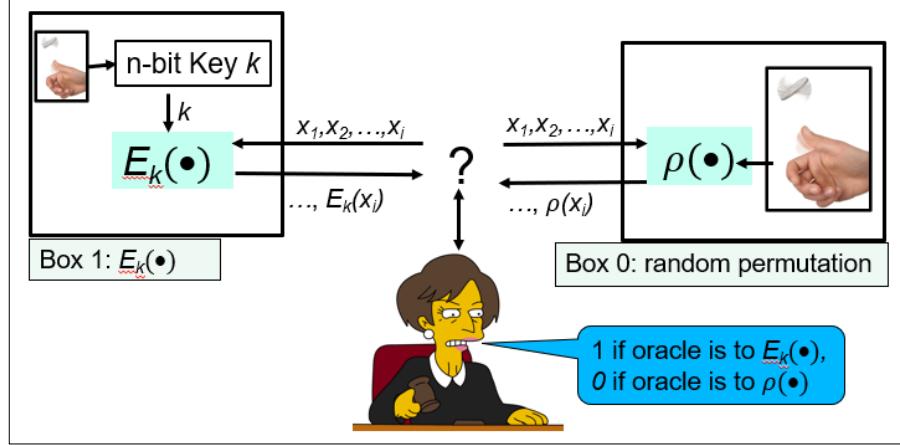


Figure 2.20: The Pseudo-Random Permutation (PRP) Indistinguishability Test. We say that function $E_k(x) : \{0, 1\}^* \times D \rightarrow D$ is a (secure) pseudo-random permutation (PRP), if no distinguisher D can efficiently distinguish between $E_k(\cdot)$ and a random permutation $\rho \xleftarrow{\$} \text{Perm}(D)$ over domain D , when the key k is a randomly-chosen sufficiently-long binary string.

Block cipher as an invertible PRP (or: PRP and its inverse). The reason that we used the letter E to denote the candidate-PRP function, is to apply a PRP as a *block cipher*. A block cipher is one of the most important cryptographic mechanisms, with multiple applications and implementations, often used as a ‘building block’ to construct other mechanisms. This definition of a block cipher is a simple extension of Definition 2.8.

Definition 2.9. Let D be a finite domain. Given a permutation $\rho : D \rightarrow D$ over domain D , define the inverse of ρ , denoted $\rho^{-1} : D \rightarrow D$, as the permutation over D such that $(\forall x \in D) x = \rho^{-1}(\rho(x))$.

A block cipher over domain D is a pair of keyed polynomial-time computable functions, (E_k, D_k) over domain D , which satisfy:

Correctness: for every $x \in D$ and every key $k \in \{0, 1\}^*$ holds $x = D_k(E_k(x))$.

Security (indistinguishability): the pair (E, D) is indistinguishable from a random invertible permutation over domain D .

Where we say that (E, D) is indistinguishable from a random invertible permutation over domain D if for every PPT algorithm \mathcal{A} , the invertible Pseudo-Random Permutation (iPRP)-advantage of \mathcal{A} against (E, D) , $\varepsilon_{\mathcal{A}, (E, D)}^{iPRP}(n)$, is a negligible function of n , where $\varepsilon_{\mathcal{A}, (E, D)}^{iPRP}(n)$ is defined as:

$$\varepsilon_{\mathcal{A}, (E, D)}^{iPRP}(n) \equiv \Pr_{k \xleftarrow{\$} \{0, 1\}^n} [\mathcal{A}^{E_k, D_k}(1^n)] - \Pr_{\rho \xleftarrow{\$} \text{Perm}(D)} [\mathcal{A}^{\rho, \rho^{-1}}(1^n)] \quad (2.42)$$

Where $\mathcal{A}^{f,g}(1^n)$ is the oracle notation, denoting algorithm \mathcal{A} running with input 1^n and oracles f and g , as in Definition 1.3.

The symbols (E, D) are often used for the functions of the block cipher, since one basic application of a block cipher is as an encryption scheme; intuitively, E is ‘encryption’ and D is ‘decryption’. However, as we will soon see, a block cipher does *not* meet the (strong) definition of secure encryption, which we present next.

One natural - and important - question is the relation between a PRP over domain D , and a PRF whose domain *and range* are both D . Somewhat surprisingly, it turns out that *a PRP over D is indistinguishable from a PRF over D* . This important result is called the *PRP/PRF Switching Lemma*, and has multiple proofs; we recommend the proof in [298]. Note that the lemma provides a relation between the advantage functions; this is an example of *concrete security*.

Lemma 2.2 (The PRP/PRF Switching Lemma). *Let E be a polynomial-time computable function $E_k(x) : \{0, 1\}^* \times D \rightarrow D \in \text{PPT}$, and let \mathcal{A} be a PPT adversary. Then:*

$$|\varepsilon_{\mathcal{A}, E}^{\text{PRF}}(n) - \varepsilon_{\mathcal{A}, E}^{\text{PRP}}(n)| < \frac{q^2}{2 \cdot |D|} \quad (2.43)$$

Where q is the maximal number of oracle queries performed by \mathcal{A} in each run, and the advantage functions are as defined in Equation 2.41 and Equation 2.29.

In particular, if the size of the domain D is exponential in the security parameter n (the length of key and of the input to \mathcal{A}), e.g., $D = \{0, 1\}^n$, then $\varepsilon_{\mathcal{A}, E}^{\text{PRF}}(n) - \varepsilon_{\mathcal{A}, E}^{\text{PRP}}(n) \in \text{NEGL}(n)$. In this case, E is a PRP over D , if and only if it is a PRF over D .

Proof idea: In a polynomial set of queries of a random function, there is negligible probability of having two values which will map to the same value. Hence, it is impossible to efficiently distinguish between a random function and a random permutation. The proof follows since a PRF (PRP) is indistinguishable from a random function (resp., permutation). \square

The PRP/PRF switching lemma is somewhat counter-intuitive, since, for large D , there are many more functions than permutations. Focusing on $D = \{0, 1\}^n$ for convenience, there are $(2^n)^2 = 2^{2n}$ functions over D , and ‘only’ $2^n!$, i.e., factorial¹¹ of 2^n , permutations.

Note that the loss of (concrete) security bounded by the switching lemma, is a disadvantage in using a block cipher directly as a PRF - it would be an (asymptotically) secure PRF, but the advantage against the PRF definition would be larger than the advantage against the PRP definition. Therefore, we would prefer to use one of several constructions of a PRF from a block cipher/PRP - that are efficient and simple, yet avoid this loss in security; see [33, 154].

¹¹Where for every integer i the *factorial* of i is denoted $i!$ and defined as $i! \equiv 1 \cdot 2 \cdot \dots \cdot (i-1) \cdot i$.

Function	Property
PRG f	‘Long’ output is pseudorandom, if ‘short’ input is random
Random function f	$(\forall x \in D) f(x) \xleftarrow{\$} R$ (random mapping for each input)
Random permutation π	Random 1-to-1 mapping: $\forall x \neq x' \in D) \pi(x) \neq \pi(x')$.
PRF $f_k(\cdot)$	Indistinguishable from a random function, if k is selected randomly
PRP $f_k(\cdot)$	Indistinguishable from a random permutation, if k is selected randomly
Block cipher (E, D)	(E, D) is indistinguishable from a <i>random invertible permutation</i> over domain D , and satisfy correctness: $(\forall k, m) m = D_k(E_k(m))$

Table 2.4: Comparison between random function, random permutation, PRG, PRF, PRP and block-cipher. Domain is denoted D , range is denoted R ; for permutations and block cipher, domain is also range.

See Table 2.4 for a summary and comparison of random function, random permutation, PRG, PRF and Pseudo-random Permutation (PRP).

2.6.2 Security of block ciphers

In addition to being correct ($(\forall m, k) m = D_k(E_k(m))$), a block cipher should also be *secure*. We say that a block cipher (E, D) is *secure*, if both E and D are *Pseudo-Random Permutations (PRP)*. We model the security requirements of a *block cipher* to be that of a pair of PRPs E_k, D_k over $\{0, 1\}^n$, which satisfy the correctness requirement, i.e.,:

$$(\forall m, k \in \{0, 1\}^n) m = D_k(E_k(m)) \quad (2.44)$$

Let us give an example.

Example 2.7. Let $E_k(m) = m \oplus k$ and $E'_k(m) = m + k \bmod 2^n$. Show the corresponding D, D' functions such that both (E, D) and (E', D') satisfy the correctness requirement; and show neither of them satisfy the security requirement, i.e., neither are pairs of invertible PRPs.

Solution: $D_k(c) = c \oplus k$, $D'_k(c) = c - k \bmod 2^n$. Correctness follows from the arithmetic properties. Let us now show that (E, D) is insecure; specifically, let us show that E_k is not a PRP. Recall that we need to provide a PPT adversary $\mathcal{A}^{E_k(\cdot)}$, s.t. $\epsilon_{E, \mathcal{A}}^{PRP}$ is not negligible. We present a simple adversary \mathcal{A} , that only makes two queries, and whose advantage is almost 1. The first query of \mathcal{A} will be for input 0^n ; if we denote the oracle response by $f(\cdot)$, then \mathcal{A} receives $f(0^n)$. If the oracle is for E , \mathcal{A} receives $E_k(0^n) = 0^n \oplus k = k$, i.e., the key k . Intuitively, this clearly ‘breaks’ the system; let us show exactly how, but from this point, our solution holds for the general case where the adversary found k (if the oracle is for $f(\cdot) = E_k(\cdot)$).

Our second query can be for any other value (not 0^n), e.g., let's make the query 1^n , so we now receive $f(1^n)$, where f is either E_k or a random permutation. Adversary \mathcal{A} checks if $f(1^n)$ (which it received from the oracle) is the same as $E_k(1^n)$ (which \mathcal{A} computes, since it believes it knows k). If the two are identical, then probably $f(\cdot) = E_k(\cdot)$, i.e., \mathcal{A} returns 1 (PRP); otherwise, then for sure f is a random permutation (and \mathcal{A} returns 0). So, the advantage of \mathcal{A} is almost 1, specifically:

$$\begin{aligned}\varepsilon_{\mathcal{A},E}^{PRP}(n) &= \Pr_{k \xleftarrow{\$} [0,1]^n} (\mathcal{A}^{E_k} = 1) - \Pr_{f \xleftarrow{\$} \text{Perm}(\{0,1\}^n)} (A^f = 1) \\ &= \Pr_k (E_k(1^n) = E_k(1^n)) - \Pr_{f \xleftarrow{\$} \text{Perm}(\{0,1\}^n)} (E_k(1^n) = f(1^n))\end{aligned}$$

Now, if f is a random permutation, then $f(1^n)$ is a random n bit string; since there are 2^n n -bit strings, then the probability of $f(1^n)$ to be any specific string, including $E_k(1^n)$, is 2^{-n} . Hence:

$$\varepsilon_{\mathcal{A},E}^{PRP}(n) = 1 - \frac{1}{2^n} \approx 1$$

Now, notice that *the same adversary* \mathcal{A} also distinguishes E' ; we leave to the reader to substitute the values as necessary; these minimal changes are only required until \mathcal{A} ‘finds’ k , from that point, the solution is exactly identical. \square

Another desirable property of block ciphers is that they have a simple *robust combiner*, i.e., a method to combine two or more candidate block ciphers (E', D') , $, (E'', D'')$ into one ‘combined’ pair (E, D) which is a secure block cipher provided one or more of the candidates is a secure block cipher. See [160]. Basically, assuming both (E', D') and (E'', D'') satisfy correctness, then their *cascade* is a secure block cipher, i.e., $E_{k',k''}(x) = E'_{k'}(E''_{k''}(x))$, $D_{k',k''}(x) = D''_{k''}(D'_{k'}(x))$. See [160].

2.6.3 The Feistel Construction: $2n$ -bit Block Cipher from n -bit PRF

It is not too difficult to design a candidate PRF from basic operations. However, designing a candidate PRP directly from basic operations seems harder, since we need to ensure that every input is mapped to a distinct output. Directly constructing a block cipher, an invertible PRP, seems harder, since we need to ensure the permutation property and find the inverse permutation, but still prevent vulnerability. This motivates the design of a PRP by using a PRF.

One way to turn a PRF into a PRP is simply to *use* a PRF as a PRP. The *PRP/PRF switching lemma* (Lemma 2.2) shows that every PRF over a domain D , is indistinguishable from a PRP over D , and vice versa. Namely, no computationally-bounded (PPT) adversary is likely to distinguish between a PRP and a PRF (over domain D). So, it seems that we can just use a PRF instead of a PRP.

However, a PRF is allowed to have some *collisions*, i.e., values $x \neq x' \in D$ such that $F_k(x) = F_k(x')$, for the same key k . Such collisions are undesirable

for a block cipher; indeed, we require that a block cipher (E, D) will ensure *correctness*, i.e., that $m = D_k(E_k(m))$, for every key k and message m . Clearly this will not hold if we use a PRF, with collisions, to implement the block cipher.

In this subsection, we study the Feistel construction of a PRP from a PRF; furthermore, the construction is of a PRP with input of $2n$ bits, given a PRF with inputs and outputs of n bits. Such a design is not trivial; see the following two exercises.

Exercise 2.15. *Let f be a PRF from n -bit strings to n -bit strings, and define $g_{k_L, k_R}(m_L \parallel m_R) \equiv f_{k_L}(m_L) \parallel f_{k_R}(m_R)$. Show that g is neither a PRF nor a PRP (over $2n$ -bit strings).*

Hint: given a black box containing g or a random permutation over $2n$ -bit strings, design a distinguishing adversary \mathcal{A} as follows. \mathcal{A} makes two queries, one with input $x = 0^{2n}$ and the other with input $x' = 0^n \parallel 1^n$. Denote the corresponding outputs by $y = y_{0, \dots, 2n-1}$ and $y' = y'_{0, \dots, 2n-1}$. If the box contained g , then $y_{0, \dots, n-1} = y'_{0, \dots, n-1}$. In contrast, if the box contained a random function, then the probability that $y_{0, \dots, n-1} = y'_{0, \dots, n-1}$ is very small - only 2^{-n} . The probability is about as small as if the box contained a PRP. \square

The next exercise presents a slightly more elaborate scheme, which is essentially a reduced version of the Feistel construction (presented next).

Exercise 2.16. *Let f be a PRF from n -bit strings to n -bit strings. Show that $g_{k_L, k_R}(m_L \parallel m_R) = m_L \oplus f_{k_R}(m_R) \parallel m_R \oplus f_{k_L}(m_L)$ is not a PRP (over $2n$ -bit strings).*

We next present the Feistel construction, the most well known and simplest construction of a PRP - in fact, an invertible PRP (block cipher) - from a PRF. As shown in Fig. 2.21, the Feistel cipher transforms an n -bit PRF into a $2n$ -bit invertible PRP.

Formally, given a function $y = f_k(x)$ with n -bit keys, inputs and outputs, the three-rounds Feistel $g_k(m)$ is defined as:

$$\begin{aligned} L_k(m) &= m_{0, \dots, n-1} \oplus F_k(m_{n, \dots, 2n-1}) \\ R_k(m) &= F_k(L_k(m)) \oplus m_{n, \dots, 2n-1} \\ g_k(m) &= L_k(m) \parallel R_k(m) \end{aligned}$$

Note that we consider only a ‘three rounds’ Feistel cipher, and use the same underlying function F_k in all three rounds, but neither aspect is mandatory. In fact, the Feistel cipher is used in the design of DES and several other block ciphers, typically using more rounds (e.g., 16 in DES), and often using different functions at different rounds.

Luby and Rackoff [220] proved that a Feistel cipher of three or more ‘rounds’, using a PRF as $F_k(\cdot)$, is an invertible PRP, i.e., a block cipher.

One may ask, why use the Feistel design rather than directly design an invertible PRP? Indeed, this is done in AES, which does not follow the Feistel

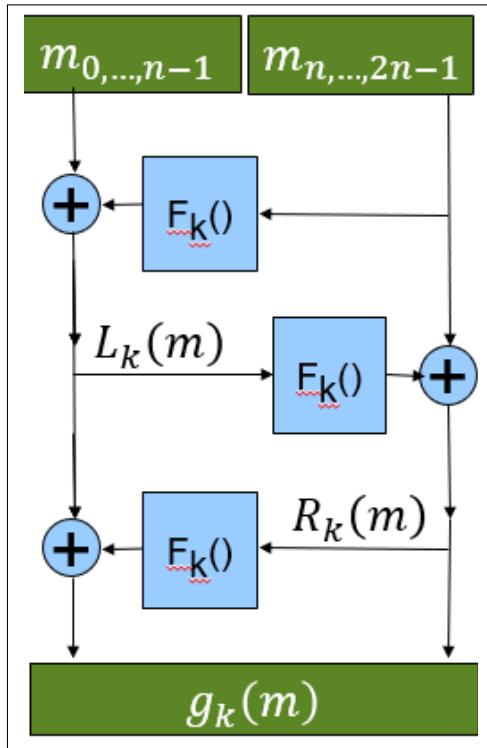


Figure 2.21: Three ‘rounds’ of the Feistel Cipher, constructing a block cipher (invertible PRP) from a PRF $F_k(\cdot)$. The Feistel cipher is used in DES (but not in AES). Note: most publications present the Feistel cipher a bit differently, by ‘switching sides’ in each round.

cipher design. An advantage of using the Feistel design is that it allows the designer to focus on the pseudo-randomness requirements when designing the PRF, without having simultaneously to make sure that the design is also an invertible permutation. Try to design a PRP, let alone an invertible PRP, and compare it to using the Feistel cipher!

2.7 Defining secure encryption

In the previous section, we defined PRG, PRF and PRP; in this section, we finally make the next step and define secure encryption.

The definition of secure encryption is quite subtle. In fact, people have been designing - and attacking - cryptosystems for millennia, without a precise definition of the security goals! This only changed with the seminal paper of Goldwasser and Micali [144], which presented the first precise definition of secure encryption, along with a design which was proven secure (under reasonable assumptions); this paper is one of cornerstones of modern cryptography.

It may be surprising that defining secure encryption is so challenging; we therefore urge you to attempt the following exercise, where you are essentially challenged to try to define secure encryption on your own, before reading the rest of this section and comparing with the definition we present.

Exercise 2.17 (Defining secure encryption). *Define secure symmetric encryption, as illustrated in Figure 1.4. Refer separately to the two aspects of security definitions: (1) the attack model, i.e., the capabilities of the attacker, and (2) the success criteria, i.e., what constitutes a successful attack and what constitutes a secure encryption scheme.*

2.7.1 Attack model

Security definition require a precise *attack model*, defining the maximal expected capabilities of the attacker. We discussed already some of these capabilities. In particular, we already discussed the *computational limitations* of the attacker: in Section 2.4 we discussed the *unconditional security* model, where attackers have unbounded computational resources, and from subsection 2.5.2 we focus on *Probabilistic Polynomial Time (PPT)* adversaries, whose computation time is bounded by some polynomial in their input size.

Another important aspect of the attacker model is the interactions with the attacked scheme and the environment. In Section 2.2, we introduced the *cipher-text only (CTO)*, *known-plaintext attack (KPA)*, *chosen plaintext attack (CPA)* and *chosen ciphertext attack (CCA)* attack models. Specifically, in a *chosen-plaintext attack*, the adversary can choose plaintext and receive the corresponding ciphertext (encryption of that plaintext), and in *chosen-ciphertext attack*, the adversary can choose *ciphertext* and receive the corresponding plaintext (its decryption), or error message if the ciphertext does not correspond to well-encrypted plaintext.

It is desirable to allow for attackers with maximal capabilities. Therefore, when we evaluate cryptosystems, we are interested in their resistance to all types of attacks, and especially the stronger ones - CCA and CPA. On the other hand, when we design systems using a cipher, we try to limit that attacker's capabilities.

For example, one approach to foil CCA attacks is to apply some simple *padding function* *pad* to add redundancy to the plaintext before encryption; the padding function may be as simple as appending a fixed string. For example, given message m , key k , encryption scheme (E, D) and a simple padding function, e.g., $pad(m) = m \parallel 0^l$, i.e., concatenate l zeros, we now encrypt by computing $c = E_k(pad(m)) = E_k(m \parallel 0^l)$. This allows the decryption process to identify *invalid ciphertexts*. Namely, given $c = E_k(pad(m)) = E_k(m \parallel 0^l)$, then $D_k(c) = m \parallel 0^l$, and we output m as usual; but if the output of D_k does not contain l trailing zeros, then we identify faulty ciphertext. This approach often helps to make it hard or infeasible for the attacker to apply a chosen-ciphertext attack; in particular, a random ciphertext would almost always be detected as faulty.

Note, however, that adding redundancy to the plaintext may make it *easier* to perform ciphertext-only attacks; see Principle 9.

Also, some encryption and padding functions may still allow CCA attacks, as we show in the following exercise.

Exercise 2.18. *Show that the simple padding function $\text{pad}(m) = m + 0^l$, fails to prevent CCA attacks against the PRG-stream-cipher (Fig. 2.11), and against at least one or two other ciphers we discussed so far.*

2.7.2 The Indistinguishability-Test for Shared-Key Cryptosystems

Intuitively, the security goal of encryption is *confidentiality*: to transform plaintext into ciphertext in such way that will allow specific parties ('recipients') - and only them - to perform decryption, transforming the ciphertext back to the original plaintext. However, the goal as stated may be interpreted to only forbid recovery of the exact, complete plaintext; but what about recovery of partial plaintext?

For example, suppose an eavesdropper can decipher half of the characters from the plaintext - is this secure? We believe most readers would not agree. What if she can decipher less, say one character? In some applications, this may be acceptable; in others, even exposure of one character may have significant consequences.

Intuitively, we require that an adversary cannot learn *anything* given the ciphertext. This may be viewed as extreme; for example, in many applications the plaintext includes known fields, and their exposure may not be a concern. However, it is best to minimize assumptions and use definitions and schemes which are secure for a wide range of applications.

Indeed, in general, when we design a security system, cryptographic or otherwise, it is important to clearly define both aspects of security: the attack model (e.g., types of attacks 'allowed' and any computational limitations), as well as the success criteria (e.g., ability to get merchandise without paying for it). Furthermore, it is difficult to predict the actual environment in which a system would be used. Therefore, following the *conservative design principle* (Principle 3), our definition should prevent the adversary from learning any information about the plaintext from the ciphertext.

Let us assume that you agree that it would be best to require that an adversary cannot learn *anything* from the ciphertext. How do we ensure this? This is not so easy. The seminar paper by Goldwasser and Micali [144] presented two definitions and showed them to be equivalent: *semantic secure encryption* and *indistinguishability*. We will only present the latter, since we find it easier to understand and use, and resembles the PRF, PRG and Turing indistinguishability tests (Figure 2.18, Figure 2.13 and Figure 2.12, respectively).

Intuitively, an encryption scheme ensures indistinguishability if an attacker cannot distinguish between encryption of any two given messages. But, again, turning this into a 'correct' and precise definition requires care.

The concept of indistinguishability is reminiscent of disguises; it may help to consider the properties we can hope to find in an ‘ideal disguise service’:

Any two disguised persons are indistinguishable: we cannot distinguish between *any* two well-disguised persons. Yes, even Rachel from Leah!¹²

Except, the two persons should have the ‘same size’: assuming that a disguise is of ‘reasonable size’ (overhead), a giant can’t be disguised to be indistinguishable from a dwarf!

Re-disguises should be different: if we see Rachel in disguise, and then she disappears and we see a new disguise, we should not be able to tell if it is Rachel again, in new disguise - or any other disguised person! This means that disguises must be randomized or stateful, i.e., every two disguises of the same person (Rachel) will be different.

We will present corresponding properties for indistinguishable encryption:

Encryptions of any two messages are indistinguishable. to allow arbitrary applications, we allow the attacker to choose the two messages. However, there is one restriction: *the two messages should be same length.*

Re-encryptions should be different: the attacker should not be able to distinguish encryptions based on previous encryptions of the same messages. This means that encryption must be randomized or stateful, so that two encryptions of same message will be different. (A weaker notion of ‘deterministic encryption’ allows detection of re-encryption of a message, and is sometimes used for scenarios where state and randomization are to be avoided.)

We are finally ready to formally present the indistinguishability-based definition of secure encryption. We only present the definition for *chosen-plaintext attack (CPA)* indistinguishability (*IND-CPA*), and only for stateless encryption (Definition 2.10).

The IND-CPA test receives two inputs: the ‘challenge bit’ b (that \mathcal{A} tries to find), and the *security parameter*, which in this case is also the key length, n . The adversary is given *oracle access* to $E_k(\cdot)$, which we denote by $\mathcal{A}_k^E(\cdot)$; namely, it may give any message m and receive its encryption $E_k(m)$ - and possibly repeat this for more messages. At some point, \mathcal{A} gives a *pair of messages* m_0, m_1 , and receives $c^* = E_k(m_b)$. As we discussed above, the two messages must be of equal length, $|m_0| = |m_1|$. Finally, \mathcal{A} outputs b^* , which is the output of the test. Intuitively, \mathcal{A} ‘wins’ if $b^* = b$.

We present the IND-CPA test informally in Figure 2.22, and using pseudocode in Figure 2.23.

¹²See: Genesis 29:23, King James Bible.

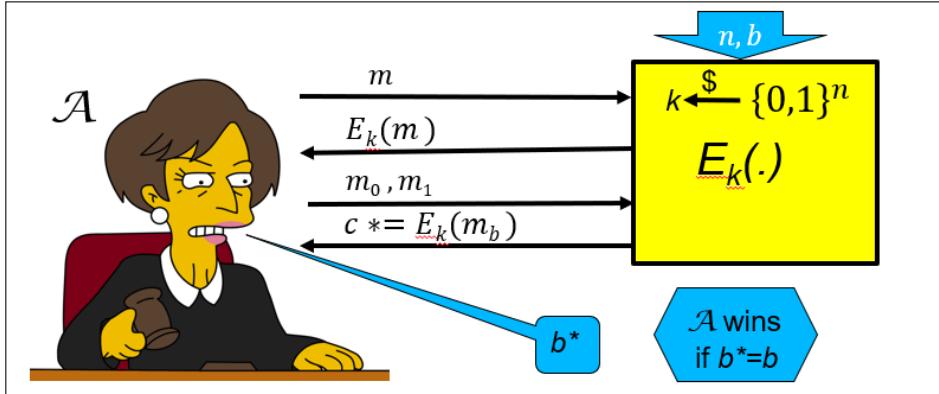


Figure 2.22: Illustration of the IND-CPA test for stateless shared-key encryption, $T_{\mathcal{A},(E,D)}^{IND-CPA}(b, n)$; see also pseudocode, Figure 2.23. Throughout the test, the adversary \mathcal{A} may ask for encryption of one or many messages m . At some point, \mathcal{A} sends two *same length* messages ($|m_0| = |m_1|$), and receives the encryption of m_b , i.e.: $E_k(m_b)$. Finally, \mathcal{A} outputs its guess b^* , and ‘wins’ if $b = b^*$. The encryption is *IND-CPA* if $\Pr(b^* = 1|b = 1) - \Pr(b^* = 1|b = 0)$ is negligible.

```

 $T_{\mathcal{A},(E,D)}^{IND-CPA}(b, n) \{$ 
     $k \xleftarrow{\$} \{0,1\}^n$ 
     $(m_0, m_1) \leftarrow \mathcal{A}^{E_k(\cdot)}(\text{‘Choose’}, 1^n)$  s.t.  $|m_0| = |m_1|$ 
     $c^* \leftarrow E_k(m_b)$ 
     $b^* = \mathcal{A}^{E_k(\cdot)}(\text{‘Guess’}, c^*)$ 
    Return  $b^*$ 
 $\}$ 

```

Figure 2.23: Pseudocode for the IND-CPA test symmetric encryption (E, D) , illustrated in Figure 2.22. The two calls to the adversary are often referred to as the ‘Choose’ phase and the ‘Guess’ phase.

Oracle notation $\mathcal{A}^{E_k(\cdot)}$. In the IND-CPA test, we use the *oracle* notation $\mathcal{A}^{E_k(\cdot)}$, defined in Def. 1.3. Namely, $\mathcal{A}^{E_k(\cdot)}$ denotes calling the \mathcal{A} algorithm, with ‘oracle access’ to the (keyed) PPT algorithm $E_k(\cdot)$, i.e., \mathcal{A} can provide arbitrary plaintext string m and receive $E_k(m)$.

Adversary \mathcal{A} chooses challenge messages. The IND-CPA test allows \mathcal{A} to choose the two challenge messages m_0, m_1 , and then receive $c^* = E_k(m_b)$, where $b \in \{0, 1\}$. Allowing \mathcal{A} to select the two messages completely may make it easier for \mathcal{A} ; in many applications, the adversary only has very limited knowledge about the possible plaintext messages. This is following the *conservative design* principle - the encryption should be appropriate for *any* application, including one in which there are only two possible plaintext messages, known to the

attacker - who ‘just’ needs to know which of them was encrypted. One classical example is when the messages are ‘attack’ or ‘retreat’; another would be ‘sell’ or ‘buy’.

Encryption must be randomized or stateful. IND-CPA encryption must either be randomized or stateful. The reason is simple: the adversary is allowed to make queries for arbitrary messages - including the ‘challenges’ m_0, m_1 . If the encryption scheme is deterministic - and stateless - then all encryptions of a message, e.g. m_0 , will return a fixed ciphertext; this will allow the attacker to trivially ‘win’ in the IND-CPA experiment. Furthermore, Exercise 2.46 shows that limiting the number of random bits per encryption may lead to vulnerability.

Using the IND-CPA test, we now define IND-CPA encryption, similarly to how we defined PRG and PRF, in Definition 2.6 and Definition 2.7, respectively. We present only the definition for stateless cryptosystems, since we mostly focus on this case.

Definition 2.10 (IND-CPA for (stateless) shared-key cryptosystems). *Let $\langle E, D \rangle$ be a stateless shared-key cryptosystem. We say that $\langle E, D \rangle$ is IND-CPA, if every efficient adversary $\mathcal{A} \in PPT$ has negligible advantage $\varepsilon_{\langle E, D \rangle, \mathcal{A}}^{IND-CPA}(n) \in NEGL(n)$, where:*

$$\varepsilon_{\langle E, D \rangle, \mathcal{A}}^{IND-CPA}(n) \equiv \Pr \left[T_{\mathcal{A}, \langle E, D \rangle}^{IND-CPA}(1, n) = 1 \right] - \Pr \left[T_{\mathcal{A}, \langle E, D \rangle}^{IND-CPA}(0, n) = 1 \right] \quad (2.45)$$

Where the probability is over the random coin tosses in IND-CPA (including of \mathcal{A} and E).

Exercise 2.19. Consider the following alternative advantage functions:

$$\begin{aligned} \tilde{\varepsilon}_{\langle E, D \rangle, \mathcal{A}}^{IND-CPA}(n) &\equiv \Pr \left[T_{\mathcal{A}, \langle E, D \rangle}^{IND-CPA}(1, n) = 1 \right] - \Pr \left[T_{\mathcal{A}, \langle E, D \rangle}^{IND-CPA}(1, n) = 0 \right] \\ \hat{\varepsilon}_{\langle E, D \rangle, \mathcal{A}}^{IND-CPA}(n) &\equiv \Pr \left[T_{\mathcal{A}, \langle E, D \rangle}^{IND-CPA}(1, n) = 1 \right] - \Pr \left[T_{\mathcal{A}, \langle E, D \rangle}^{IND-CPA}(0, n) = 0 \right] \end{aligned}$$

Show that both are not reasonable definitions for advantage function, by presenting (simple) adversaries which achieve significant advantage for any cryptosystem.

Indistinguishability for the CTO, KPA and CCA attack models
Definition 2.10 focuses on Chosen-Plaintext Attack (CPA) model.

Modifying this definition for the case of chosen-ciphertext (CCA) attacks requires a further (quite minor) change and extension, to prevent the attacker from ‘abusing’ the decryption oracle to decrypt the challenge ciphertext.

Modifying the definition for Cipher-Text-Only (CTO) attack and Known-Plaintext Attack (KPA) is more challenging. For KPA, the obvious question is which plaintext-ciphertext messages are known; this may be solved by using

random plaintext messages, however, in reality, the known-plaintext is often quite specific.

It is similarly challenging to modify the definition so it covers CTO attacks, where the attacker must know *some* information about the plaintext distribution. This information may be related to the specific application, e.g., when the plaintext is English.

In other cases, information about the plaintext distribution may be derived from system design. One example is text encoded using a code with some built-in error-detection capability, e.g., the ASCII encoding [80], where one of the bits in every character is the parity of the other bits. An even more extreme example is in GSM, where the plaintext is the result of the application of an Error-Correcting Code (ECC), providing significant redundancy which even allows a CTO attack on GSM's A5/1 and A5/2 ciphers [20]. In such a case, the amount of redundancy in the plaintext can be compared to that provided by a KPA attack. We consider it a CTO attack, as long as the attack does not require knowledge of all or much of the plaintext corresponding to the given ciphertext messages.

Some systems, including GSM, allow the attacker to guess all or much of the plaintext for some of the ciphertext messages, e.g., when sending a predictable message at a specific time. Such systems violate the Conservative Design Principle (principle 3), since a KPA-vulnerability of the cipher renders the system vulnerable. A better system design would limit the adversary's knowledge about the distribution of plaintexts, requiring a CTO vulnerability to attack the system.

2.7.3 The Indistinguishability-Test for Public-Key Cryptosystems (PKCs)

We next define the CPA-indistinguishability for public key cryptosystems (PKC; see Figure 1.5). The definition is a minor variation of the indistinguishability-test for shared-key cryptosystems (Definition 2.10), and even a bit simpler. In fact, let us *first* present the definition, as well as the IND-CPA test for PKCS (Figure 2.24), and only then point out and explain the differences; this would allow the reader to play ‘find the differences’, comparing to Definition 2.10.

Definition 2.11 (IND-CPA-PK). Let $\langle KG, E, D \rangle$ be a public-key cryptosystem. We say that $\langle KG, E, D \rangle$ is IND-CPA-PK, if every efficient adversary $\mathcal{A} \in PPT$ has negligible advantage $\varepsilon_{\langle KG, E, D \rangle, \mathcal{A}}^{IND-CPA-PK}(n) \in NEGL(n)$, where:

$$\varepsilon_{\langle KG, E, D \rangle, \mathcal{A}}^{IND-CPA-PK}(n) \equiv \Pr \left[T_{\mathcal{A}, \langle KG, E, D \rangle}^{IND-CPA}(1, n) = 1 \right] - \Pr \left[T_{\mathcal{A}, \langle KG, E, D \rangle}^{IND-CPA}(0, n) = 1 \right] \quad (2.46)$$

Where the probability is over the random coin tosses in IND-CPA (including of \mathcal{A} and E).

In IND-CPA-PK (Definition 2.11), the adversary is *given* the public key e . Hence, ADV can encrypt at will, without the need to make encryption queries,

```

 $T_{\mathcal{A},(KG,E,D)}^{IND-CPA}(b,n) \{$ 
     $(e,d) \xleftarrow{\$} KG(1^n)$ 
     $(m_0, m_1) \leftarrow \mathcal{A}(\text{'Choose'}, e) \text{ s.t. } |m_0| = |m_1|$ 
     $c^* \leftarrow E_e(m_b)$ 
     $b^* = \mathcal{A}(\text{'Guess'}, (c^*, e))$ 
    Return  $b^*$ 
}

```

Figure 2.24: The *IND-CPA-PK* test for public-key encryption (KG, E, D) . Notice that this test does not use the decryption key d , generated in the first step.

as enabled by the oracle calls in Definition 2.10, and we removed the oracle. Another change is purely syntactic: the cryptosystem includes an explicit key generation algorithm KG , while for the shared-key cryptosystem, we assumed the (typical) case where the keys are just random n -bit strings.

We discuss three specific public key cryptosystems, all in Chapter 6: the DH and El-Gamal PKCs in Section 6.4, and the RSA PKC in Section 6.5.

2.7.4 Design of Secure Encryption: the Cryptographic Building Blocks Principle

We next discuss the *design of secure symmetric encryption schemes*. It would be great if we could use encryption schemes which are provably secure, e.g., proven to be IND-CPA (Definition 2.10), *without assumptions* on computational-hardness of some underlying functions. However, this is unlikely; let us explain why.

A provably IND-CPA encryption implies $P \neq NP$. IND-CPA implies that there is no efficient (PPT) algorithm that can distinguish between encryption of two given messages, i.e., the IND-CPA test is not in the polynomial-complexity class P , containing problems which have a polynomial-time algorithm. On the other hand, surely it is easy to ‘win’ in the test, given the key; which implies that the IND-CPA test *is* in the non-deterministic polynomial complexity class NP , containing problems which have a polynomial-time algorithm - if given a *hint* (in our case, the key). Taken together, this would have shown that the complexity class P is *strictly smaller* than the complexity class NP , i.e., $P \neq NP$. Now, that would be a solution to the most fundamental question in the theory of computational complexity!

It is not practical to require the encryption algorithm to have a property whose existence implies a solution to such a basic, well-studied open question. Therefore, both theoretical and applied cryptography consider designs whose *security relies on failed attempts in cryptanalysis*. The big question is: should

we rely on failed cryptanalysis of the scheme *itself*, or on failed cryptanalysis of *underlying components* of the scheme?

It may seem that the importance of encryption schemes should motivate the first approach, i.e., relying of failed attempts to cryptanalyze the scheme. Surely this was the approach in historical and ‘classical’ cryptology.

However, in modern applied cryptography, it is much more common to use the second approach, i.e., to construct encryption using ‘simpler’ underlying primitives, and to base the security of the cryptosystem on the security of these component modules. We summarize this approach in the following principle, and then give some justifications.

Principle 8 (Cryptographic Building Blocks). *The security of cryptographic systems should only depend on the security of a few basic building blocks. These blocks should be simple and with well-defined and easy to test security properties. More complex schemes should be proven secure by reduction to the security of the underlying blocks.*

The advantages of following the cryptographic building blocks principle include:

Efficient cryptanalysis: by focusing cryptanalysis effort on few schemes, we obtain much better validation of their security. The fact that the building blocks are simple and are selected to be easy to test makes cryptanalysis even more effective.

Replacement and upgrade: by using simple, well-defined modules, we can replace them for improved efficiency - or to improve security, in particular after being broken or when doubts arise.

Flexibility and variations: complex systems and schemes naturally involve many options, tradeoffs and variants; it is better to build all such variants using the same basic building blocks.

Robust combiners: there are known, efficient robust-combiner designs for the basic cryptographic building blocks [160]. If desired, we can use these as the basic blocks for improved security.

The cryptographic building blocks principle is key to both applied and theoretical modern cryptography. From the theoretical perspective, it is important to understand which schemes can be implemented given another scheme. There are many results exploring such relationships between different cryptographic schemes and functions, with many positive results (constructions), few negative results (proofs that efficient constructions are impossible or improbable), and very few challenging open questions.

In modern applied cryptography, the principle implies the need to define a small number of basic building blocks, which would be very efficient, simple functions - and convenient for many applications. The security of these building blocks would be established by extensive (yet unsuccessful) cryptanalysis efforts

- instead of relying on provably-secure reductions from other cryptographic mechanisms.

In fact, most cryptographic libraries contain the four such widely-used building blocks: shared-key s, s^{−1}, public-key encryption and signature schemes. Cryptographic hash functions and block ciphers are much more efficient than the public key schemes (see Table 6.1) and hence are preferred, and used in most practical systems - when public-key operations may be avoided.

In particular, block ciphers are widely used as cryptographic building blocks, as they satisfy most of the requirements of the Cryptographic Building Blocks principle. They are simple, deterministic functions with Fixed Input Length (FIL), which is furthermore identical to their output length. This should be contrasted with ‘full fledged encryption schemes’, which are randomized (or stateful) and have Variable Input Length (VIL). Which brings us to the natural question: can we use block ciphers for secure encryption - and how ?

Block ciphers vs. Secure encryption. Could we simply use a block cipher for encryption? This seems natural; block ciphers, in particular DES and AES, are often referred to as encryption schemes, and even typically use the notation (E, D) for their keyed functions. However, block ciphers do not satisfy the requirements of most definitions of encryption, e.g., the IND-CPA test of Def. 2.10.

Exercise 2.20. Explain why a PRP and a block cipher, fail the IND-CPA test (Def. 2.10).

Solution: Consider $E_k(m)$, which is either a PRP or the ‘encryption’ operation of a block cipher (i.e., a pair (E, D) of a PRP and its reverse). Then $E_k(m)$ is a function; whenever we apply it to the same message m , with the same key k , we will receive the same output $(E_k(m))$. The attacker \mathcal{A} would choose any two different messages as (m_0, m_1) , confirm that $c_0 = E_k(m_0) \neq c_1 = E_k(m_1)$, and then use these as a challenge, to receive $c^* = E_k(m_b)$. It then outputs b' s.t. $c_{b'} = c^*$. \square

On the other hand, in the next section, we discuss multiple constructions of secure encryption schemes based on block ciphers; such constructions are often referred to as *mode of operation*.

PRF as a building block. pseudorandom functions (PRFs) are also widely used in applied cryptography, however, they are not defined as a standard, and not all cryptographic libraries contain their explicit implementation. Instead, they are usually implemented using another mechanism - most often, using a block cipher. The *PRP/PRF switching lemma* (Lemma 2.2) shows that a block cipher is indistinguishable from a PRF, and hence, every block cipher is also a PRF; however, this may involve some *loss in (concrete) security*, which is bounded by the lemma - but definitely significant. Instead, one should use one of several constructions of PRF from a block cipher, which are simple,

Mode	Encryption	Flip $c_i[j] \Rightarrow$	Properties
Electronic code book (ECB)	$c_i = E_k(m_i)$	Corrupt m_i	Deterministic (distinguishable)
Per-Block Random (PBR)	$r_i \xleftarrow{\$} \{0, 1\}^n$, $c_i = (r_i, m_i \oplus E_k(r_i))$	Flip $m_i[j]$	Long ciphertext
Counter (CTR) [simplified]	$c_i = m_i \oplus E_k(i)$	Flip $m_i[j]$	Fast online, stateful (i)
Output Feed-back (OFB)	$r_0 \xleftarrow{\$} \{0, 1\}^n$, $r_i = E_k(r_{i-1})$, $c_0 \leftarrow r_0$, $c_i \leftarrow r_i \oplus m_i$	Flip $m_i[j]$	Fast online (precompute)
Cipher Feedback (CFB)	$c_0 \xleftarrow{\$} \{0, 1\}^n$, $c_i \leftarrow m_i \oplus E_k(c_{i-1})$	Corrupt m_i , flip $m_{i+1}[j]$	Can decrypt in parallel
Cipher-Block Chaining (CBC)	$c_0 \xleftarrow{\$} \{0, 1\}^n$, $c_i \leftarrow E_k(m_i \oplus c_{i-1})$	Flip $m_{i-1}[j]$, corrupt m_i	Can decrypt in parallel

Table 2.5: Encryption Modes of Operation using n -bit block cipher. ECB, OFB, CFB and CBC are from NIST ([112, 244]). The plaintext is given as a concatenation of n -bit blocks $m_1 \parallel m_2 \parallel \dots$, where each block has n bits, i.e., $m_i \in \{0, 1\}^n$. Similarly the ciphertext is produced as a set of n -bits blocks $c_0 \parallel c_1 \parallel \dots \in \{0, 1\}^n$, where $c_i \in \{0, 1\}^n$ (except for PBR, where $c_i \in \{0, 1\}^{2n}$). We use $m_i[j]$ ($c_i[j]$) to denote the j^{th} bit of the plaintext (respectively, ciphertext).

almost as efficient as the block cipher - and avoid the loss in (concrete) security; see [33, 154].

2.8 Encryption Modes of Operation

Finally we get to design symmetric encryption schemes. Following the Cryptographic Building Blocks principle, the designs are based on the much simpler block ciphers. We use the term *mode of operation* for such construction of encryption and other cryptographic schemes from block ciphers. This term, and several standard modes of operation, were defined in the DES specifications [244], and redefined in the AES specification [112], which added one more standard mode of operation (the CTR mode). Additional modes of operation were defined and proposed in different standards and publications.

In this section, we describe the standard modes of operations from [112, 244], slightly simplifying the CTR mode. For didactic purposes, we add one non-standard (and inefficient) mode of operation, which we call the *Per-Block Random (PBR)* mode. These modes are summarized in Table 2.5.

The ‘modes of operations’ in Table 2.5 are designed to turn block ciphers into more complete cryptosystems, handling goals such as:

Variable length and padding: allow encryption of arbitrary, *variable input length (VIL)* messages. All modes of operation are defined for input whose length is an integral number l of blocks. If the input may not

be an integral number of blocks, then the input should be *padded* to an integral number of blocks, before applying the encryption (i.e., the mode of operation). Correct padding can be quite simple, however, surprisingly, incorrect padding can result in serious vulnerability; We discuss padding and possible vulnerabilities in Section 2.9.

Randomization/state: Most modes use randomness to ensure independence between two encryptions of the same (or of related) messages, as required for indistinguishability-based security definitions. The exceptions are the CTR mode, which uses state instead of randomization, and the ECB mode, that uses neither - and, therefore, is not IND-CPA.

PRF: most modes (PBR, OFB, CFB and CTR), use only the encryption function E - even for decryption. This has an important implication: they may be implemented using a PRF instead of a block cipher. This may have imply better security, esp. when the same key is used for an extensive number of messages, due to improved concrete-security (smaller advantage to attacker). However, notice that there will not be such advantage if we simply use a block cipher as a PRP, relying on the PRP/PRF switching lemma (Lemma 2.2); we should use one of the (simple and efficient) constructions of PRF from block cipher, which avoid an increase in the adversary's advantage; see [33, 154]. See [27, 280].

Efficiency is important - and multi-faceted. All of the modes we present, use one block-cipher operation per message-block, and allow parallel decryption. The OFB, CTR and PBR modes also allow parallel encryption, or ‘random access’ decryption - decryption of only specific blocks from the plaintext. Another efficiency consideration is *offline precomputation*; in the CTR modes, we may conduct all the block-ciphers computations offline; after receiving the plaintext/ciphertext, we only need a single XOR operation (per block). The OFB mode has similar property but only for encryption; decryption requires the ciphertext as input to the block-cipher.

Integrity/authentication: Some modes, which, unfortunately, we do not discuss, ensure both confidentiality and integrity, preventing an attacker from modifying intercepted messages to mislead the recipient, or from forging messages as if they were sent by a trusted sender. These include the *Counter and CBC-MAC (CCM) mode* and the (more efficient) *Galois/Counter Mode (GCM) mode*. Other modes ensure *only* authenticity; we discuss one such mode, the *CBC-MAC mode*, in subsection 4.5.2.

Error localization and weak integrity: In the OFB and CTR, corruption of any number m of ciphertext bits, results in corruption of only the corresponding plaintext bits. This may help to recover from some corruptions of bits during communication, since no additional bits are lost, but also implies that the attacker may ‘flip’ plaintext bits by ‘flipping’ the corresponding ciphertext bits. In CFB and CBC, corruption of a single

ciphertext block, flips a bit in one block, and ‘corrupts’ another block - with some exceptions; this is sometimes considered as a weak form of protection of integrity, but the defense is very fragile and relying on it has resulted in several vulnerabilities. See details below.

2.8.1 The Electronic Code Book mode (ECB) mode

ECB is a naïve mode, which isn’t really a proper ‘mode’: it simply applies the block cipher separately to each block of the plaintext. Namely, to decrypt the plaintext string $m = m_1 \parallel m_2 \parallel \dots$, where each m_i is a block (i.e., $|m_i| = n$), we simply compute $c_i = E_k(m_i)$. Decryption is equally trivial: $m_i = D_k(c_i)$, and correctness of encryption, i.e., $m = D_k(E_k(m))$ for every $k, m \in \{0, 1\}^*$, follows immediately from the correctness of the block cipher $E_k(\cdot)$.

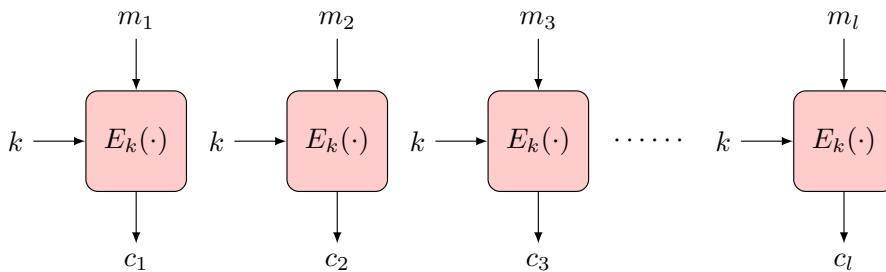


Figure 2.25: Electronic Code Book (ECB) mode encryption of plaintext message m consisting of l blocks, $m = m_1, \dots, m_l$. Adapted from [181].

Note: notations are not exactly consistent with text, should be fixed.

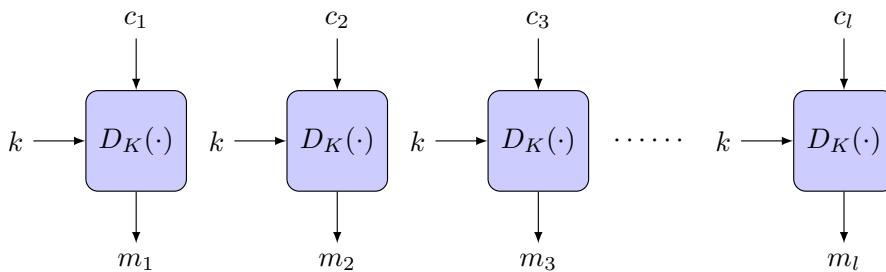


Figure 2.26: Electronic Code Book (ECB) mode decryption of ciphertext c consisting of l blocks, $c = c_1, \dots, c_l$. Adapted from [181].

The reader may have already noticed that ECB is simply a monoalphabetic substitution cipher, as discussed in subsection 2.1.3. The ‘alphabet’ here is indeed large: each ‘letter’ is a whole n -bit block. For typical block ciphers, the block size is significant, e.g., $n_{DES} = 64$ bits for DES and $n_{AES} = 128$ bits; this definitely improves security, and may make it challenging to decrypt ECB-mode messages in many scenarios.

However, obviously, this means that ECB may expose some information about plaintext, in particular, all encryptions of the same plaintext block will result in the same ciphertext block. Even with relatively long blocks of 64 or 128 bits, such repeating blocks are quite likely in practical applications and scenarios, since *inputs are not random strings*. Essentially, this is a generalization of the letter-frequency attack of subsection 2.1.3 (see Fig. 2.5).

This weakness of ECB is often illustrated graphically by the example illustrated in Fig. 2.27, using the ‘Linux Penguin’ image [121, 320].

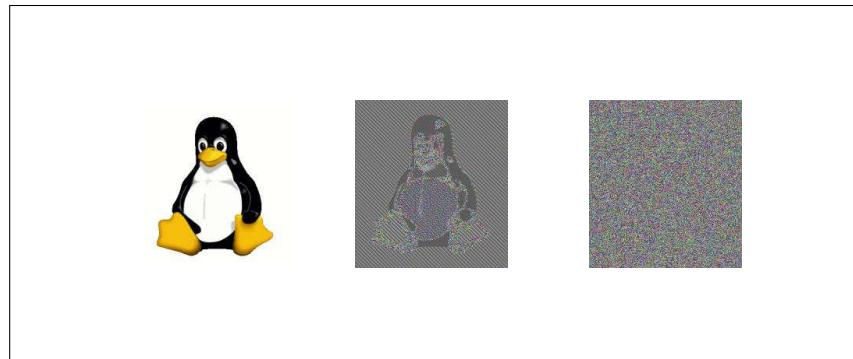


Figure 2.27: The classical visual demonstration of the weakness of the ECB mode. The middle and the right ‘boxes’ are encryptions of the bitmap image shown on the left. Which of the two is ‘encrypted’ using ECB, and which is encrypted with one of the secure encryption modes?

2.8.2 The Counter (CTR) mode and the Per-Block Random Mode (PBR)

We next present the Counter (CTR) mode and the Per-Block Random Mode (PBR).

Let us begin with the PBR mode. The PBR is not a standard, possibly since it is inefficient: a block of random bits is generated for each plaintext block, and sent as part of the ciphertext, resulting in ciphertext whose length is twice that of the plaintext. We present it since it provide a simple way to construct a secure stateless encryption scheme from a PRF, PRP or block cipher, and since it is very similar to the stateful CTR mode.

The PBR mode is illustrated in Figure 2.28. Let $m = m_1 \# m_2 \# \dots \# m_M$ be a plaintext message, where each m_i is one n -bits block, i.e., $m_i \in \{0, 1\}^n$, and let E denote a block cipher for n -bit blocks, and k denote a key for E . Then we compute the *PBR-mode encryption* of m using block cipher E and key k , denoted $PBR_Enc_k^E(m)$, as follows:

$$PBR_Enc_k^E(m) \equiv c_1 \# \dots \# c_M \text{ WHERE } \left\{ \begin{array}{l} r_i \xleftarrow{\$} \{0, 1\}^n \\ c_i \leftarrow (r_i, m_i \oplus E_k(r_i)) \end{array} \right\} \quad (2.47)$$

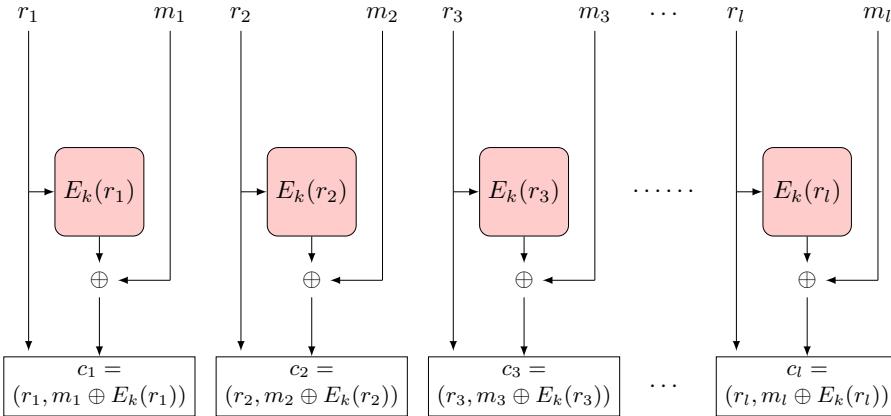


Figure 2.28: Per-Block Random (PBR) mode encryption of plaintext message \$m\$ consisting of \$l\$ blocks, \$m = m_1, \dots, m_l\$.

Namely, encrypt each message block \$m_i\$ with the corresponding random block \$r_i\$. Note that we can encode each \$c_i\$ simply as \$c_i = r_i \# m_i \oplus E_k(r_i)\$, i.e., as a string of \$2n\$ bits; the pairwise notation is equivalent, and a bit easier to work with.

PBR decryption performs the dual operation. Namely, given key \$k\$ and ciphertext \$c = (r_1, c'_1) + \dots + (r_n, c'_n)\$, where each \$r_i\$ and \$c'_i\$ are one block (\$n\$ bits), we compute the *PBR-mode decryption* of \$c\$, denoted \$PBR_Dec_k^E(c)\$, as:

$$PBR_Dec_k^E(c) = m_1 \# \dots \# m_M \text{ WHERE } m_i \leftarrow c_i \oplus E_k(r_i) \quad (2.48)$$

It is not difficult to show that PBR mode ensures correctness.

Exercise 2.21. Show that PBR mode ensures correctness, i.e., that for every \$l\$-blocks message \$m = m_1, \dots, m_l\$ and \$l\$ random blocks \$r_1, \dots, r_l\$ holds: \$m = PBR_Dec_k^E(PBR_Enc_k^E(m))\$.

Note that PBR mode does not use at all the ‘decryption’ function \$D\$ of the underlying block cipher (invertible PRP). Indeed, PBR can be instantiated using a PRF or PRP instead of using a block cipher. As can be seen from Table 2.5 and Exercise 2.47, this also holds for OFB and CFB modes.

PBR is *not* a standard mode, and rightfully so, since it is wasteful: it requires the use of one block of random bits per each block of the plaintext, and all these random blocks also become part of the ciphertext and are used for decryption, i.e., the length of the ciphertext is double the length of the plaintext. However, PBR is *secure* - allowing us to discuss a simple provably-secure construction of a symmetric cryptosystem, based on the security of the underlying block cipher (invertible PRP).

Theorem 2.1. If \$E\$ is a PRF or PRP, or \$(E, D)\$ is a block cipher (invertible PRP), then \$(PBR_Enc^E, PBR^D)\$ is a CPA-indistinguishable symmetric encryption.

Proof. We present the proof when E is a PRF; the other cases are similar. We also focus, for simplicity, on encryption of a single-block message, $m = m_1 \in \{0, 1\}^n$.

Denote by $(PBR\text{-}Enc^{\hat{E}}, PBR^{\hat{D}})$ the same construction, except using, instead of E , a ‘truly’ random function $f \xleftarrow{\$} \{0, 1\}^n \rightarrow \{0, 1\}^n$. In this case, for any pair of plaintext messages m_0, m_1 selected by the adversary and randomness r used for encrypting, the probability of $c^* = (r, r \oplus f(m_0))$ is exactly the same as the probability of $c^* = (r, r \oplus f(m_1))$, from symmetry of the random choice of f . Hence, the attacker’s success probability, when ‘playing’ the IND-CPA game (Def. 2.1) ‘against’ $(PBR\text{-}Enc^{\hat{E}}, PBR^{\hat{D}})$ is exactly half. Note that this holds even for computationally-unbounded adversary.

Assume, to the contrary, that there is some PPT adversary \mathcal{A} , that is able to gain a non-negligible advantage against $(PBR\text{-}Enc^E, PBR^D)$. Recall that this holds, even assuming E is a PRF - however, as argued above, \mathcal{A} succeeds with probability exactly half, i.e., with exactly zero advantage, against $(PBR\text{-}Enc^{\hat{E}}, PBR^{\hat{D}})$, i.e., if E was a truly random function.

We can use \mathcal{A} to distinguish between $E_k(\cdot)$ and a random function, with significant probability, contradicting the assumption that E_k is a PRF; see Def. 2.7. Namely, we run \mathcal{A} against the PBR construction instantiated with either a true random function or $E_k(\cdot)$, resulting in either $(\hat{E}^{PBR}, \hat{D}^{PBR})$ or (E^{PBR}, D^{PBR}) , correspondingly. Since \mathcal{A} wins with significant advantage against (E^{PBR}, D^{PBR}) , and with no advantage against $(\hat{E}^{PBR}, \hat{D}^{PBR})$, this allows distinguishing, proving the contradiction. \square

Error propagation, integrity and CCA security. Since PBR mode encrypts the plaintext by bitwise XOR, i.e., $c_i = (r_i + m_i \oplus E_k(r_i))$, flipping a bit in the second part result in flipping of the corresponding bit in the decrypted plaintext, with no other change in the plaintext. We say that such bit errors are *perfectly localized* or have no error propagation. On the other hand, bit errors in the random pad part r_i corrupt the entire corresponding plaintext block, i.e., are propagated to the entire block. In any case, errors are somewhat localized - other plaintext blocks are decrypted intact.

This property may seem useful, to limit the damage of such errors, but that value is very limited. On the other hand, this property has two negative security implications. The first is obvious: an attacker can flip specific bits in the plaintext, i.e., PBR provides no integrity protection. Of course, we did not require cryptosystems to ensure integrity; in particular, the situation is identical for other bitwise-XOR ciphers such as OTP.

The other security drawback is that PBR is not IND-CCA secure. This directly results from the error localization property. Since all the modes we show in this chapter localize errors (perfectly, or to a single or two blocks), it follows that *none* of these modes is IND-CCA secure.

Exercise 2.22 (Error localization conflicts with IND-CCA security). *Consider a cryptosystem E, D with localized errors. Specifically, if ciphertext consisting*

of three blocks, $c = c_1, c_2, c_3$ decrypts into plaintext m_1, m_2 , then for any c'_3 holds that c_1, c_2, c'_3 would decrypt into m_1, m'_2 for some m'_2 . Namely, corruption of (only) the third ciphertext block, does not corrupt the decryption of the first plaintext block. Show that such cryptosystem cannot be IND-CCA secure.

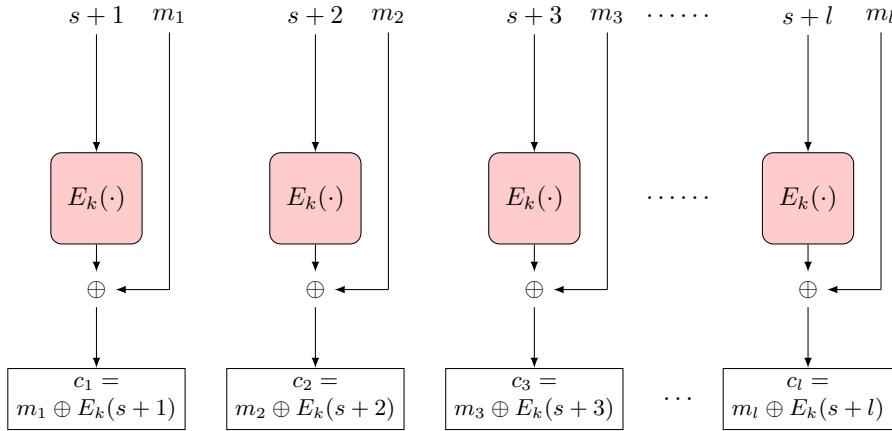


Figure 2.29: Counter (CTR) mode encryption of plaintext message m consisting of l blocks, $m = m_1, \dots, m_l$. The *counter* (state) s is the number of blocks encrypted so far. Initially, $s = 0$; after encrypting l blocks, add l to s , i.e., $s \leftarrow s + l$.

The Counter (CTR) mode. Let us now discuss the Counter (CTR) mode, a standard mode of operation, defined¹³ in [112], which is similar in design to the PBR mode. CTR mode is unique among the modes of operation we discuss in being *stateful*; it maintains a counter of the number of blocks encrypted/de-crypted so far, which is incremented whenever encrypting/decrypting a new block. See Figure 2.29.

The security of CTR mode follows, very similarly to that of PBR, from the PRF/PRP assumption of the block cipher E . By using state, we avoid the need to generate and send a new random block for each plaintext block; therefore, when we can reliably use state, CTR mode offers efficiency. Another advantage is that senders and recipients can pre-compute the block-cipher operations even before the receive the plaintext or ciphertext, requiring only block-wise XOR when the data (plaintext or ciphertext) arrives.

2.8.3 The Output-Feedback (OFB) Mode

We now proceed to discuss standard modes, which provably-ensure secure encryption, with randomization, for multiple-block messages - yet are more efficient compare to the PBR mode.

¹³Our description slightly simplifies CTR mode; see exact details in [112].

We begin with the simple Output-Feedback (OFB) Mode. In spite of its simplicity, this mode ensures provably-secure encryption - and requires the generation and exchange of only a *single block of random bits*, compare to one block of random bits per each plaintext block, as in PBR.

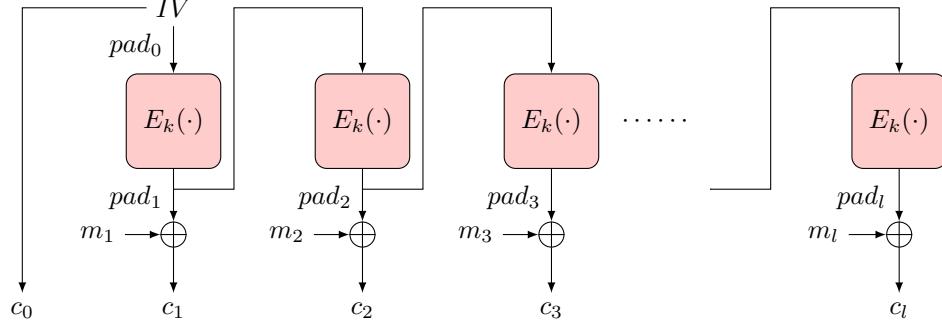


Figure 2.30: Output Feedback (OFB) mode encryption. Adapted from [181].

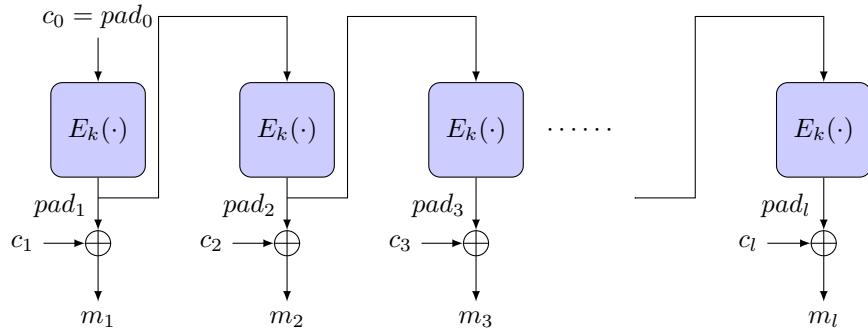


Figure 2.31: Output Feedback (OFB) mode decryption. Adapted from [181].

The OFB mode is illustrated in Figs. 2.30 (encryption) and 2.31 (decryption). OFB is a variant on the PRF-based stream cipher discussed in subsection 2.5.1 and illustrated in Fig. 2.17, and, like it, operates on input which consists of l blocks of n bits each. The difference is that OFB uses a PRP (block cipher) E_k instead of the PRF PRF_k .

We use a random *Initialization Vector (IV)* as a ‘seed’ to generate a long sequence of pseudo-random n -bit *pad blocks*, pad_1, \dots, pad_l , to encrypt plaintext blocks m_1, \dots, m_l . We next compute the bitwise XOR of the pad blocks pad_1, \dots, pad_l , with the corresponding plaintext blocks m_1, \dots, m_l , resulting in the ciphertext which consists of the random IV c_0 and the results of the XOR operation, i.e. $c_1 = m_1 \oplus pad_1$, $c_2 = m_2 \oplus pad_2, \dots$

Let us now define $OFB - E_k(m)$, the OFB mode for a given block cipher (E, D) . For simplicity we define $OFB - E_k(m)$ for messages m which consist of some number l of n -bit blocks, i.e., $m = m_1 \# \dots \# m_l$, where $(\forall i \leq l) |m_i| = n$.

Then $OFB - E_k(m)$ is defined as:

$$OFB_k^E(m_1 + \dots + m_l) = (c_0 + c_1 + \dots + c_l) \quad (2.49)$$

where:

$$pad_0 \xleftarrow{\$} \{0, 1\}^n \quad (2.50)$$

$$pad_i \leftarrow E_k(pad_{i-1}) \quad (2.51)$$

$$c_0 \leftarrow pad_0 \quad (2.52)$$

$$c_i \leftarrow pad_i \oplus m_i \quad (2.53)$$

Offline pad precomputation The OFB mode allows both the encryption process and the decryption process to precompute the pad, ‘offline’ - i.e., before the plaintext and ciphertext, respectively, are available. Offline pad precomputation is possible since the pad does not depend on the plaintext (or ciphertext). This can be important, e.g., when a CPU with limited computation speed needs to support a limited number of ‘short bursts’, without adding latency. Once the plaintext/ciphertext is available, we only need one XOR operation per block.

Parallelism. The pad is computed sequentially; there does not appear to be a way to speed up its computation using parallelism.

Error localization, correction and integrity Another important property for encryption schemes is *error localization*, namely, the number of bits changes in the deciphered plaintext, as result of corruption of a single ciphertext bit. Since OFB operates as a bit-wise stream cipher, then it is 1-localized (or perfectly localized): a change in any ciphertext bit simply causes a change in the corresponding plaintext bit - and no other bit.

The ‘perfect bit error localization’ property implies that *error correction and detection* code work can be applied to the plaintext (before encryption, with correction applied to plaintext after decryption) or applied to the ciphertext. Without localization, a single bit error in the ciphertext could translate to many bit errors in the plaintext. This implies that error correction, and, to lesser degree, detection, should be applied to the ciphertext.

Encode-then-Encrypt considered harmful. Some designers prefer to apply error correction to the plaintext, and rely on the ‘perfect bit error localization’ property to allow recovery from corresponding errors in the ciphertext. This motivated the use of OFB or a similar XOR-based stream cipher, allowing application of *error detection code (EDC)* or *error correction code (ECC)* on the plaintext; we refer to this as the *Encode-then-Encrypt* design. We now explain why this design could cause vulnerability (hence ‘considered harmful’).

Our discussion of Encode-the-Encrypt applies to both EDC and ECC; let us focus on ECC. ECC codes have two functions, $encode(\cdot)$ and $decode(\cdot)$; the

input domain to *encode*, denoted Dom , may be fixed-length strings (*block code*) or variable-length strings (*convolution code*). All ECC codes must satisfy the basic correctness property, which is that for every input string $m \in \text{Dom}$ holds: $m = \text{decode}(\text{encode}(m))$.

An ECC code should ensure *noise correction* property, i.e., it should ensure recovery from some set of possible errors in encoded messages; often, an ECC also allows detection of some additional errors. For example, one classical noise model are Hamming errors, which are simply bit-flips; these errors are defined using the *Hamming distance*. The Hamming distance $H(x, y)$ between two equal-length binary strings ($|x| = |y|$ and $x, y \in \{0, 1\}^*$) is the number of different bits, i.e.:

$$(\forall l \in \mathbb{N}, x, y \in \{0, 1\}^l) \quad H(x, y) \equiv |\{i : x[i] \neq y[i]\}| \quad (2.54)$$

An ECC ensures correction of t_c -bit-flip errors, or corrects errors up to Hamming distance t_c , if for every message $m \in \text{Dom}$ and every binary string $y \in \{0, 1\}^*$:

$$H(y, \text{code}(m)) \leq t_c \Rightarrow m = \text{decode}(y) \quad (2.55)$$

Similarly, an ECC (or EDC) ensures detection of t_d -bit-flip errors, or detects errors up to Hamming distance t_d , if for every message $m \in \text{Dom}$ and every binary string $y \in \{0, 1\}^*$:

$$H(y, \text{code}(m)) \leq t_d \Rightarrow (\text{decode}(y) \in \{m, \text{FALSE}\}) \quad (2.56)$$

The classical *Hamming code* allows correction of a single bit-flip error, and detection of two bit-flip errors, i.e., $t_c = 1$ and $t_d = 2$.

However, applying error correction or detection codes on the plaintext, and relying on perfect bit error localization to allow error detection/correctness of the ciphertext, is not recommended. The reason is that such codes create structured redundancy in the plaintext, which may facilitate *CTO attacks*. Let us give an example.

Example 2.8 (CTO attack on GSM). *This is an important vulnerability: a Ciphertext-Only (CTO) attack on the A5/1 and A5/2 stream ciphers [20], which are defined as part of the GSM protocol. This attack exploits the known relationship between ciphertext bits, due to the fact that an Error Correction Code was applied to the plaintext. This redundancy suffices to attack the ciphers, using techniques that normally can be applied only in Known Plaintext attacks. Unfortunately, complete details of this beautiful and important result are beyond our scope; for details, see [20]. As a result of this attack, the use of the (weaker) A5/2 was completely discontinued, and the use of A5/1 is not recommended. However, the GSM protocol still applied Encode-then-Encrypt, facilitating CTO cryptanalysis attacks.*

One may wonder, why would designers prefer to apply error correction to the plaintext rather than to the ciphertext? One motivation may be the hope that this may make cryptanalysis harder, e.g., corrupt some plaintext statistics such

as letter frequencies. This may hold for some codes; but we better design such defenses explicitly into the cryptosystem and not rely on such fuzzy properties of encoding.

Another motivation may be the hope that applying error correction/detection to the plaintext may provide integrity. Note that due to the perfect bit error localization of OFB, an attacker can easily flip a specific plaintext bit - by flipping the corresponding ciphertext bit. If we applied error detection to the plaintext, then corruption of a single bit will corrupt the entire plaintext. However, since the attacker can flip multiple ciphertext bits, thereby flipping the corresponding plaintext bits, there are cases where the attacker can modify the ciphertext in such a way as to flip specific bits in the plaintext while also ‘fixing’ the error detection/correction code, to make the message appear correct. We conclude the following principle.

Principle 9 (Minimize plaintext redundancy). *Plaintext should preferably have minimal redundancy. In particular, plaintext should preferably not contain Error Correction or Detection codes.*

Namely, applying error correction to plaintext is a bad idea - certainly when using stream-cipher design such as OFB. This raises the obvious question: can an encryption mode of a block cipher also protect the *integrity* of the decrypted plaintext? Both of the following modes, CFB and CBC, provide some defense of integrity - by *ensuring errors do propagate*.

Provable security of OFB. The above discussed weaknesses are due to incorrect *deployments* of OFB; correctly used, OFB is secure. Proving the security of OFB follows along similar lines to Theorem 2.1, except that in order to deal with multi-block messages, we will need to use a more elaborate proof technique called ‘hybrid proof’; we leave that for courses and books focusing on Cryptology, e.g., [141, 303].

2.8.4 The Cipher Feedback (CFB) Mode

We now present the Cipher Feedback (CFB) Mode. Like most standard modes, it uses a random first block (‘initialization vector’, IV). In fact, CFB resembles OFB; the IV is also the first block of the ciphertext ($c_0 = IV$). Then, iteratively, each ciphertext block c_i is used to generate the following pseudo-random pad block $pad_{i+1} = E_k(c_i)$; note that there is no pad_0 (as c_0 is simply the IV). Finally, the next ciphertext block, c_{i+1} , is computed by bitwise XOR between the corresponding pseudorandom pad block pad_{i+1} and the corresponding plaintext block m_{i+1} .

Namely, we define $CFB - E_k(m)$, the CFB mode for a given block cipher (E, D) , as follows. For simplicity we define $CFB - E_k(m)$ for messages m which consist of some number l of n -bit blocks, i.e., $m = m_1 + \dots + m_l$, where $(\forall i \leq l)|m_i| = n$. The ciphertext $CFB_k^E(m)$ consists of $l + 1$ blocks

$c_0 \parallel c_1 \parallel \dots \parallel c_l$, defined by:

$$CFB_k^E(m) \equiv c_0 \parallel c_1 \parallel \dots \parallel c_l, \text{ where:}$$

$$\begin{aligned} c_0 &= IV \xleftarrow{\$} \{0,1\}^n \\ c_i &= m_i \oplus E_k(c_{i-1}) \quad (\text{for } i \in \{1, \dots, l\}) \end{aligned} \quad (2.57)$$

Note that the difference between CFB and OFB is in the ‘feedback’ mechanism, namely, the computation of the pads pad_i (for $i > 1$). In CFB mode, this is done using the ciphertext rather than the previous pad. See Fig. 2.32.

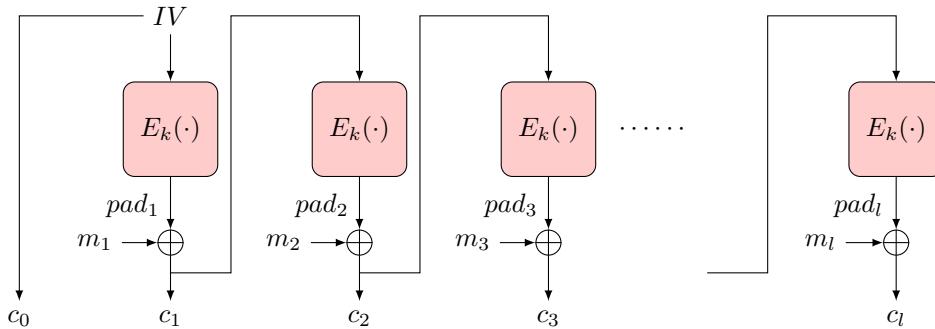


Figure 2.32: Cipher Feedback (CFB) mode encryption. Adapted from [181].

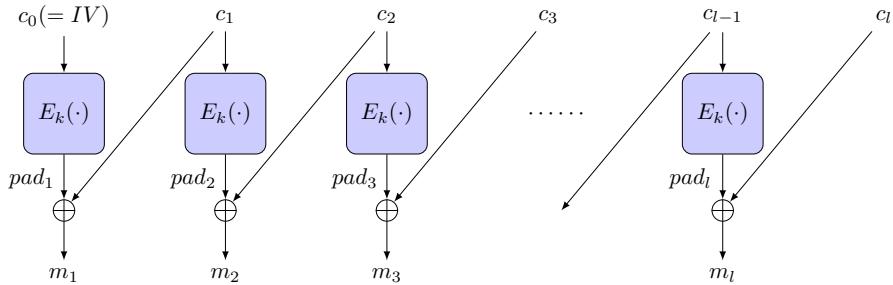


Figure 2.33: Cipher Feedback (CFB) mode decryption. Adapted from [181].

Optimizing implementations: parallel decryption, but no precomputation Unlike OFB, the CFB mode does not support offline precomputation of the pad, since the pad depends on the ciphertext (of the previous block).

One optimization that is possible is to parallelize the decryption operation. Namely, decryption may be performed for all blocks in parallel, since the decryption m_i of block i is $m_i = c_i \oplus p_i = c_i \oplus E_k(c_{i-1})$, i.e., can be computed based on the ciphertexts of this block and of the previous block.

Error localization and integrity Error localization in CFB is not perfect; a single bit error in one ciphertext block completely corrupts the following plaintext block.

As we discussed for OFB, this reduction in error localization may be viewed as an advantage in ensuring integrity. Like OFB mode, the CFB mode allows the attacker to flip specific bits in the decrypted plaintext, by flipping corresponding bits in the ciphertext. However, as a result of such bit flipping, say in block i , the decrypted plaintext of the following block is completely corrupted. Intuitively, this implies that applying an error-detection code to the plaintext would allow detection of such changes, in contrast to the situation with OFB mode.

However, this dependency on the error detection code applied to the plaintext may cause some concerns. First, it is an assumption about the way that OFB is used; can we provide some defense for integrity that will not depend on such additional mechanisms as an error detection code? Second, it seems challenging to prove that the above intuition is really correct, and this is likely to depend on the specifics of the error detection code used. Finally, adding error detection code to the plaintext increases its redundancy, in contradiction to Principle 9. We next present the CBC mode, which provides a different defense for integrity, which addresses these concerns.

2.8.5 The Cipher-Block Chaining (CBC) mode

Among the modes of operation defined in [112], the most widely-used, by far, is the Cipher-Block Chaining (CBC) mode.

The CBC mode, like the OFB and CFB modes, uses a random Initialization Vector (IV) as the first block of the ciphertext, $c_0 \xleftarrow{\$} \{0,1\}^n$. However, in contrast to OFB and CFB, to encrypt the i^{th} plaintext block m_i , CBC XOREs m_i with the previous ciphertext block c_{i-1} , and *then* applies the block cipher. Namely, $c_i = E_k(c_{i-1} \oplus m_i)$. See Fig. 2.34.

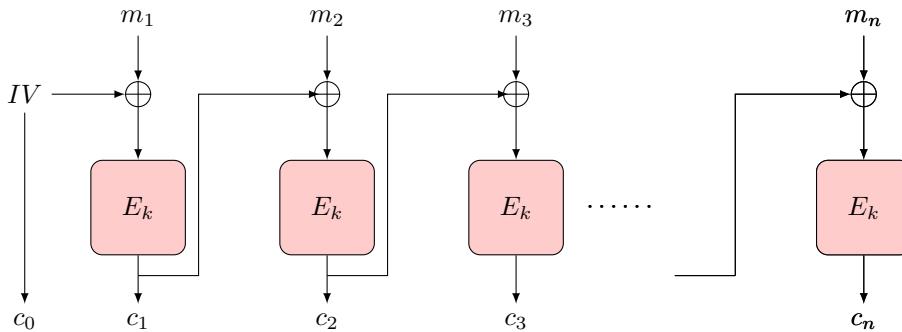


Figure 2.34: Cipher Block Chaining (CBC) mode encryption. Adapted from [181].

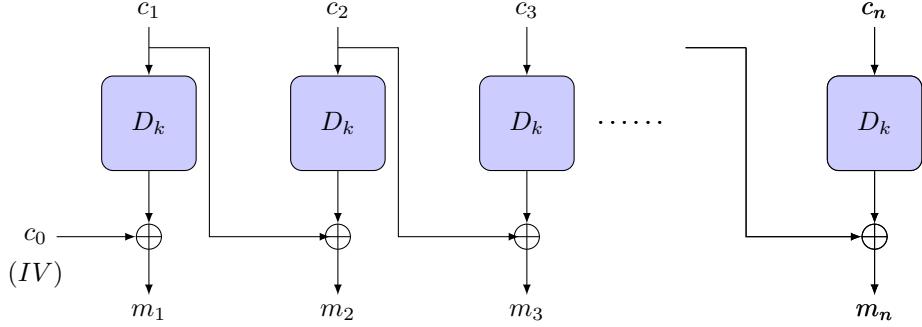


Figure 2.35: Cipher Block Chaining (CBC) mode decryption. Adapted from [181].

More precisely, let (E, D) be a block cipher, and let $m = m_1 \# \dots \# m_n$ be a message (broken into blocks). Then the CBC encryption of m using key k and initialization vector $IV \in \{0, 1\}^l$ is defined as:

$$CBC_k^E(m_1 \# \dots \# m_l) = (c_0 \# c_1 \# \dots \# c_l) \quad (2.58)$$

where:

$$c_0 = IV \xleftarrow{\$} \{0, 1\}^n \quad (2.59)$$

$$(i \in \{1, \dots, l\}) c_i \leftarrow E_k(c_{i-1} \oplus m_i) \quad (2.60)$$

We see that the CBC mode, like CFB, allows parallel decryption, but not offline pad precomputation.

The CBC mode, like the other modes (except ECB), ensures IND-CPA, i.e., security against CPA attacks, provided that the underlying block cipher is a secure invertible PRP. However, it is not secure against CCA attacks.

Exercise 2.23. Demonstrate that CBC mode does not ensure security against CCA attacks.

Hint: the solution is quite similar to that of Exercise 2.22. □

Incorrect use vulnerability: BEAST exploit of predictable IV. While CBC mode ensures security in the sense of IND-CPA, i.e., against Chosen Plaintext Attack, this is only true if CBC is used correctly; in particular, the IV (also used as c_0) must be random. The SSL protocol, as well as version 1.0 of TLS, use CBC encryption in the following incorrect way. They select the IV randomly only for the first message m_0 in a connection; for subsequent messages, say m_i , the IV is simply the last ciphertext block of the previous message. This creates a vulnerability exploited, e.g., by the *BEAST attack*. For details, see subsection 7.2.4, Exercise 7.8 and [19, 110].

Error propagation and integrity Any change in the CBC ciphertext, even of one bit, results in unpredictable output from the block cipher's 'decryption' operation, and hence unpredictable decryption. Namely, flipping a bit in the ciphertext block i does not flip the corresponding bit in plaintext block i , as it did in the OFB and CFB modes.

However, the flipping of a bit in the ciphertext block c_{i-1} , without change to block c_i , results in the flipping of the corresponding bit in the i^{th} decrypted plaintext block. Namely, bit-flipping is still possible in CBC, it is just a bit different - and in order to flip a bit in the decrypted-plaintext block i , the adversary has to flip the corresponding bit in the *previous* block ($i - 1$), which results in corruption of the decryption of block $i - 1$. Indeed, this kind of tampering is used in several attacks on systems deploying CBC, such as the Poodle attack [238]. Note also that bit flipping in the first decrypted-plaintext block only requires flipping of the corresponding IV block - and hence does not corrupt any plaintext block.

2.8.6 Modes of Operation: Ensuring CCA Security?

We already observed, in Exercise 2.22, that any cryptosystem (and mode) that ensures error localization to some extent cannot be IND-CCA secure. This implies that none of the modes we discussed is IND-CCA secure. Such failure can occur even for the much weaker - and more common - case of Feedback-only CCA attacks, where the attacker does not receive the decrypted plaintext, but only an indication of whether the plaintext was 'valid' or not.

How can we ensure security against CCA attacks? One intuitive defense is to avoid giving any feedback on invalid-plaintext failures. However, this is harder than it may seem. For example, often, after (successful) decryption, a response is immediately sent, which may be hard to emulate when the plaintext is invalid - we may be even unable to identify the sender, e.g., if the sender identity is encrypted for anonymity. By observing if a response is sent, or the timing of the response, an attacker may obtain feedback on the attack. Such unintentional indications are referred to as *side channels*; for example, when the feedback is based on the time the response is sent, this is a *timing side channel*.

A better approach may be to *prevent response to chosen-ciphertext queries*, **without** decrypting them. One simple way to do this is to *authenticate* the ciphertext, typically by appending to the ciphertext an *authentication tag*, which allows secure detection of any modification in the ciphertext. Several of the more modern, widely used modes of operation, e.g., GCM [113, 285], combine authentication and encryption, with one benefit being the protection against chosen-ciphertext attacks. Authentication is the subject of the next chapter.

2.9 Padding Schemes and Padding Oracle Attacks

All modes of operation are defined for input whose length is an integral number l of blocks. In most applications, the input may not be an integral number of blocks, but an string of arbitrary number of bits or, more commonly, of bytes.

The principle of all padding schemes is quite simple. Before encryption, the plaintext is *padded* to an integral number of blocks, by appending a *pad* string to the message, which is removed after decryption. The length of the pad is between one byte and a whole block (l bytes), and is chosen to ensure that the padded message (message plus pad) fits in an integral number of blocks.

Padding schemes mainly differ in the *contents* of the pad string, and in the validation of the pad after decryption. The two most commonly used padding schemes applied to plaintext before shared-key encryption are:

X9.23 padding: Several protocols, most notably the SSL protocol discussed in Chapter 7, use the following padding scheme, which we refer to as *X9.23 padding*, since it was defined in the ANSI X9.23 standard [10]. In X9.23 padding, the last byte of the pad contains the length of the pad minus one, i.e., the length except this byte. The length of the pad is restricted¹⁴ to the block-length l , hence the value of the last byte must be a number between zero and $l - 1$. If the results of decryption does not end with a byte between zero and $l - 1$, then the ciphertext is considered to have invalid padding, and a padding error is returned. The other bytes of the X9.23 pad can have arbitrary values (e.g., all zeros, or random values).

PKCS#5 padding: Other protocols, most notably the TLS protocol (also discussed in Chapter 7), use the following padding scheme, which we refer to as *PKCS#5 padding*. PKCS#5 padding is defined in several standards including PKCS#5, PKCS#7 and RFC 5652 [172]. It is similar to ANSI X9.23 padding, the main difference being that *all padding bytes* must contain the same value as the last byte, i.e., the length of the pad minus one. Namely, a one-byte pad will contain the byte 01, a two-byte pad will contain pad 0202, and so on. If the results of decryption does not have this pattern, then a padding error is returned. Another difference is that PKCS#5 padding allows the pad to be longer than a single block (up to 256 bytes, to ensure that the pad length minus one can fit in one byte.).

With both padding schemes, the decrypted plaintext should end with a valid pad, which should be (efficiently) verified by the recipient. We say that ciphertext c has *invalid pad*, and return a padding error, if the decrypted plaintext has invalid pad.

Consider an l -bytes block m , or a multi-block message whose last block is m . Let $m[i]$, for $i = 1, \dots, l$ denote the i^{th} most-significant byte of m . Block m has *valid X9.23 padding* if $m[l] < l$. For example, for blocks of $l = 16$ bytes, the four most-significant bits of the last byte must be all zeroes, i.e., the last byte must be of the form $0x0\phi$ in hexadecimal notation. In this case, ϕ can be any hexadecimal digit, from 0 to F , representing the corresponding four bits in binary.

¹⁴For convenience, we consider this restriction of pad length to one block to be a mandatory property of X9.23 padding, although some implementations may not enforce it.

Similarly, plaintext string m has *valid PKCS#5 padding* if the value of the last byte of m , which we denote $x = m[|m|]$, is also the value of preceding $x - 1$ bytes of m , i.e., $m[(|m| - x) : |m|] = x^x$. Namely, *the last x bytes of m contain the same value x .*

The reader may wonder why we bother describing these two simple and very similar schemes. The reason is that padding error indications, which we refer to as *padding oracles*, are exploited in many attacks. In Chapter 7, we discuss practical padding oracle attacks against SSL/TLS. In this section, we describe the *Padding Oracle Attack model*, and then present a simple padding oracle attack against ECB and CBC modes using X9.23 padding, extended in Exercise 2.24 to an attack against CBC mode using PKCS#5 padding.

The Padding Oracle Attack Model. In many systems, an attacker may be able to detect when an message with invalid padding is received. We refer to the indication of valid/invalid padding over the (decrypted) plaintext as a *padding oracle*. The padding oracle is sometimes provided by an explicit ‘invalid pad’ error message, and sometimes inferred by detecting the different behavior of the recipient due to invalid pad, such as different timing of a response; the later is a special case of a *timing side channel*.

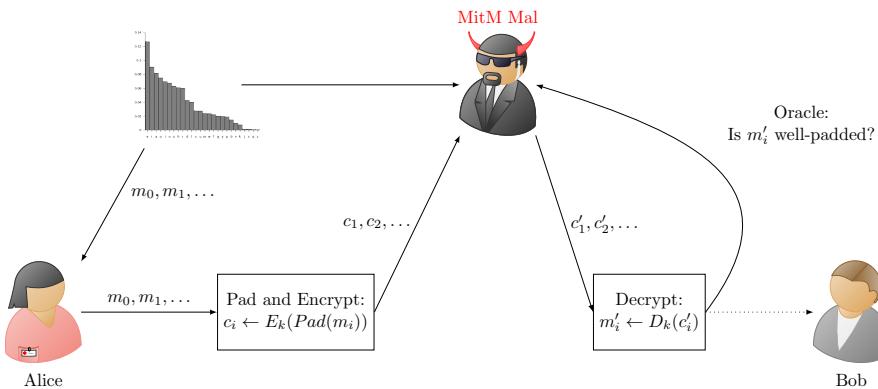


Figure 2.36: The Padding Oracle Attack model.

We illustrate the basic Padding Oracle Attack model in Figure 2.36; this is basically a CTO attack with the addition of the *padding oracle* capability.

Of course, the padding oracle capability may also complement other attacker capabilities. In particular, in subsection 7.2.2, we present the CPA-Oracle Attack model, where the attacker also has chosen-plaintext capability. The CPA-Oracle Attack model is quite realistic, although it is more powerful than the Padding Oracle Attack model. Indeed, the CPA-Oracle Attack model is often used for security evaluation of practical protocols. In particular, in subsection 7.2.2 we discuss practical attacks against different versions of the SSL/TLS protocol, using the CPA-Oracle Attack model.

Simple padding-oracle attacks. Let us present simple padding oracle attacks against the ECB and CBC modes of operation, when using X9.23 padding. Assume, for example, that we use blocks of 16 bytes. Hence, the length of the pad is between one and 16, and the encoded value in the last byte should be $0x0\phi$ in hexadecimal notation (where ϕ is one hexadecimal digit). Typical encryption schemes use shorter blocks (8 bytes for DES and 12 bytes for AES), which requires a tiny change in the attack and slightly increases the exposure due to the attack.

Padding-Oracle attack on X9.23 padding using ECB mode. Consider plaintext message $m = m_1 + \dots + m_n$, containing $n - 1$ full blocks and one empty or non-full block m_n . In ECB mode, each non-final plaintext block m_i (i.e., $i < n$), is encrypted directly as $c_i = E_k(m_i)$. The final plaintext block m_n is padded before encryption, i.e., encrypted as $c_n = E_k(\text{pad}(m_n))$, where $\text{pad}(\cdot)$ is the X9.23 padding function.

The attack can be applied to any block of the ciphertext, except the last block; let us focus on some block c_i where $i < n$. Instead of sending c_i as an intermediate block of a longer ciphertext message, the attacker sends c_i as if it is the *last* block of a ciphertext message, e.g., the ciphertext consist only of this single block c_i . (If the ciphertext should have multiple blocks, prepend some blocks before c_i - we only need c_i to be the last block of the ciphertext.)

Following the Padding Oracle Attack model, the attacker receives an indication whether the decryption of c_i , i.e., $m_i = D_k(c_i)$, has valid padding or not. If m_i has valid padding, then the value of its last byte must be less than the block length, $l = 16$; namely, the last byte of m must be of the form $0x0\phi$ (in hexadecimal notation), i.e., its four most significant bits must be zeros. This is an exposure of (limited) information about these four bits of the plaintext, i.e., an indication if these four bits are all zeros or not.

Padding-Oracle attack on X.923 padding using CBC mode. This attack, like the one on ECB mode, is applicable to every non-last ciphertext block c_i . Again, the basic idea is to send c_i as the *last* block of ciphertext messages to the recipient, and learn information about the value of few bits of plaintext m_i , using the response of the padding oracle.

Specifically, let $m_i[j]$ denote the j^{th} bit of m_i . Recall that we use l to denote the number of bits in each block, i.e., $m_i = m_i[1] + \dots + m_i[l]$. We show how, when using X9.23 padding, the attack finds the four plaintext bits $m_i[j]$ for $j = l - 7, l - 6, l - 5$ and $l - 4$. This requires only access to c_i and to the padding oracle for CBC mode. Exercise 2.24 extends the attack to PKCS#5 padding, where the attack finds the entire last byte of m_i .

Recall that in CBC mode, the ciphertext block c_i is computed as $c_i = E_k(c_{i-1} \oplus m_i)$. The attacker now sends ciphertexts c' containing i blocks, where $c'_i = c_i$, and using different previous ciphertext blocks c'_{i-1} . The value of c_{i-1} may be different from the value of c_{i-1} , the original previous ciphertext block. (The other blocks of c' do not have any impact on the attack and can even be eliminated.)

Let m'_i denote the last decrypted plaintext block would be m'_i ; since we use

CBC, it is computed as:

$$m'_i = D_k(c_i) \oplus c'_{i-1} = (m_i \oplus c_{i-1}) \oplus c'_{i-1} \quad (2.61)$$

The attacker has (only) access to the padding oracle, which indicates if the last block of the decrypted message, m'_i , has correct padding or not. This depends only on the value of the last byte of m'_i ; correct padding requires that last byte of m'_i be of the form $0x0\phi$, i.e., its four most significant bits should be all zeros. In other words, $m'_i[l-7:l-4] = 0000$ (in binary).

Recall that $m'_i = D_k(c_i) \oplus c'_{i-1}$; therefore, the attacker can try the 16 different values of four most significant bits of the last byte of c'_{i-1} , until it finds a value c'_{i-1} that has correct padding, i.e., resulting in plaintext m'_i whose last byte is of the form $0x0\phi$, i.e., $m'_i[l-7:l-4] = 0000$. Hence, from Equation 2.61, we have:

$$m_i[l-7:l-4] = c_{i-1}[l-7:l-4] \oplus c'_{i-1}[l-7:l-4] \quad (2.62)$$

The reader may have noticed that the attack exploits the fact that X9.23 padding limits the pad length to one block. PKCS#5 padding does not make this restriction, which may seem to make it secure against such padding oracle attack. However, in fact, Exercise 2.24 extends the attack for the case of PKCS#5 padding. The extended attack requires more padding-oracle queries, but is also more rewarding, as it allows *recovery of the entire plaintext*.

Exercise 2.24. *Present a padding attack, assuming the availability of a padding oracle and the use PKCS#5 padding. The attack should find the entire last byte of a non-last plaintext block m_i , given only the corresponding ciphertext block c_i and previous ciphertext block c_{i-1} , and access to the padding oracle.*

Hint: A random plaintext block would have valid padding if its last byte contains $0x00$, i.e., the entire pad is this single last byte; other random plaintext blocks are unlikely to have valid padding (why?). Use this to find the last byte of m_i . Once this byte is found, we repeat a similar logic to find the preceding byte of m_i , using the fact that a valid random plaintext whose last byte contains $0x01$, would also have $0x01$ in the preceding byte. We then proceed similarly to find all plaintext bytes of m_i . \square

The reader may also find the solution to this exercise by reading [312]. Additional padding oracle attacks, focusing on the SSL/TLS protocols, are described in subsection 7.2.3.

2.10 Case study: the (in)security of WEP

We conclude this chapter, and further motivate the next, by discussing a case study: vulnerabilities of the *Wired Equivalency Privacy (WEP)* standard [85]. WEP stands for *Wired Equivalency Privacy*; it was developed as part of the IEEE 802.11b standard, to provide some protection of data over wireless local area networks (also known as WiFi networks). As the name implies, the original

goals aimed at a limited level of privacy (meaning confidentiality), which was deemed ‘equivalent’ to the (physically limited) security offered by a wired connection.

These critical vulnerabilities were discovered long ago, mostly in [66], relatively soon after the standard was published; yet, products and networks supporting WEP still exist. This is an example of the fact that once a standard is published and adopted, it is often very difficult to fix security. Hence, it is important to carefully evaluate security in advance, in an open process that encourages researchers to find vulnerabilities, and, where possible, with proofs of security. To address these vulnerabilities, WEP was replaced - possibly in too much haste - with a new standard, the Wi-Fi Protected Access (WPA), which so far has three versions (WPA1, WPA2, WPA3). Vulnerabilities were also found in these, e.g., see [310, 311], but these are more subtle and harder to exploit; WEP should only be used for educational purposes, as we do here.

WEP assumes a symmetric key between the mobile device and an *access point*, which is used as the key (seed) for the RC4 cipher. WEP networks share the same key with all mobiles; this means that *each device which has the key, can eavesdrop on all communication*; this vulnerability exists also for the common use of more advanced WiFi security protocols, such as WPA (versions 1 to 3). We discuss specific additional vulnerabilities which are specific to WEP, and make it insecure even against an attacker that is not given the key to connect to the network.

Confidentiality in WEP is protected using the RC4 PRG, used as a stream cipher as in subsection 2.5.1. RC4 is initiated with a secret shared key, which, in WEP, is specified to be only 40 bits long. This short key size was chosen intentionally, to allow export of the hardware, since when the standard was drafted, the United States and many other countries had export limitations on strong cryptography, which necessarily uses longer keys. Many WEP implementations also support longer, 104-bit keys for RC4; however, attacks published show that even with 104-bit keys, RC4 is still vulnerable, see subsection 2.5.6, subsection 7.2.5 and [9, 194, 224].

The WEP PRG is initiated with a 24-bit per-packet random *Initialization Vector (IV)*. We use $RC4_{IV,k}$ to denote the string output by RC4 when initialized using a given IV, k pair. More specifically, we use $RC4_{IV,k}[l]$ to denote the first l bits in $RC4_{IV,k}$, i.e., in the output by RC4 when initialized using given IV, k pair.

WEP packets use the *CRC-32* error detection code, computed over the plaintext message m . CRC-32 [198] is one of the standard variants of *CRC cyclic redundancy check* code. CRC codes are popular *error-detecting codes (EDC)*; they are simple to implement and efficiently detect errors in data, caused by random noise corruptions. Namely, if m' is the result of such random corruption of m , then, with high probability, m and m' will have different CRC codes, i.e., $CRC(m) \neq CRC(m')$, allowing detection of the corruption by comparing their CRC codes. Note that, for simplicity, and since we do not discuss other CRC codes, we use $CRC(m)$ to denote the CRC-32 code computed over message m .

CRC codes, and in particular CRC-32, are *linear*, in the sense that for any two strings $m, m' \in \{0, 1\}^*$ of equal length ($|m| = |m'|$), holds:

$$\text{CRC}(m \oplus m') = \text{CRC}(m) \oplus \text{CRC}(m') \quad (\text{if } |m| = |m'|) \quad (2.63)$$

WEP uses CRC as follows. To send a message m using secret key k , WEP implementations select a random 24-bit IV, and transmit the IV together with $\text{WEP}_k(m, \text{IV})$, defined as:

$$\text{WEP}_k(m, \text{IV}) \equiv \text{RC4}_{\text{IV}, k}[32 + |m|] \oplus (m \# \text{CRC}(m)) \quad (2.64)$$

The length of the WEP transmission is, therefore, the length of the message m , plus 56 bits: 24 bits for the IV and 32 bits for the CRC-32 code.

2.10.1 CRC-then-XOR does not ensure integrity

CRC-32 is a quite good error detection code. By encrypting the output of CRC, specifically by XORing it with the pseudo-random pad generated by RC4, the WEP designers hoped to protect message integrity, i.e., not only detect random corruptions, but also prevent intentional modification or forgery of messages.

However, *error-detection codes are designed to detect random, not intentional, corruptions; i.e., for reliability, not for security*. In particular, it is easy to find a *collision*, i.e., messages m and m' such that $m \neq m'$ yet $\text{CRC}(m) = \text{CRC}(m')$. Furthermore, we next show how the linearity of the CRC (Equation 2.63) allows an attacker to change the message m sent in a WEP packet, by flipping any desired bits and appropriately adjusting the CRC field.

Specifically, let Δ represent the string of length $|m|$ containing 1 for bit locations that the attacker wishes to flip. Having eavesdropped and obtained $\text{WEP}_k(m, \text{IV})$, the attacker can compute a *valid* $\text{WEP}_k(m \oplus \Delta, \text{IV})$ as follows:

$$\begin{aligned} \text{WEP}_k(m \oplus \Delta, \text{IV}) &= \text{RC4}_{\text{IV}, k}[32 + |m \oplus \Delta|] \oplus ([m \oplus \Delta] \# \text{CRC}(m \oplus \Delta)) \\ &= \text{RC4}_{\text{IV}, k}[32 + |m|] \oplus (([m \oplus \Delta] \# [\text{CRC}(m) \oplus \text{CRC}(\Delta)]) \\ &= \text{RC4}_{\text{IV}, k}[32 + |m|] \oplus (m \# \text{CRC}(m)) \oplus (\Delta \# \text{CRC}(\Delta)) \\ &= \text{WEP}_k(m, \text{IV}) \oplus (\Delta \# \text{CRC}(\Delta)) \end{aligned}$$

Namely, the CRC mechanism, XOR-encrypted, *does not provide any meaningful integrity protection*. An attacker can easily flip bits in a WEP message - and have it properly received.

WEP authentication-based vulnerabilities. We have just seen that WEP failed to provide integrity; however, as we now show, *WEP also fails to ensure confidentiality*. Interestingly, the most devastating vulnerability, which is the one we show, takes advantage of WEP's *shared-key authentication* mode; WEP also defines a mode called *open-system authentication*, which simply means that there is no authentication, and is therefore not vulnerable to this specific attack. Even when using open-system authentication, i.e., giving up on authentication,

WEP is still vulnerable to other cryptanalysis attacks exploiting RC4 weaknesses, e.g., see [224].

However, we focus on the vulnerability of WEP when using the *shared-key authentication* mode. It works very simply: the access point sends a random challenge R ; and the mobile sends back $WEP_k(R, IV)$, i.e., a proper WEP packet containing the ‘message’ R .

This authentication mode is currently rarely used, since it allows attacks on the encryption mechanism. First, notice that it provides a trivial way for the attacker to obtain ‘cribs’ (known plaintext - ciphertext pairs). Of course, encryption systems *should* be protected against known-plaintext attacks; however, following the *conservative design principle* (principle 3), system designers should try to make it difficult for attackers to obtain cribs. In the common, standard case of 40-bit WEP implementations, a crib is deadly - an attacker can now do a trivial exhaustive search to find the key.

Even when using longer keys (104 bits), the shared-key authentication exposes WEP to a simple cryptanalysis attack. Specifically, since R is known, the attacker learns $RC4_{IV,k}$ for a given, random IV . Since the length of the IV is just 24 bits, it is feasible to obtain a collection of most IV values and the corresponding $RC4_{IV,k}$ pads, allowing decryption of most messages.

As a result of these concerns, most WEP systems use only open-system authentication mode, i.e., do not provide any authentication.

Further WEP encryption vulnerabilities We briefly mention two further vulnerabilities of the WEP encryption mechanism.

The first vulnerability exploits the integrity vulnerability discussed earlier. As explained there, the attacker can flip arbitrary bits in the WEP payload message. WEP is a link-layer protocol; the payload is usually an Internet Protocol (IP) packet, whose header contains, in known position, the destination address. An attacker can change the destination address, causing forwarding of the packet directly to the attacker!

The second vulnerability is the fact that WEP uses ‘plain’ RC4, which has been shown in [224] to be vulnerable.

2.11 Encryption: Final Words

Confidentiality, as provided by encryption, is the oldest goal of cryptology, and is still critical to the entire area of cybersecurity. Encryption has been studied for millennia, but for many years, the design of cryptosystems was kept secret, in the hope of improving security. Kerckhoffs’s principle, however, has been widely adopted and caused cryptography to be widely studied and deployed, in industry and academia.

Cryptography was further revolutionized by the introduction of precise definitions and proofs of security by reduction, based on the theory of complexity. In particular, modern study of applied cryptography makes extensive use of provable security, especially *computational security*, i.e., ensuring security

properties with high probability, against probabilistic polynomial time (PPT) adversaries. We have seen a small taste of such definitions and proofs in this chapter; we will see a bit more later on, but for a real introduction to the theory of cryptography, see appropriate textbooks, such as [140, 141].

2.12 Lab and Additional Exercises

Lab 2 (Ransomware and Encryption). *In this lab, we explore the abuse of cryptography by ransomware. Ransomware encrypts the user files, and requires the user to pay ‘ransom’, with the promise of sending back the decryption key or program.*

As for the other labs in this textbook, we will provide Python scripts for generating and grading this lab (`LabGen.py` and `LabGrade.py`). If not yet posted online, professors may contact the author to receive the scripts. The lab-generation script generates random challenges for each student (or team), as well as solutions which will be used by the grading script. We recommend to make the scripts available to the students, as example of how to use the cryptographic functions. It is easy and permitted to modify these scripts to use other languages/libraries or to modify and customize them as desired.

The lab has two parts.

1. In this part, you are given a ransomware program, `R1.py`, and your task is to reverse-engineer and break it, i.e., decrypt the files without paying ransom. You will be able to do it, since `R1.py` is a simple Python program *using a shared-key cryptosystem*, specifically, the AES block cipher in CBC mode; see Section 2.6. In the next part, we will discuss more realistic ransomware, that uses public-key encryption rather than shared-key encryption, making it infeasible to recover the files without paying ransom, by reverse-engineering of the ransomware.

The ransomware `R1.py` has two outputs for each input file, say `example.txt`: its encryption, `example.txt.enc`, and a token, `example.txt.token`, to be sent with the ransom payment. The token is needed since `R1.py` selects a different random shared key to encrypt each file, e.g., `example.txt`; the attacker uses `example.txt.token` to find the decryption key.

Input: The ‘weak ransomware program’, given conveniently (and unrealistically) as a Python script, `R1.py`, the encrypted file `example.txt.enc` and the token `example.txt.token`.

Goal: reverse-engineer `R1.py` and then, using the token `example.txt.token`, recover the original file, `example.txt`.

Submission: the recovered `example.txt` file and your program, `A1.py` that produced it (given `example.txt.enc` and `example.txt.token` as input).

Note: Before encryption, the plaintext (`example.txt`) was *padded* so that its length would be a multiple number of ‘blocks’. The padding has to be removed after applying the block-cipher’s decryption function.

2. In this part you develop ‘strong’ ransomware, using public key (asymmetric) encryption. You will develop and submit the following programs:

Identifier	Cipher	Ciphertext	Plaintext	Time
A	Caesar	JUHDW SDUWB		
B	AzBy	ILFMW GZYOV		
C	ROT13	NYBAR NTNVA		
D		BLFMT OLEVI		
E		FZNEG UBHFR		
F		EUHDN UXOHV		

Table 2.6: Ciphertexts for Exercise 2.25. All plaintexts are pairs of two simple five-letter words. The three upper examples have the cipher spelled out, the three lower examples hide it ('obscurity'). It does not make them secure, but decryption may take a bit longer.

- a) A key-generation program *KG2.py*, that outputs a keypair of a public encryption key e and a private decryption key d .
- b) A ransomware program *R2.py*, which uses the encryption key e and outputs, for each input file *example.txt*, two files: its encryption, *example.txt.enc*, and a payment token, *example.txt.pay*, to be sent with the ransom payment.
- c) A token-processing program *TP2.py*, which uses the private decryption key d , and outputs, for each input payment token *example.txt.pay*, a decryption token *example.txt.d*.
- d) A decryption program *D2.py*, using the decryption token *example.txt.d*, and recovering the plaintext *example.txt* given its encryption, *example.txt.enc*, as input.

Exercise 2.25. Table 2.6 shows six ciphertexts, all using very simple substitution ciphers. The ciphers used to encrypt the top three ciphertexts are indicated, but the ciphers used to encrypt the bottom three ciphertexts are not indicated. Decipher, in random order, all six ciphertexts and measure the time it took you to decipher each of them. Fill in the blanks in the table: the plaintexts, the time it took you to decipher each message, and the ciphers used for ciphertexts D, E and F. Did the knowledge of the cipher significantly ease the cryptanalysis process?

Exercise 2.26. ConCrypt Inc. announces a new symmetric encryption scheme, CES. ConCrypt announces that CES uses a 128-bit keys and is five times faster than AES, and is the first practical cipher to be secure against computationally-unbounded attackers. Is there any method, process or experiment to validate or invalidate these claims? Describe or explain why not.

Exercise 2.27. ConCrypt Inc. announces a new symmetric encryption scheme, CES512. ConCrypt announces that CES512 uses a 512-bit keys, and as a result, is proven to be much more secure than AES. Can you point out any concerns with using CES512 instead of AES?

Exercise 2.28. Compare the following pairs of attack models. For each pair (A, B) , state whether every cryptosystem secure under attack model A is also secure under attack model B and vice versa. Prove (if you fail to prove, at least give compelling argument) your answers. The pairs are:

1. (Ciphertext only, Known plaintext)
2. (Known plaintext, Chosen plaintext)
3. (Known plaintext, Chosen ciphertext)
4. (Chosen plaintext, Chosen ciphertext)

Exercise 2.29. Alice is communicating using the GSM cellular standard, which encrypts all calls between her phone and the access tower. Identify the attacker model corresponding to each of the following cryptanalysis attack scenarios:

1. Assume that Alice and the tower use a different shared key for each call, and that Eve knows that specific, known message is sent from Bob to Alice at given times.
2. Assume (only) that Alice and the tower use a different shared key for each call.
3. Assume all calls are encrypted using a (fixed) secret key k_A shared between Alice's phone and the tower, and that Eve knows that specific, known control messages are sent, encrypted, at given times.
4. Assume (only) that all calls are encrypted using a (fixed) secret key k_A shared between Alice's phone and the tower

Exercise 2.30. We covered several encryption schemes in this chapter, including At-Bash (AzBy), Caesar, Shift-cipher, general monoalphabetic substitution, OTP, PRG-based stream cipher, RC4, block ciphers, and the ‘modes’ in Table 2.5. Which of these is: (1) stateful, (2) randomized, (3) FIL, (4) polynomial-time?

Exercise 2.31. Consider use of AES with key length of 256 bits and block length of 128 bit, for two different 128 bit messages, A and B (i.e., one block each). Bound, or compute precisely if possible, the probability that the encryption of A will be identical to the encryption of B , in each of the following scenarios:

1. Both messages are encrypted with the same randomly-chosen key, using ECB mode.
2. Both messages are encrypted with two keys, each of which is chosen randomly and independently, and using ECB mode.
3. Both messages are encrypted with the same randomly-chosen key, using CBC mode.

4. Compute now the probability the same message is encrypted to the same ciphertext, using a randomly-chosen key and CBC mode.

Exercise 2.32. Present a very efficient CPA attack on the mono-alphabetic substitution cipher, which allows complete recovery of arbitrary messages, using the encryption of one short plaintext string.

Exercise 2.33 (PRG constructions). Let $G : \{0, 1\}^n \rightarrow \{0, 1\}^{n+1}$ be a secure PRG. Is G' , as defined in each of the following sections, a secure PRG? Prove.

1. $G'(s) = G(s^R)$, where s^R means the reverse of s .
2. $G'(r \# s) = r \# G(s)$, where $r, s \in \{0, 1\}^n$.
3. $G'(s) = G(s \oplus G(s)_{1\dots n})$, where $G(s)_{1\dots n}$ are the n most-significant bits of $G(s)$.
4. $G'(s) = G(\pi(s))$ where π is a (fixed) permutation.
5. $G'(s) = G(s + 1)$.
6. (harder!) $G'(s) = G(s \oplus s^R)$.

A. Solution to $G'(r \# s) = r \# G(s)$:

B. Solution to $G'(s) = G(s \oplus G(s)_{1\dots n})$: may not be a PRG. For example, let g be a PRG from any number m bits to $m+1$ bits, i.e., output is pseudorandom string just one bit longer than the input. Assume even n ; for every $x \in \{0, 1\}^{n/2}$ and $y \in \{0, 1\}^{n/2} \cup \{0, 1\}^{1+n/2}$, let $G(x \# y) = x \# g(y)$. If g is a PRG, then G is also a PRG (why?). However, when used in the above construction:

$$\begin{aligned} G'(x \# y) &= G[(x \# y) \oplus G(x \# y)] \\ &= G[(x \# y) \oplus (x \# g(y))] \\ &= G[(x \oplus x) \# (y \oplus g(y))] \\ &= G[0^{n/2} \# y] \oplus (x \# g(y)) \\ &= 0^{n/2} \# y \oplus g(y) \end{aligned}$$

As this output begins with $n/2$ zero bits, it can be trivially distinguished from random. Hence G' is clearly not a PRG. \square

Exercise 2.34. Let $G_1, G_2 : \{0, 1\}^n \rightarrow \{0, 1\}^{2n}$ be two different candidate PRGs (over the same domain and range). Consider the function G defined in each of the following sections. Is it a secure PRG - assuming both G_1 and G_2 are secure PRGs, or assuming only that one of them is secure PRG? Prove.

1. $G(s) = G_1(s) \oplus G_2(s)$.
2. $G(s) = G_1(s) \oplus G_2(s \oplus 1^{|s|})$.
3. $G(s) = G_1(s) \oplus G_2(0^{|s|})$.

Exercise 2.35. Let $G : \{0, 1\}^n \rightarrow \{0, 1\}^m$ be a secure PRG, where $m > n$.

1. Let $m = n + 1$. Use G to construct a secure PRG $G' : \{0, 1\}^n \rightarrow \{0, 1\}^{2n}$.
2. Let $m = 2n$, and consider $G'(x) = G(x) \oplus G(x+1)$. Is G' a secure PRG?
3. Let $m = 2n$. Use G to construct a secure PRG $G' : \{0, 1\}^n \rightarrow \{0, 1\}^{4 \cdot n}$.
4. Let $m = 4n$. Use G to construct a secure PRG $\hat{G} : \{0, 1\}^n \rightarrow \{0, 1\}^{64 \cdot n}$.

Exercise 2.36. Let f, f' be two functions from n bit binary strings to n' bit binary strings, i.e., $f, f' : \{0, 1\}^n \rightarrow \{0, 1\}^{n'}$.

1. Let $m_1 \neq m_2 \in \{0, 1\}^n$ be two different random n bit strings. Present the best upper and lower bounds you can, for the probability that $f(m_1) = f(m_2)$, assuming $n > n'$:

$$\underline{\hspace{2cm}} \leq \Pr_{m_1 \neq m_2 \leftarrow \{0, 1\}^n}(f(m_1) = f(m_2)) \leq \underline{\hspace{2cm}}$$

Justify your answer.

2. Repeat, when f is a random function from $\{0, 1\}^n$ to $\{0, 1\}^{n'}$:

$$\underline{\hspace{2cm}} \leq \Pr_{m_1 \neq m_2 \leftarrow \{0, 1\}^n}(f(m_1) = f(m_2)) \leq \underline{\hspace{2cm}}$$

3. Repeat items 1 and 2 for the case $n = n'$.

4. Repeat items 1 and 2 for the case $n' > n$.

5. Repeat items 1 and 2, for the probability that $f(m_1) = f'(m_2)$. In item 2, only f is chosen randomly.

Exercise 2.37. Let f_k be a (secure) Pseudo-Random Function (PRF) from n bit binary strings to n' bit binary strings, i.e., $f : \{0, 1\}^* \times \{0, 1\}^n \rightarrow \{0, 1\}^{n'}$. Assume that the key k is chosen randomly as a string of length l ; the key length l is specified so you can reference it in your responses, but do so only if you find it relevant; otherwise, you may ignore it. Assume l is sufficiently large for the PRF to be secure.

1. Is it possible that $n < n'$? Is it possible that $n' < n$? Explain.
2. Let $m_1 \neq m_2 \in \{0, 1\}^n$ be two different random n bit strings. Present the best upper and lower bounds you can, for the probability that $f_k(m_1) = f_k(m_2)$, assuming $n < n'$:

$$\underline{\hspace{2cm}} \leq \Pr_{m_1 \neq m_2 \leftarrow \{0, 1\}^n}(f_k(m_1) = f_k(m_2)) \leq \underline{\hspace{2cm}}$$

Justify your answer.

3. Repeat for the case $n = n'$.

4. Repeat for the case $n > n'$.
5. Compare your answers with Exercise A.9.

Exercise 2.38 (Ad-Hoc PRF competition project). *In this exercise, you will experiment in trying to build directly a cryptographic scheme - in this case, a PRF - as well as in trying to ‘break’ (cryptanalyze) it. This exercise is best done by multiple groups, with each group consisting of one or few persons.*

1. *In the first phase, each group will design a PRF, whose input, key and output are all 64 bits long. The PRF should be written in Python (or some other agreed programming language), and only use the basic mathematical operations: table lookup, modular addition/subtraction/multiplication/-division/remainder, XOR, max, min, and rotations. You may also use comparisons and conditional code. The length of your program should not exceed 400 characters, and it must be readable. You will also provide (separate) documentation.*
2. *All groups will be given the documentation and code of the PRFs of all other groups, and try to design programs to distinguish these PRFs from a random function (over same input and output domains). A distinguisher is considered successful if it is able to distinguish in more than 1% of the runs.*

Exercise 2.39. Let f be a secure Pseudo-Random Function (PRF) with n bit keys, domain and range, and let k be a secret, random n bit key. Derive from k , using f , two pseudorandom keys k_1, k_2 , e.g., one for encryption and one for authentication. Each of the derived keys k_1, k_2 should be $2n$ -bits long, i.e., twice the length of k . Note: the two keys should be independent, i.e., each of them (e.g., k_1) should be pseudorandom, even if the adversary is given the other (e.g., k_2).

1. $k_1 = \underline{\hspace{1cm}}$
2. $k_2 = \underline{\hspace{1cm}}$

Exercise 2.40 (PRF constructions). Let $F_k(m) : \{0, 1\}^n \times \{0, 1\}^n \rightarrow \{0, 1\}^n$ be a secure PRF. Is F' , as defined in each of the following sections, a secure PRF? Justify your answers. Where the function is not a secure PRF, present the adversary that can distinguish between the function and a random function (as in Exercise 2.11). Where the function is a PRF, a precise proof as in Exercise 2.12 is best, but a good intuitive argument will also do.

1. $\hat{F}_k(m) = F_k(m^R)$, where m^R means the reverse of m .
2. $\hat{F}_k(m_L \# m_R) = F_k(m_L) \# F_k(m_R)$.
3. $\hat{F}_k(m_L \# m_R) = (m_L \oplus F_k(m_R)) \# (F_k(m_L) \oplus m_R)$.

4. $\hat{F}_k(m) = LSb(F_k(m))$, where LSb returns the least-significant bit of the input.

Exercise 2.41 (Key dependent message security). Several works design cryptographic schemes such as encryption schemes, which are secure against a ‘key dependent message attack’, where the attacker specifies a function f and receives encryption $E_k(f(k))$, i.e., encryption of the message $f(k)$ where k is the secret key. See [56].

1. Extend the definition of secure pseudo-random function for security against key-dependent message attacks.
2. Suppose that F is secure pseudo-random function. Show a (‘weird’) function F' which is also a secure pseudo-random function, but not secure against key-dependent message attacks.

Exercise 2.42 (KDM security). Several works design cryptographic schemes such as encryption schemes, which are secure against a ‘key dependent message attack’. Recall that in many cryptographic definitions, the attacker has access to one or more oracle function, e.g., the chosen-plaintext oracle for encryption $E_k(\cdot)$ using secret key k . In a key-dependent message attack, the attacker has similar access to the oracle function, but instead of specifying directly the value of the input to the oracle, the attacker specifies a function f and the oracle is applied to $f(k)$ where k is receives the relevant cryptographic function. For example for chosen-plaintext oracle for encryption using a shared key k , the attacker receives $E_k(f(k))$. See, e.g., [56].

1. Extend the definition of secure pseudo-random function, to define a PRF scheme secure against key-dependent message attacks.
2. Repeat, for a block cipher (reversible PRP).
3. Suppose that F is secure pseudo-random function. Show a (‘weird’) function F' which is also a secure pseudo-random function, but not secure against key-dependent message attacks.
4. Extend the definition of IND-CPA secure encryption to allow for key-dependent message attacks.

Exercise 2.43 (Stateful PRG, ANSI X9.31 and the DUHK attack). The ANSI X9.31 is a well-known design of a stateful PRG design built using a block cipher E , illustrated in Fig. 2.37. In this exercise we investigate a weakness in it, presented in [189]; it was recently shown to be still relevant for some devices using this standard, in the so-called DUHK attack [84]. Our presentation is a slight simplification of the X9.31 design but retains the important aspects of the attack.

The stateful PRG is used in ‘rounds’, with the state of round i being output from round $i-1$. Specifically, the X9.31 PRG works as follows, given the current state s_{i-1} , with s_0 selected randomly, and some ‘timestamp’ T_i . First, compute

an ‘internal’ value, $x_i = E_k(T_i)$. Then, output the values $r_i = E_k(x_i \oplus s_{i-1})$ and $s_i = E_k(r_i \oplus x_i)$. See Fig. 2.37. The specification does not restrict the choice of k , and several implementations use constant k as part of their code; assume, therefore, that k is known.

1. Assume T_i , k and s_{i-1} are known. Show how the attacker can find r_i and s_i .
2. Assume that the values $\{T_j\}$ are known for all j , and that k , r_i are known (for specific i). Show how an attacker can compute $\{s_j, r_j\}$ for every j .

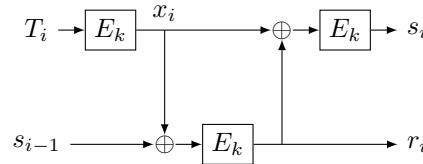


Figure 2.37: A single round of the ANSI X9.31 stateful pseudorandom generator (PRG), using block cipher $E_k(x)$, e.g., AES. This figure was adapted from [84].

Exercise 2.44 (Cascade is not a robust combiner for PRFs). Let $F', F'' : \{0,1\}^* \times D \rightarrow D$ be two polynomial-time computable functions, and let their cascade, denoted $F \equiv F' \circ F''$ be defined as:

$$F_{(k', k'')}(x) \equiv F'_{k'} \circ F''_{k''}(x) \equiv F'_{k'}(F''_{k''}(x)) \quad (2.65)$$

Give an example of F', F'' s.t. one of them is a PRF, yet their cascade $F \equiv F' \circ F''$ is not a PRF. This shows that cascade is not a robust combiner for PRFs.

Exercise 2.45. A message m of length 256 bytes is encrypted using a 128-bit block cipher, resulting in ciphertext c . During transmission, the 200th bit was flipped due to noise. Let c' denote c with the 200th bit flipped, and m' denote the result of decryption of c' .

1. Which bits in m' would be identical to the bits in m , assuming the use of each of the following modes: (1) ECB, (2) CBC, (3) OFB, (4) CFB? Explain (preferably, with diagram).
2. For each of the modes, specify which bits is predictable as a function of the bits of m and the known fact that the 200th bit flipped.

Exercise 2.46. Consider a scenario where randomness is scarce, motivating attempts to design encryption schemes that use less randomization. Specifically, consider the following two variants of CBC mode, both using, per message, only twenty random bits, rather than n random bits (block size) in standard CBC. Both variants are identical to CBC mode, except for the choice of c_0 (the initialization vector), which is as specified below; both use a twenty-bits random string $r \xleftarrow{\$} \{0,1\}^{20}$. Show that neither variant suffices to ensure IND-CPA.

Append zeros: $c_0 = r||0^{n-20}$.

Pseudorandomly: $c_0 = E_k(r)$.

Note: your solution may require up to few million queries; just make sure the number of queries is polynomial in n .

Exercise 2.47 (Modes of operation: decryption and correctness). *Table 2.5 specifies only the encryption process for each mode. Write the decryption process for each mode and show that correctness is satisfied.*

Exercise 2.48. *Hackme Bank protects money-transfer orders digitally sent between branches, by encrypting them using a block cipher. Money transfer orders have the following structure: $m = f \# r \# t \# x \# y \# p$, where f, r are each 20-bits representing the payer (from) and the payee (recipient), t is a 32-bit field encoding the time, x is a 24 bit field representing the amount, y is a 128-bit comment field defined by the payer and p is 32-bit parity fields, computed as the bitwise-XOR of the preceding 32-bit words. Orders with incorrect parity, outdated or repeating time field, or unknown payer/payee are ignored.*

Mal captures ciphertext message x containing money-transfer order of 1\$ from Alice to his account. You may assume that Mal can ‘trick’ Alice into including a comment field y selected by Mal. Assume 64-bit block cipher. Can Mal cause transfer of larger amount to his account, and how, assuming use of the following modes:

1. ECB
2. CBC
3. OFB
4. CFB

Solution: The first block contains f, r (10 bits each), and top 24 bits of the time t , the second block contains 8 more bits of the time, x (24 bits) and 32 bits of the comment; block three contains 64 bits of comments, and block four contains 32 bits of comment and 32 bits of parity. Denote these four plaintext blocks by $m_1 \# m_2 \# m_3 \# m_4$.

Denote the ciphertext blocks captured by Mal as $c_0 \# c_1 \# c_2 \# c_3 \# c_4$, where c_0 is the IV.

1. ECB: attacker select the third block (completely comment) to be identical to the second block, except for containing the maximal value in the 24 bits from bit 8 to bit 31. The attacker then switches between the third and fourth block before giving to the bank. Parity bits do not change.
2. CBC: Attacker chooses y s.t. $m_3 = m_2$. Then, the attacker sends to the bank the manipulated message $z_0 \# c_3 \# c_3 \# c_3 \# c_4$, where $z_0 = m_1 \oplus m_3 \oplus c_2$. As a result, decryption of the first block retrieves m_1 correctly (as $m_1 = z_0 \oplus m_3 \oplus c_2$), and decryption of the last block

similarly retrieves m_4 correctly (no change in c_3, c_4). However, both the second and the third block, decrypt to the value $(c_3 \oplus c_2 \oplus m_3)$. Hence, the 32 bit XOR of the message does not change. The decryption of the second block (to $c_3 \oplus c_2 \oplus m_3$) is likely to leave the time value valid - and to increase the amount considerably.

3. OFB: the solution is trivial since Mal can flip arbitrary bits in the decrypted plaintext (by flipping corresponding bits in the ciphertext).
4. CFB: as in CBC, attacker chooses y s.t. $m_3 = m_2$. Attacker sends to the bank the manipulated message $c_0 \parallel c_1 \parallel c_1 \parallel c_1 \parallel z_4$ where $z_4 = p_4 \oplus c_2 \oplus p_2$.

Exercise 2.49 (Affine block cipher). *Hackme Inc. proposes the following highly-efficient block cipher, using two 64-bit keys k_1, k_2 , for 64-bit blocks: $E_{k_1, k_2}(m) = (m \oplus k_1) + k_2 \pmod{2^{64}}$.*

1. Show that E_{k_1, k_2} is an invertible permutation (for any k_1, k_2), and the inverse permutation D_{k_1, k_2} .
2. Show that (E, D) is not a secure block cipher (invertible PRP).
3. Show that encryption using (E, D) is not CPA-IND, when used in the following modes: (1) ECB, (2) CBC, (3) OFB, (4) CFB.

Exercise 2.50 (How not to build PRP from PRF). *Suppose F is a secure PRF with input, output and keyspace all of length n bits. For $x_L, x_R \in \{0, 1\}^n$, let $F'_k(x_L \parallel x_R) = F_k(x_L) \parallel F_k(x_R)$ and $F''_k(x_L \parallel x_R) = F_k(x_L \oplus x_R) \parallel F_k(x_L \oplus F_k(x_L \oplus x_R))$. Prove that neither F'_k nor F''_k are a PRP.*

Exercise 2.51 (Building PRP from a PRF). *Suppose you are given a secure PRF F , with input, output and keyspace all of length n bits. Show how to use F to construct:*

1. A PRP, with input and output length $2n$ bit and key length n bits,
2. A PRP, with input, output and key all of length n bits.

Exercise 2.52. *Show that the simple padding function $\text{pad}(m) = m \parallel 0^l$, fails to prevent CCA attacks against most of the modes-of-operation (Fig. 2.5), when $l \leq n$. The attacker may perform CPA and CCA queries, and the plaintext contains multiple blocks.*

Exercise 2.53 (Indistinguishability definition). *Let (E, D) be a stateless shared-key encryption scheme, and let p_1, p_2 be two plaintexts. Let x be 1 if the most significant bits of p_1, p_2 are identical and 0 otherwise, i.e., $x = \{1 \text{ if } \text{MSb}(p_1) = \text{MSb}(p_2), \text{ else } 0\}$. Assume that there exists an efficient algorithm X that computes x given the ciphertexts, i.e., $x = X(E_k(p_1), E_k(p_2))$. Show that this implies that (E, D) is not IND-CPA secure, i.e., there is an efficient algorithm*

ADV which achieves significant advantage in the IND-CPA experiment. Present the implementation of ADV by filling in the missing code below:

$$\begin{aligned} \text{ADV}^{E_k}(\text{'Choose'}, 1^n) : & \{ \underline{\hspace{10em}} \} \\ \text{ADV}^{E_k}(\text{'Guess'}, s, c^*) : & \{ \underline{\hspace{10em}} \} \end{aligned}$$

Exercise 2.54 (Robust combiner for PRG). 1. Given two candidate PRGs, say G_1 and G_2 , design a robust combiner, i.e., a ‘combined’ function G which is a secure PRG is either G_1 or G_2 is a secure PRG.

2. In the design of the SSL protocol, there were two candidate PRGs, one (say G_1) based on the MD5 hash function and the other (say G_2) based on the SHA-1 hash function. The group decided to combine the two; a simplified version of the combined PRG is $G(s) = G_2(s \# G_1(s))$. Is this a robust-combiner, i.e., a secure PRG provided that either G_1 or G_2 is a secure PRG?

Hint: Compare to Lemma 2.1. You may read on hash functions in Chapter 3, but the exercise does not require any knowledge of that; you should simply consider the construction $G(s) = G_2(s \# G_1(s))$ for arbitrary functions G_1 , G_2 .

□

Exercise 2.55 (Using PRG for independent keys). In Example 2.6, we saw how to use a PRF to derive multiple pseudo-random keys from a single pseudo-random key, using a PRF.

1. Show how to derive two pseudo-random keys, using a PRG, say from n bits to $2n$ bits.
2. Show how to extend your design to derive four keys from the same PRG, or any fixed number of pseudo-random keys.

Exercise 2.56. Let (E, D) be a block cipher which operates on 20 byte blocks; suppose that each computation of E or D takes 10^{-6} seconds (one microsecond), on given chips. Using (E, D) you are asked to implement a secure high-speed encrypting/decrypting gateway. The gateway receives packets at line speed of 10^8 bytes/second, but with maximum of 10^4 bytes received at any given second. The goal is to have minimal latency, using minimal number of chips. Present an appropriate design, argue why it achieves the minimal latency and why it is secure.

Exercise 2.57. Consider the AES block cipher, with 256 bit key and 128 bit blocks, and two random one-block (128 bit) messages, m_1 and m_2 , and two random (256-bit) keys, k_1 and k_2 . Calculate (or approximate/bound) the probability that $E_{k_1}(m_1) = E_{k_2}(m_2)$.

Exercise 2.58 (PRF→PRG). Present a simple and secure construction of a PRG, given a secure PRF.

Exercise 2.59 (Independent PRGs). *Often, a designer has one random or pseudo-random ‘seed/key’ binary string $k \in \{0,1\}^*$, from which it needs to generate two or more independently pseudorandom strings $k_0, k_1 \in \{0,1\}^*$; i.e., each of these is pseudorandom, even if the other is given to the (PPT) adversary. Let PRG be a pseudo-random generator, which on input of arbitrary length l bits, produces $4l$ output pseudorandom bits. For each of the following designs, prove its security (if secure) or its insecurity (is insecure).*

1. For $b \in \{0,1\}$, let $k_b = \text{PRG}(b \# k)$.
2. For $b \in \{0,1\}$, let $k_b = \text{PRG}(k)[(b \cdot 2 \cdot |k|) \dots ((2 + b) \cdot |k| - 1)]$.

Solution:

1. Insecure, since it is possible for a secure PRG to ignore the first bit, i.e., $\text{PRG}(b \# s) = \text{PRG}(\bar{b} \# s)$, resulting in $k_0 = \text{PRG}(0 \# k) = \text{PRG}(1 \# k) = k_1$. We skip the (simple) proof that such a PRG may be secure.
2. Secure, since each of these is a (non-overlapping) subset of the output of the PRG.

Exercise 2.60 (Indistinguishability hides partial information). *In this exercise we provide an example to the fact that a cryptosystem that ensures indistinguishability (IND-CPA), is guaranteed not to leak partial information about plaintext, including relationships between the plaintext corresponding to different ciphertexts. Let (E, D) be an encryption scheme, which leaks some information about the plaintexts; specifically we assume that there exists an efficient adversary A s.t. for two ciphertexts c_1, c_2 of E , holds $A(c_1, c_2) = 1$ if and only if the plaintexts share a common prefix, e.g., $c_1 = E_k(ID \# m_1)$ and $c_2 = E_k(ID \# m_2)$ (same prefix, ID). Show that this implies that (E, D) is not IND-CPA secure.*

Exercise 2.61 (Encrypted cloud storage). *Consider a set P of n sensitive (plaintext) records $P = \{p_1, \dots, p_n\}$ belonging to Alice, where $n < 10^6$. Each record p_i is $l > 64$ bits long ($(\forall i)(p_i \in \{0,1\}^l)$). Alice has very limited memory, therefore, she wants to store an encrypted version of her records in an insecure/untrusted cloud storage server S ; denote these ciphertext records by $C = \{c_1, \dots, c_n\}$. Alice can later retrieve the i^{th} record, by sending i to S , who sends back c_i , and then decrypting it back to p_i .*

1. Alice uses some secure shared key encryption scheme (E, D) , with l bit keys, to encrypt the plaintext records into the ciphertext records. The goal of this part is to allow Alice to encrypt and decrypt each record i using a unique key k_i , but maintain only a single ‘master’ key k , from which it can easily compute k_i for any desired record i . One motivation for this is to allow Alice to give keys to specific record(s) k_i to some other users (Bob, Charlie,...), allowing decryption of only the corresponding ciphertext

c_i , i.e., $p_i = D_{k_i}(c_i)$. Design how Alice can compute the key k_i for each record (i), using only the key k and a secure block cipher (PRP) (F, F^{-1}) , with key and block sizes both l bits. Your design should be as efficient and simple as possible. Note: do not design how Alice gives k_i to relevant users - e.g., she may do this manually; and do not design (E, D) .

Solution: $k_i = \underline{\hspace{2cm}}$

2. Design now the encryption scheme to be used by Alice (and possibly by other users to whom Alice gave keys k_i). You may use the block cipher (F, F^{-1}) , but not other cryptographic functions. You may use different encryption scheme (E^i, D^i) for each record i . Ensure confidentiality of the plaintext records from the cloud, from users (not given the key for that record), and from eavesdroppers on the communication. Your design should be as efficient as possible, in terms of the length of the ciphertext (in bits), and in terms of number of applications of the secure block cipher (PRP) (F, F^{-1}) for each encryption and decryption operation. In this part, assume that Alice stores P only once, i.e., never modifies records p_i . Your solution may include a new choice of k_i , or simply use the same as in the previous part.

Solution: $k_i = \underline{\hspace{2cm}}$,

$E_{k_i}^i(p_i) = \underline{\hspace{2cm}}$,

$D_{k_i}^i(c_i) = \underline{\hspace{2cm}}$.

3. Repeat, when Alice may modify each record p_i few times (say, up to 15 times); let n_i denote number of modifications of p_i . The solution should allow Alice to give (only) her key k , and then Bob can decrypt all records, using only the key k and the corresponding ciphertexts from the server. Note: if your solution is the same as before, this may imply that your solution to the previous part is not optimal.

Solution: $k_i = \underline{\hspace{2cm}}$,

$E_{k_i}^i(p_i) = \underline{\hspace{2cm}}$,

$D_{k_i}^i(c_i) = \underline{\hspace{2cm}}$.

4. Design an efficient way for Alice to validate the integrity of records retrieved from the cloud server S . This may include storing additional information A_i to help validate record i , and/or changes to the encryption/decryption scheme or keys as designed in previous parts. As in previous parts, your design should only use the block cipher (F, F^{-1}) .

Solution: $k_i = \underline{\hspace{2cm}}$,

$E_{k_i}^i(p_i) = \underline{\hspace{2cm}}$,

$D_{k_i}^i(c_i) = \underline{\hspace{2cm}}$,

$A_i = \underline{\hspace{2cm}}$.

5. Extend the keying scheme from the first part, to allow Alice to also compute keys $k_{i,j}$, for integers $i, j \geq 0$ s.t. $1 \leq i \cdot 2^j + 1, (i+1) \cdot 2^j \leq n$, where $k_{i,j}$ would allow (efficient) decryption of ciphertext records $c_{i \cdot 2^j + 1}, \dots, c_{(i+1) \cdot 2^j}$. For example, $k_{0,3}$ allows decryption of records c_1, \dots, c_8 , and $k_{3,2}$ allows

decryption of records c_{13}, \dots, c_{16} . If necessary, you may also change the encryption scheme (E^i, D^i) for each record i .

Solution: $k_{i,j} = \underline{\hspace{2cm}}$,

$E_{k_i}^i(p_i) = \underline{\hspace{2cm}}$,

$D_{k_i}^i(p_i) = \underline{\hspace{2cm}}$.

Exercise 2.62 (Modes vs. attack models.). For every mode of encryption we learned (see Table 2.5):

1. Is this mode always secure against any of the attack models we discussed (CTO, KPA, CPA, CCA)?
2. Assume this mode is secure against KPA. Is it then also secure against CTO? CPA? CCA?
3. Assume this mode is secure against CPA. Is it then also secure against CTO? KPA? CCA?

Justify your answers.

Exercise 2.63. Recall that WEP encryption is defined as: $WEP_k(m; IV) = [IV, RC4_{IV,k} \oplus (m + CRC(m))]$, where IV is a random 24-bit initialization window, and that CRC is a error-detection code which is linear, i.e., $CRC(m \oplus m') = CRC(m) \oplus CRC(m')$. Also recall that WEP supports shared-key authentication mode, where the access point sends random challenge r , and the mobile response with $WEP_k(r; IV)$. Finally, recall that many WEP implementations use 40-bit key.

1. Explain how an attacker may efficiently find the 40-bit WEP key, by eavesdropping on the shared-key authentication messages between the mobile and the access point.
2. Present a hypothetical scenario where WEP would have used a fixed value of IV to respond to all shared-key authentication requests, say $IV=0$. Show another attack, that also finds the key using the shared-key authentication mechanism, but requires less time per attack. Hint: the attack may use (reasonable) precomputation process, as well as storage resources; and the attacker may send a ‘spoofed’ challenge which the client believes was sent by the access point.
3. Identify the attack models exploited in the two previous items: CTO, KPA, CPA or CCA?
4. Suppose now that WEP is deployed with a long key (typically 104 bits). Show another attack which will allow the attacker to decipher (at least part) of the encrypted traffic.

Chapter 3

Integrity: Cryptographic Hash Functions and Accumulators

Integrity is the ability to check if an object was modified, using a concise *digest* of the (original) object. In this chapter, we discuss the two main type of cryptographic integrity mechanisms: *hash functions* and *accumulator schemes*.

Much of this chapter deals with cryptographic *hash functions*, used to compute the digest of a binary string. Cryptographic hash functions are among the most widely-used cryptographic schemes, and have many diverse properties, uses, and applications. In Section 3.1 we introduce cryptographic hash functions, their properties and variants. In Section 3.2 and Section 3.3, respectively, we discuss the two main *integrity* properties of hash functions, *collision resistance* hash functions (*CRHF*) and *second preimage resistant* (*SPR*) hash functions. In Section 3.4 we discuss *one-way functions* (*OWF*), which is another property often expected from cryptographic hash functions, and used for different purposes. In Section 3.5 we discuss one final important property often expected from cryptographic hash functions: *randomness extraction*; and in Section 3.6 we introduce the *Random Oracle Model*, an important paradigm often used to provide a simplified security analysis of protocols and schemes using cryptographic hash functions. Later, in Chapter 4, we discuss the *use* of cryptographic hash functions for *authentication*, in MAC and signature schemes.

In Section 3.7 we introduce *cryptographic accumulators*. Accumulators can be seen as a generalization of hash functions, accepting a sequence/set of binary strings as input, and providing additional functionalities such as *Proof of Inclusion* (*PoI*). In Section 3.8 we present the *Merkle-Damgård* accumulator, a construction that is more known for its use in construction cryptographic hash functions from their fixed-input-length variant, called *compression functions*. Finally, in Section 3.9 we introduce the widely used *Merkle tree* accumulator.

Both hash functions and accumulators have been extensively studied and widely deployed in practice. However, as we will see, their correct, secure use requires precise understanding of their properties; designs based on intuitive understanding have often proved vulnerable, and we will see some examples.

Therefore, precise definitions are critical. We define what we consider as the most important notions, mostly focusing, where possible, on the more-applied case of keyless hash functions; in more advanced texts on cryptography, you will find additional, and sometimes different, definitions.

3.1 Introducing cryptographic hash functions, their properties and variants

Hash functions map variable input length (VIL) binary strings to n -bit strings, referred to as the *digest* of the input. Since the input may be arbitrarily long, and the output is always n bits, the basic property of hash functions is that the digest (output) is normally shorter than the input, a property often referred to as *compression*¹. The compression property, on its own, may be achieved trivially, e.g., by truncating the input; hash functions are expected to satisfy additional properties, as we will discuss.

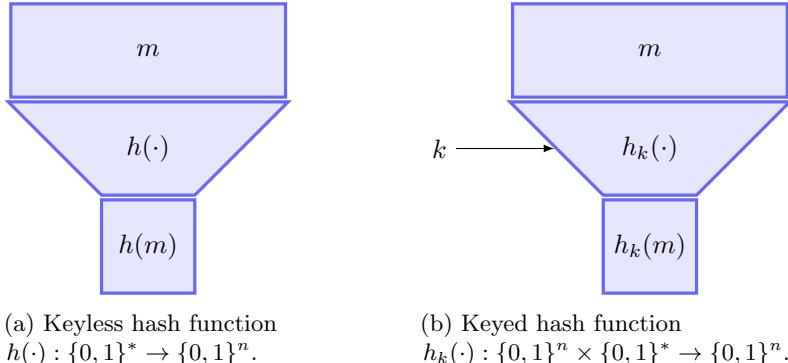


Figure 3.1: Keyless and Keyed Hash Functions: mapping from a variable-length input to n -bits output (digest). For simplicity, the keyed hash use n as length of both digest and key.

As illustrated in Figure 3.1, hash functions may be keyed ($h_k(\cdot) : \{0, 1\}^n \times \{0, 1\}^* \rightarrow \{0, 1\}^n$) or keyless ($h(\cdot) : \{0, 1\}^* \rightarrow \{0, 1\}^n$). We discuss *cryptographic hash functions*, i.e., hash functions which should satisfy different security properties, e.g., collision resistance, although hash functions are also used for non-security applications.

To key or not to key? Existing standards of cryptographic hash functions, are all of *keyless* hashes, and use a fixed digest length n . For example, the SHA-1 standard cryptographic hash function subsection 3.1.4 uses $n = 160$, i.e., the output length is 160 bits. However, as we will see, keyless hash function

¹Do not be confused with *compression functions*, which we define later, which compress from m bits strings to $n < m$ bits strings.

cannot ensure important security requirements, e.g., collision resistance, which motivates using keyed hash functions.

We discuss both keyed and keyless hash functions, focusing, where possible, on keyless hash functions, since they are simpler and more common in applied cryptographic protocols and systems. For the same reasons, we also focus on hash functions of fixed digest length n .

The key k of keyed hash functions, including cryptographic keyed hash functions, is usually *non-secret*².

3.1.1 Warm-up: hashing for efficiency

Before we focus on cryptographic hash functions, we first discuss briefly the use of hash functions for randomly mapping data, as used (also) for load-balancing and other ‘classical’, non-adversarial scenarios. Our goal is to provide intuition for the required security properties and awareness of some of the challenges.

A common (non-cryptographic) application of hash functions is to map the inputs into the possible digests values (‘bins’) in a ‘random’ manner, i.e., with a roughly equal probability of assignment to each bin (digest value). For the typical case of n -bit digest values, there would be 2^n bins. This property is used in many algorithms and data structures, to improve efficiency and fairness. This is illustrated in Fig. 3.2. Here, a hash function h maps from the set of names (given as unbounded-length strings), to a smaller set, say the set of n -bit binary strings. This is a special case of *load balancing*, i.e., avoiding unevenly use of computing resources, which may result in overload of one resource concurrently with under-utilization of an alternative resource. The goal is to roughly balance the number of entries (names) assigned to each bin.

Of course, in cryptography, and cybersecurity in general, we mainly consider adversarial settings. In the context of load-balancing applications as shown in Fig. 3.2, this refers to an adversary who can manipulate some of the input names, and whose goal may be to cause imbalanced allocation of names to bins, i.e., many collisions - which can cause bad performance, potentially even a *Denial of Service (DoS)*, i.e., a disruption or degradation of the service provided.

Consider an attacker whose goal is to degrade the performance for a particular name, say Bob, as part of a *Denial-of-Service (DoS)* attack. The attacker may provide to the system a long list of names x_1, x_2, \dots , deviously selected such that all of them are mapped to the same bin as Bob. We refer to inputs x, x_1 that have the same digest, i.e., $h(x) = h(x_1)$, as a *collision*. The attacker’s goal, therefore, is to find many values x_1, x_2, \dots , which all collide with the string $x = \text{‘Bob’}$, i.e., $h(\text{‘Bob’}) = h(x_1) = h(x_2) = \dots$. See Figure 3.3.

DoS attacks of this type, which cause high computational overhead, are usually referred to as an *Algorithmic-Complexity DoS Attacks*. One way in which attackers may exploit an algorithmic complexity DoS attack, is to cause excessive overhead for network security devices, such as *malware/virus scanners*,

²Some works refer to the use of hashes with secret keys. However, the applications are typically of a MAC or PRF function, possibly constructed from a cryptographic hash function, a topic we discuss in subsection 4.6.3.

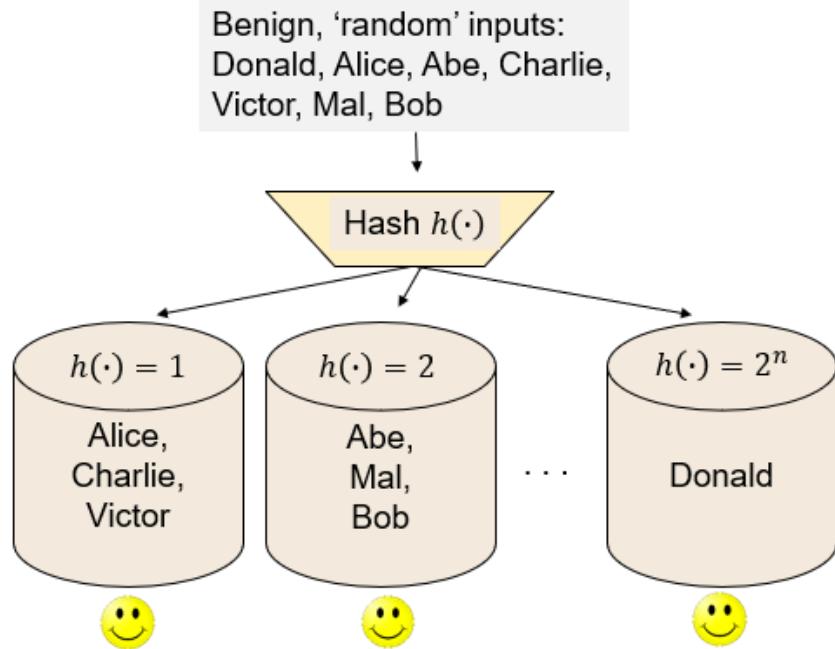


Figure 3.2: Load-balancing with (keyless) hash function $h(\cdot)$

Intrusion-Detection Systems (IDS) and *Intrusion-Prevention Systems (IPS)*. The attack may cause the IDS/IPS systems to become ineffective, allowing the attacker to avoid detection. For further discussion of algorithmic-complexity and other Denial-of-Service (DoS) attacks, see [89, 162].

Note that for *any* hash function h and input x (e.g., $x = \text{'Bob'}$), it is possible to find other inputs $\{x'_1, x'_2, \dots\}$ which collide with x , i.e., $(\forall i)h(x'_i) = h(x)$, by randomly testing different inputs and collecting those whose hash is the same as $h(x)$. However, for digest length n , there are 2^n bins, hence the probability of such random guess to be a collision with ‘Bob’ is only $\frac{1}{2^n} = 2^{-n}$, i.e., negligible in n .

For some hash functions, including hash functions used for non-cryptographic applications, there are *efficient* ways for the attacker to find collisions, rather than testing random inputs. This includes hash functions that provide sufficiently-randomized mapping for ‘natural’ inputs, which result from a benign selection process. We say that such hash functions are *not collision resistant*, i.e., these are functions where collisions can be found efficiently when the inputs are selected by an attacker. See the following exercise.

Exercise 3.1. Given an alphabetic string x , let $\text{num}(x, i)$ be the alphabetical-position of the i^{th} letter in x with the first letter (‘ a ’) being in position one.

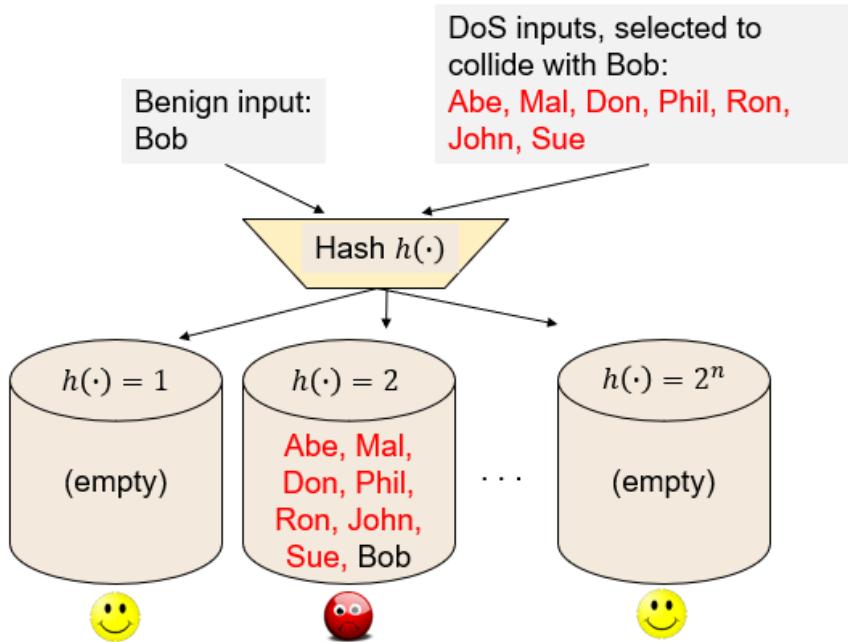


Figure 3.3: Algorithmic Complexity Denial-of-Service Attack exploiting insecure hash function h to cause many collisions

For example, $\text{num}(\text{'hello'}, 2) = 5$ since 'e' is the fifth letter in the alphabet, and $\text{num}(\text{'abcdef'}, i) = i$ for $1 \leq i \leq 6$. Consider hash function $h(x) = \sum_{i=1}^{|x|} \text{num}(x, i) \bmod 27$, i.e., sum of all the letters (mod 27). Show how an attacker may easily generate a set $\{x_1, x_2, \dots\}$ of any desired number of strings colliding with Bob, i.e., for every x_i holds: $h(x_i) = h(\text{'Bob'})$. The attacker should not need to compute the hash value for many different strings. Give three examples of such colliding strings. As an extra challenge, try to have your strings be real names!

It isn't very surprising that collisions can be found efficiently for a hash function not designed for collision resistance. However, this indicates the importance of defining carefully the collision resistance requirement and other security requirements from cryptographic hash functions. This follows the *attack model and security requirements principle* (Principle 1). In Section 3.2 we define *collision-resistant hash functions (CRHF)*; intuitively, in such functions, an attacker cannot efficiently find a collision. This should foil the Algorithmic Complexity Denial-of-Service Attack of Fig. 3.3, provided we use a sufficiently-long digest length n (i.e., 2^n bins). See Figure 3.4.

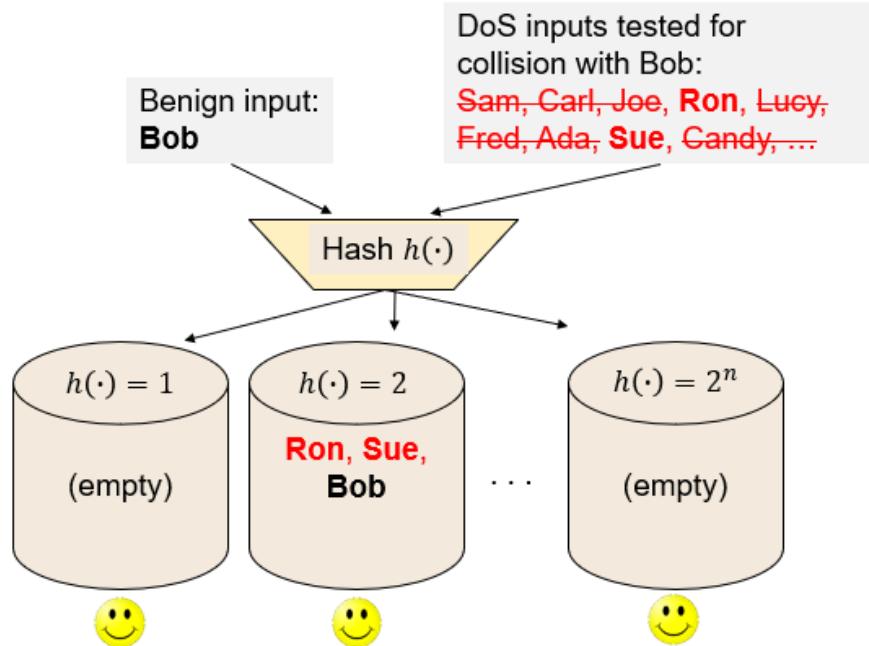


Figure 3.4: Load balancing with a *collision-resistant hash function (CRHF)* with n bits digest, i.e., using 2^n bins. The probability of a random name to collide with ‘Bob’ is only 2^{-n} ; furthermore, the probability of collision is negligible for guesses by all efficient algorithms.

3.1.2 Properties of cryptographic hash functions

Cryptographic hash functions are used for many different applications, often assuming different security properties; unfortunately, these assumptions are not always made explicitly, and the assumed properties are not always clearly defined. Roughly, these properties fall into three broad goals: *integrity*, *confidentiality* and *randomness*. Intuitively, *integrity* ensures the uniqueness of the message, given the digest; *confidentiality* ensures the digest does not ‘expose’ the message; and *randomness* ensures that the digest is pseudorandom, provided that the input ‘contains sufficient randomness’.

We define four security requirements: *collision resistance*, *second-preimage resistance*, *one-way function* (also referred to as preimage resistance), and *randomness extraction*. Table 3.1 maps these requirements to the three goals, and gives an abridged descriptions of these properties, for keyless hash functions.

Assuming that a given hash function has these security properties makes it possible to use the hash function for different applications, ensuring security as long as the hash function indeed satisfies the properties assumed. However, we should only assume properties which the hash function was designed and tested

Goal	Requirement	Abridged description
Integrity	Collision resistance (CRHF; Definition 3.1)	Can't find collision (m, m') , i.e., $m \neq m'$ yet $h(m) = h(m')$.
Integrity	Second-preimage resistance (SPR; Definition 3.7)	Can't find collision to a random m : $m' \neq m$ yet $h(m) = h(m')$.
Confidentiality	One-way function (OWF; Definition 3.9)	Given $h(m)$ for random m , can't find m' s.t. $h(m) = h(m')$.
Randomness	Randomness extractor (Section 3.5)	If input is sufficiently random, then output is pseudorandom.
All	Random oracle model (ROM; §3.6)	Consider h as random function

Table 3.1: Goals and Requirements for keyless cryptographic hash functions, presented for the hash function $h : \{0, 1\}^* \rightarrow \{0, 1\}^n$.

to ensure. Even seemingly minor differences between the security requirements for which the hash function was designed and tested, and the security properties required for the application, may result in a vulnerability. For example, Table 3.1 presents two integrity requirements, CRHF and SPR; we later show applications which are secure using a cryptographic hash function which satisfies the CRHF requirement, but may be vulnerable assuming ‘only’ the SPR requirement. Namely, even if the definitions may appear similar, it is still important to use the correct definition; the differences can be meaningful and even critical. Table 3.1 also includes the *random oracle model (ROM)*, which we discuss (in Section 3.6). In the ROM, we analyze the security of a system using cryptographic hash functions *as if* we use a random function (from binary-strings to n -bit binary strings).

Here is an exercise which may strengthen your intuitive understanding of the different security requirements in Table 3.1.

Exercise 3.2 (Examples of (insecure, simple) hash functions). *Let $h(x) = x \bmod 2^n$, $h'_k(x) = k + x \bmod 2^n$, and $h''(x) = x^2 \bmod 2^n$, all computed by considering their inputs as integers in binary representation. To avoid confusion, we denote numbers in binary representation by subscript of 2, e.g., 10000_2 is the number 16 in decimal, i.e., 2^4 , and similarly, $100_2 = 4 = 2^2$. Notice that h' is keyed, while h and h'' are keyless.*

1. For $n = 4$, compute $h(11010_2)$, $h(10101010_2)$, $h'(11010_2)$ and $h(10101010_2)$.
Note: inputs are binary strings, and should be viewed as integers in binary representation, which we denote with the subscript 2.
2. Show that $h(x) = x \bmod 2^n$ is not a CRHF, SPR, OWF or randomness extractor, based on the abridged descriptions in Table 3.1.
3. Repeat for $h'_k(x) = k + x \bmod 2^n$.
4. Repeat for $h''(x) = x^2 \bmod 2^n$. Beware: the OWF property can be challenging.

Solution for the first three items:

- Recall that inputs are binary strings, and should be viewed as integers in binary representation; we write the outputs similarly.

$$\begin{aligned}
 h(11010_2) &= 11010_2 \bmod 2^4 = 1010_2 \\
 h(10101010_2) &= 10101010_2 \bmod 2^4 = 1010_2 \\
 h'(11010_2) &= 11010^2 \bmod 2^4 = (11010_2 \bmod 2^4)^2 \bmod 2^4 = \\
 &= (1010_2)^2 \bmod 2^4 = (1000_2 + 10_2)^2 \bmod 2^4 = \\
 &= (1,000,000_2 + 10000_2 + 10000_2 + 100_2) \bmod 2^4 = 100_2 \\
 h'(10101010_2) &= \dots = 100_2 \text{ (similarly)}
 \end{aligned}$$

- Show that $h(x) = x \bmod 2^n$ is *not* a CRHF, SPR, OWF or randomness extractor.

- SPR and CRHF: We show that h is not a CRHF or an SPR hash function, by showing a collision for any given input x . Specifically, let $x' = x + 2^n$. Clearly $x' \neq x$, and yet $h(x') = (x + 2^n) \bmod 2^n = x \bmod 2^n = h(x)$, namely, x' is a collision (second preimage) with x .
- OWF: We next show that h is not a one-way function (OWF). Specifically, given $h(x)$ for any preimage x , let $x' = h(x)$; clearly:

$$\begin{aligned}
 h(x') &= x' \bmod 2^n = h(x) \bmod 2^n \\
 &= (x \bmod 2^n) \bmod 2^n = x \bmod 2^n = h(x)
 \end{aligned}$$

Namely, x' is a preimage of $h(x)$, and hence h is not a OWF.

- Finally we show that h is not a *randomness extractor* hash function. Specifically, let $r \xleftarrow{\$} \{0,1\}^n$ be a random n bit string, and let $x = r \# 0$, i.e., let x be an $n+1$ -bits binary string, whose least significant bit is zero and the other bits selected randomly. The value of $h(x) = x \bmod 2^n$ is the same as the n least-significant bits of n , and in particular, the least significant bit of $h(x)$ is zero. Hence, $h(x)$ is easily distinguishable from a random n bit string.

- Show that $h'_k(x) = k + x \bmod 2^n$ is *not* a CRHF, SPR, OWF or randomness extractor. Recall that the key k is known to the adversary (not secret). This makes it easy to adapt the solutions of the previous item. Specifically:

SPR and CRHF: The same collision $x' = x + 2^n$ applies here too. Clearly $x' \neq x$, and yet for every key k holds: $h_k(x') = k + (x + 2^n) \bmod 2^n = k + x \bmod 2^n = h_k(x)$, namely, x' is a collision (second preimage) with x .

OWF: We next show that h_k is not a one-way function (OWF), i.e., given $h_k(x)$ (and k), we can find x' such that $h_k(x') = h_k(x)$. Notice

that we are given the key - in our definitions of keyed hash function, the key is known to the attacker, i.e., not a secret. Specifically, given $h_k(x)$ for any preimage x , and the key k , let $x' = h_k(x) - k$; clearly:

$$\begin{aligned} h_k(x') &= x' \bmod 2^n = h_k(x) - k \bmod 2^n \\ &= (k + x \bmod 2^n) \bmod 2^n = k + x \bmod 2^n = h_k(x) \end{aligned}$$

Namely, x' is a preimage of $h_k(x)$, and hence h is not a OWF.

Randomness extractor hash: Finally we show that h_k is not a randomness extractor hash function. Specifically, let $x = r \# 0^n$, where r is a random n bit string; for simplicity, assume that the key k is also n bits long. Then $h_k(x) = (r \# 0^n) + k \bmod 2^n = k$, which is obviously not a random string.

□

3.1.3 Applications of cryptographic hash functions

The broad security requirements of cryptographic hash functions facilitate their use in many systems and for an extensive variety of applications. These different applications and systems rely on different security requirements. As in any security system, it is important to identify the exact security requirements and assumptions; however, published designs, and even standards, do not always define the requirements precisely. Important applications of cryptographic hash functions include:

Integrity of a string or a set : the hash $h(m)$ is a short *digest* of a typically much longer string m , which allows validation of the integrity of m . Hash functions are also used in the construction of accumulator *schemes*, e.g., the *Merkle tree* design; accumulators also produce a digest, but of an ordered or unordered set of strings, rather than of a single string. We discuss accumulators in Section 3.7.

Hash-then-Sign : Signature schemes are usually defined with *Fixed Input Length (FIL)*, typically quite limited (e.g., < 1024 bits). To sign longer messages, we apply the FIL signing function to the digest $h(m)$ of the message m being signed; this is called the *Hash-then-Sign* paradigm. See subsection 3.2.6.

Improved login mechanisms : Hash functions are used to improve the security of password-based login authentication, in several ways. The most widely deployed method is using a *hashed password file*, which makes exposure of the server's password file less risky - since it contains only the hashed passwords. Another approach is to use a hash-based *one-time password*, which is a random number allowing the server to authenticate the user, with drawbacks of single use and having to remember or have this random number. We discuss login mechanisms in Chapter 9, and some basic aspects in subsection 3.4.1.

Proof-of-Work : cryptographic hash functions are often used to provide *Proof-of-Work (PoW)*, i.e., to prove that an entity performed a considerable amount of computation. This is used by Bitcoin and other cryptocurrencies, and for other applications. See subsection 3.10.3.

Key derivation and randomness generation : hash functions are used to extract pseudorandom bits, given input with ‘sufficient randomness’. In particular, this is used to derive secret shared keys. See §3.5.

3.1.4 Standard cryptographic hash functions

Due to their efficiency, simplicity and wide applicability, cryptographic hash functions are probably the most commonly used ‘cryptographic building blocks’, as discussed in the cryptographic building blocks principle (Principle 8). This implies the importance of defining and adopting standard functions, which can be widely evaluated for security - mainly by cryptanalysis - and the need for definitions of security.

There have been many proposed cryptographic hash functions; however, since security is based on failed efforts for cryptanalysis, designers usually avoid less-well-known (and hence less tested) designs. The most well-known cryptographic hash functions include the MD4 and MD5 functions proposed by RSA Inc., the SHA-1, SHA-2³ and SHA-3 functions standardized by NIST ([95, 266]), the RIPEMD and RIPEMD-160 standards, and others, e.g., BLAKE2. Several of these, however, were already ‘broken’, i.e., shown to fail some of the security requirements. In particular, *collisions* - and specifically, *chosen-prefix collisions* - were found for RIPEMD, MD4 and MD5 in [301], and later also for SHA-1 [216]; see subsection 3.3.1. As a result, these functions should be avoided and replaced, at least in applications which depend on the collision-resistance property.

Existing standards define only *keyless* cryptographic hash functions. However, as we later explain, there are strong motivations to use *keyed cryptographic hash functions*, which use a non-secret key. In particular, collision-resistance cannot be achieved by any keyless function.

3.2 Collision Resistant Hash Function (CRHF)

3.2.1 Keyless Collision Resistant Hash Function (Keyless-CRHF)

A keyless hash function $h : \{0,1\}^* \rightarrow \{0,1\}^n$ maps unbounded length binary strings $m \in \{0,1\}^*$, to their n -bit *digest* $h(m) \in \{0,1\}^n$. Since the input domain is the (infinite) set of all binary strings, and the range is finite ($\{0,1\}^n$), it follows that there are infinitely many *collisions*, i.e., messages $m \neq m'$ s.t. $h(m) = h(m')$. Indeed, even if we limit ourselves to the input set of messages

³The SHA-2 specifications defines six variants for SHA-2, with digest lengths of 224, 256, 384 or 512 bits; these variants are named SHA-224, SHA-256, SHA-384, SHA-512, SHA-512/224, and SHA-512/256.

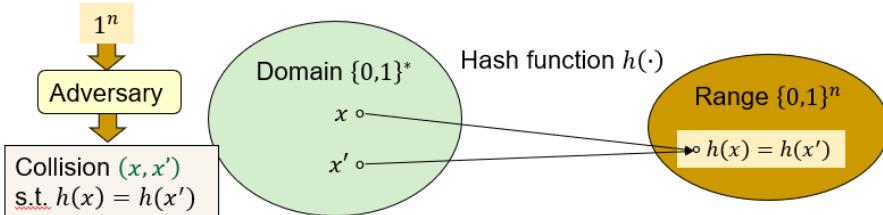


Figure 3.5: Keyless collision resistant hash function (CRHF): it is infeasible to efficiently find a collision, i.e., a pair of inputs $x, x' \in \text{Domain}$, which are mapped by the hash function h to the same output, $h(x) = h(x')$, except with negligible probability.

$m \in \{0,1\}^{n+1}$, the number of messages is 2^{n+1} and the number of digests is only 2^n , so clearly at least half (2^n) of the inputs must result in a collision. Namely, *collisions are common* - in *every* hash function.

However, for large digest length n , it is conceivable that *finding* a collision may be hard. Intuitively, we say that a hash function is *collision resistant*, if it is hard to find *any* collision, as illustrated in Figure 3.5. The definition follows; notice, that in this and (most) other definitions of keyless hash function, we actually view the hash function as if it is defined for different digest lengths n , allowing us to discuss the computational complexity as a function of n . For more precise definitions that explicitly express the digest length n as a parameters, see, e.g., [140].

Definition 3.1 (Keyless Collision Resistant Hash Function (CRHF)). *A keyless hash function $h(\cdot) : \{0,1\}^* \rightarrow \{0,1\}^n$ is collision-resistant if for every efficient (PPT) algorithm \mathcal{A} , the advantage $\varepsilon_{h,\mathcal{A}}^{\text{CRHF}}(n)$ is negligible in n , i.e., smaller than any positive polynomial for sufficiently large n (as $n \rightarrow \infty$), where:*

$$\varepsilon_{h,\mathcal{A}}^{\text{CRHF}}(n) \equiv \Pr[(x, x') \leftarrow \mathcal{A}(1^n) \text{ s.t. } (x \neq x') \wedge (h(x) = h(x'))] \quad (3.1)$$

Where the probability is taken over the random coin tosses of \mathcal{A} .

Let us define h_{sum} , a simple example of an *insecure* hash function, which is handy to give examples of the different cryptographic hash function definitions - all of which, h_{sum} fails to satisfy.

It is convenient to define the h_{sum} function for the input domain $\{0, 1, \dots, 9\}^*$, i.e., strings of digits. The h_{sum} function repeatedly sums up the digits of its input, until obtaining only one digit, which is the output. For example, $h_{\text{sum}}(13) = 4$, $h_{\text{sum}}(345) = 3$ and $h_{\text{sum}}(5) = 5$. This function is probably familiar from elementary school, where it is introduced as a way to check if a number divides by three (or by nine). We next define h_{sum} precisely and find collisions in it.

Example 3.1 (The h_{sum} hash function and collisions in it). Let $x \in \{0, 1, \dots, 9\}^*$, i.e., a string of digits. We define $h_{sum}(x)$ as follows:

$$h_{sum}(x) = \begin{cases} x & \text{if } x < 10 \\ h_{sum}\left(\sum_{i=1}^{|x|} x[i]\right) & \text{otherwise} \end{cases}$$

Let us show that h_{sum} is not a CRHF. This is easy; in fact, given any integer $x > 0$, let $x' = 10 \cdot x$. Then $x \neq x'$, yet $h_{sum}(x') = h_{sum}(x)$ - i.e., h_{sum} is not a CRHF.

So, how do we find a CRHF? Possibly surprisingly, we next show that we can; namely, we show that *there exists no keyless CRHF*.

3.2.2 There are no Keyless CRHFs!

Standard cryptographic hash functions, discussed in subsection 3.1.4, are all *keyless*; and practical deployment almost always use these designs. By now, the readers should not be surprised to learn that none of these were *proven* secure; we discussed in subsection 2.7.4 the fact that ‘real’, unconditional proofs of security for (most) cryptographic schemes would imply that $P \neq NP$, and, therefore, would be major news. Cryptographic hash functions are among the basic cryptographic building blocks, which are typically validated by accumulated evidence of failed attempts to cryptanalyze them.

However, the following lemma may be surprising: *all keyless hash functions fail* to satisfy the keyless-CRHF definition (Definition 3.1) - namely, a keyless-CRHF - using this definition - simply does not exist. We present and prove this, and then discuss the implications.

The reader is quite right to be suspect a ‘trick’ here - after all, we just explained that all *standard* cryptographic hash functions are keyless! Well, that is correct: the proof uses a ‘trick’ to show that there is an efficient attacker that can find a collision in the keyless hash function. The proof shows that for any given keyless hash function h , there *exists* an efficient adversarial algorithm \mathcal{A}_h , that outputs a collision for h , i.e., a pair (m, m') s.t. $m \neq m'$ but $h(m) = h(m')$. Furthermore, \mathcal{A}_h is not just efficient in n : its time complexity is basically the time required to print out the collision. In fact, printing the collision is basically the only thing that \mathcal{A}_h does. And note that \mathcal{A}_h does not just succeeds with a ‘significant’ probability; *it always succeeds*. Namely, h is very very far from the requirements of Definition 3.1!

Would you like to see the trick - or did you already figure it out? You may have, since we have essentially already did the trick - ‘hidden in plain sight’ exactly in the paragraph above. Can you find it? Try to find it, before you read the proof and the explanation of the trick.

Lemma 3.1 (Keyless CRHF do not exist.). *There is no keyless CRHF hash function $h : \{0, 1\}^* \rightarrow \{0, 1\}^n$.*

Proof: Given $h(\cdot)$, we prove that there exists an efficient adversary algorithm \mathcal{A}_h that *always* finds a collision, i.e. $\varepsilon_{h,\mathcal{A}}^{CRHF}(n) = 1$ - clearly showing that $h(\cdot)$ does not satisfy the definition of a keyless CRHF.

Recall that, since the domain of h is infinite while the range is the finite set $\{0,1\}^n$, then h must *have* collisions, i.e., pairs of binary messages $m \neq \hat{m}$ s.t. $h(m) = h(\hat{m})$. Let m, \hat{m} denote one such collision. It does not matter which collision we pick or *how* do we pick the collision.

The adversary \mathcal{A}_h simply outputs the collision (m, \hat{m}) , i.e., $\mathcal{A}_h(1^n) = (m, \hat{m})$. Obviously, \mathcal{A}_h is efficient, and *always* outputs a collision. Therefore, $h(\cdot)$ is not a keyless CRHF (as defined in Definition 3.1). \square

The ‘trick’ was that we proved that such an attacker \mathcal{A}_h exists - but in a *non-constructive* way, i.e., we did not *present* such an adversary or showed an efficient way to find it. We only showed that such an adversary *exists*. This shows that *there is no keyless CRHF*, as defined in Definition 3.1. Of course, there may be some other reasonable notion for collision resistance for keyless hash, for which the lemma does not apply. Indeed, we later define *Second-Preimage Resistant (SPR)* hash, which is essentially a weaker collision-resistance property.

In this textbook, for simplicity, we usually use keyless hash functions, often assuming collision resistance, i.e., the keyless hash function is a cryptanalysis-resistant CRHF. In the constructions and designs we discuss, it is not too hard to add the ‘missing’ keys, when desired (e.g., for provably secure reductions). Namely, we use ‘keyless CRHFs’ as a convenient simplification. A justification may be that if the system using the hash is insecure, then we may be able to use the attack to find the collision - which seems hard, as cryptanalysts failed so far to find such collision. Another justification is that in any practical implementations, the output length is fixed, while we only discuss asymptotic security definitions. In fact, many cryptographic designs use an even stronger simplification - the random oracle model (ROM), which we discuss in §3.6.

Another approach is to design the application without assuming a CRHF at all, and instead, rely on other properties, which may exist for keyless hash. One especially-relevant property is *second-preimage resistance (SPR)*, which is, essentially, a weaker form of collision resistance. Of course, care must be taken to ensure that SPR is really sufficient for the application; there could be subtle vulnerabilities due to use of SPR in an application requiring ‘real’ collision-resistance. We discuss SPR in Section 3.3.

A final alternative is to use a *keyed CRHF* instead of keyless CRHF. Considering that existing standards define only keyless hash, a common approach is to use construction of a keyed hash from a keyless hash. Often, this would be the *HMAC* construction, originally designed as a construction of a MAC function from hash. We discuss HMAC in subsection 4.6.3.

3.2.3 Keyed Collision Resistance

We next discuss *keyed collision resistant* hash functions (*keyed CRHF*). The definition for *keyed* CRHF seems very similar; the only difference is that the

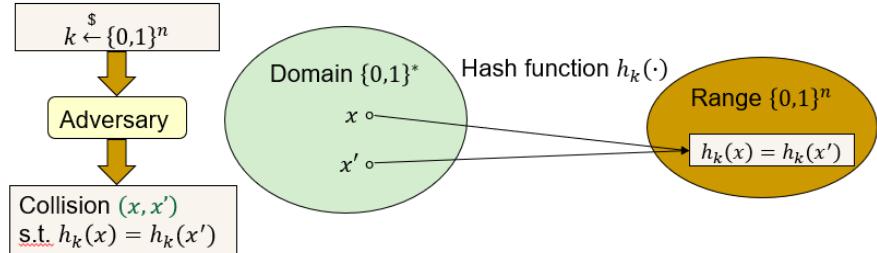


Figure 3.6: Keyed collision resistance hash function (CRHF): given random key k , it is hard to find a collision for h_k , i.e., a pair of inputs $x, x' \in \{0, 1\}^*$ s.t. $h_k(x) = h_k(x')$.

probability is also taken over the key, and the key is provided as input to the adversary. Recall that, for simplicity, we use n as the length of both the digest and the key; hence, we do not need to provide n as an additional input (since it is equal to the key length). We next define keyed collision resistance, which we illustrate in Figure 3.6. Recall that for keyed cryptographic hash functions, we assume, for simplicity, that n denotes both the length of the key and the length of the digest length, i.e., for every message $m \in \{0, 1\}^n$ holds: $|k| = |h_k(m)| = n$.

Definition 3.2 (Keyed Collision Resistant Hash Function (CRHF)). *Consider a keyed hash function $h_k(\cdot) : \{0, 1\}^n \times \{0, 1\}^* \rightarrow \{0, 1\}^n$, defined for any $n \in \mathbb{N}$. We say that h is collision-resistant if for every efficient (PPT) algorithm \mathcal{A} , the advantage $\varepsilon_{h, \mathcal{A}}^{CRHF}(n)$ is negligible in n , i.e., $\varepsilon_{h, \mathcal{A}}^{CRHF}(n) \in NEGL(n)$, where:*

$$\varepsilon_{h, \mathcal{A}}^{CRHF}(n) \equiv \Pr_{k \leftarrow \{0,1\}^n} [(x, x') \leftarrow \mathcal{A}(k) \text{ s.t. } (x \neq x') \wedge ((h_k(x) = h_k(x')))] \quad (3.2)$$

Where the probability is taken over the random coin tosses of the adversary \mathcal{A} and the random choice of k .

Let us now define a simple, insecure *keyed* hash function - specifically, h_k^{sum} - essentially, a keyed-version of the h_{sum} hash function (Example 3.1).

Definition 3.3 (The keyed h_k^{sum} (insecure) hash function.). *Let $k, x \in \{0, 1, \dots, 9\}^*$. Then we define $h_k^{sum}(x)$ as follows:*

$$h_k^{sum}(x) = h_{sum}(k || x)$$

The following exercise uses the simple h_k^{sum} hash function to demonstrate the CRHF definition.

Exercise 3.3. Show that h_{sum}^k is not a keyed CRHF.

Hint: see Example 3.1 for guidance. \square

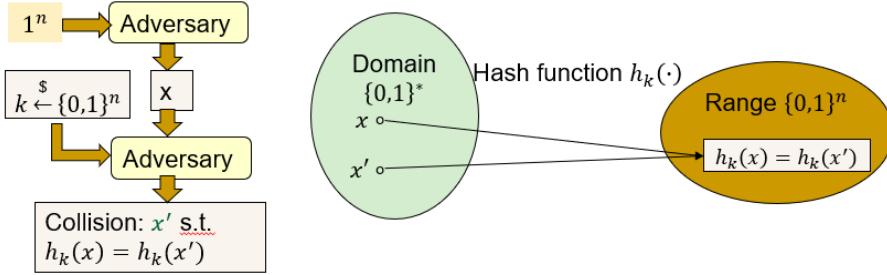


Figure 3.7: Target collision resistant (TCR) hash function: adversary cannot find *target* x , to which it would be able to find a collision x' , once it would be given the random key k .

Target Collision Resistant (TCR) vs. ACR / Keyed CRHF. Definition 3.2 uses the term *keyed CRHF*, following Damgård [93]. Another term for this definition is *any collision resistance (ACR hash)*, proposed by Bellare and Rogaway in [38]. They preferred this term, to emphasize that this definition allows the attacker to choose the specific collision as function of the key, since the key is given to the attacker *before* the attacker outputs the entire collision (both x and x' s.t. $h_k(x) = h_k(x')$).

Bellare and Rogaway preferred to use the term ACR to the term ‘keyed CRHF’, to emphasize the difference from a weaker notion of collision-resistance that they (and we) call⁴ *Target Collision Resistant (TCR)* hash. The term TCR emphasizes that, to ‘win against’ the TCR definition, the attacker has to first *select the target* x , i.e., one of the two colliding strings, *before* it receives the (random) key k . Only *then* the attacker is given the random key k , and has to output the colliding string x' s.t. $h(x) = h(x')$. Intuitively, this makes sense: it seems that on most applications, a collision between two ‘random’ strings x, x' may not help the attacker; the attacker often needs to match some specific ‘target’ string x . The TCR definition still allows the attacker to choose the target - but at least not as a function of the key!

We next define target collision resistance, which we illustrate in Figure 3.7.

Definition 3.4 (Target collision resistant (TCR) hash). *A keyed hash function $h_k(\cdot) : \{0,1\}^* \times \{0,1\}^* \rightarrow \{0,1\}^*$ is called a target collision-resistant (TCR) hash, if for every efficient (PPT) algorithm \mathcal{A} , the advantage $\varepsilon_{h,\mathcal{A}}^{TCR}(n)$ is negligible in n , i.e., smaller than any positive polynomial for sufficiently large n (as $n \rightarrow \infty$), where:*

$$\varepsilon_{h,\mathcal{A}}^{TCR}(n) \equiv \Pr_{k \leftarrow \{0,1\}^n} \left[\left\{ \begin{array}{l} x \leftarrow \mathcal{A}(1^n); \\ x' \leftarrow \mathcal{A}(x, k) \end{array} \right\} \text{ s.t. } (x \neq x') \wedge (h_k(x) = h_k(x')) \right] \quad (3.3)$$

Where the probability is taken over the random coin tosses of \mathcal{A} and the random choice of k .

⁴TCR is a different name for the notion, which was earlier defined by Naor and Yung in [243], but with a different name: *universal one-way hash functions*.

Clearly, every keyed CRHF, i.e., Any-Collision-Resistant (ACR) hash, is also a Target-Collision-Resistant (TCR) hash function: if there is some value x with whom the adversary can find a collision for a random key (with high probability), then surely the adversary can find *some* collision for a random key (with high probability) - e.g., collision with the same x . However, the reverse appear, intuitively, unlikely: maybe it is possible to find a collision once given the key k , but not with a pre-committed value x ? The following *counterexample exercise/argument* shows that indeed, this may be possible, i.e., there may be a keyed hash which is TCR but not a CRHF (not an ACR hash).

Exercise 3.4. Let $h_k(\cdot)$ be TCR hash function. Show a keyed hash function $h'_k(\cdot)$ which is also TCR but not a keyed CRHF (i.e., no an ACR hash).

Solution: Recall that the length of the key is n bits. Define $h'_k(x)$ as follows:

$$h'_k(x) = \{0^n \text{ if } x[1:n] = k, \text{ otherwise } h_k(x)\}$$

Namely, if the n most significant bits of the input x are the same as k , then $h'_k(x) = 0^n$, otherwise, $h'_k(x) = h_k(x)$. Clearly, for any key k holds $h'_k(k) = h'_k(k||0) = 0^n$. Recall that in the definition of a keyed CRHF (Definition 3.2), the adversary \mathcal{A} is given the key k . Hence it is easy for \mathcal{A} to output a collision, e.g., the pair (x, x') where $x = k$ and $x' = k + 0$. Namely, h' is not a CRHF.

It remains to show that h' is a TCR hash. In the TCR test, the key k is also chosen randomly; the difference is that the key is given to the adversary \mathcal{A} only when \mathcal{A} selects the *second (colliding)* input, x' , and not earlier, when \mathcal{A} selects the first input, x . Since k is chosen uniformly at random, there is probability $\frac{1}{2^n}$ for \mathcal{A} to pick x whose n^{th} most significant bits are the same as k ; and $\frac{1}{2^n}$ is negligible (in n). Therefore, with overwhelming probability, \mathcal{A} selects x whose n^{th} most significant bits are *not* the same as k , and therefore, $h'_k(x) = h_k(x)$.

Assume, to the contrary, that, given k , the adversary \mathcal{A} find a collision to h' , i.e., $x' \neq x$ such that $h'_k(x) = h'_k(x')$. However, with overwhelming probability $h'_k(x) = h_k(x)$. Also, the n^{th} most significant bits of x' cannot be the same as k (or $h'_k(x')$ would be 0^n). Hence, $h'_k(x') = h_k(x')$. Namely, we have found a collision for h : a pair $x \neq x'$ such that $h_k(x) = h'(x')$, in contradiction to h being a TCR. Hence, an adversary \mathcal{A} that finds a target-collision for h' , would also find such collision for h . Since h is a TCR, finding such collisions is infeasible; hence, h' must be also a TCR hash. \square

If possible, it is preferable to design protocols which use a TCR keyed hash, rather than protocols requiring the use of a keyed CRHF (ACR hash). That is since we have seen that a keyed hash h may satisfy the (weaker) TCR requirement, but not the (stronger) ACR (keyed CRHF) requirement. Furthermore, for the protocol to rely on a keyed CRHF rather than TCR, we must use sufficient digest length to protect against the *birthday attack*, which we discuss in the following subsection.

3.2.4 Birthday and exhaustive attacks on CRHFs

Our definitions of collision-resistance (for both keyless and keyed hash, Definition 3.2 and Definition 3.1, respectively) place two significant requirements on the attacker. First, the attacker must find collisions using *only polynomial time (PPT)*. Second, the attacker must succeed to find a collision with *non-negligible probability*. Let us explain why, without these requirements on the attacker, we cannot hope to achieve collision resistance. Both arguments hold for both keyless and keyed CRHFs; for simplicity, we focus on the keyless hash case.

A PPT attacker can find collisions with exponentially-small probability in every hash function. Consider the same set X as before, and an algorithm that selects two random elements in X ; with small probability, this algorithm would output the collision $x \neq x'$ s.t. $h(x) = h(x')$. Therefore, the definitions allow the adversary to have negligible probability of finding a collision.

Attacker can find collisions in exponential time in every hash function. Consider a hash function $h : \{0, 1\}^* \rightarrow \{0, 1\}^n$, and a set X containing $2^n + 1$ distinct input binary strings. The output of h is the set of n -bits strings, which contains 2^n elements; hence, there must be at least two elements $x \neq x'$ in the set X , which collide, i.e., $h(x) = h(x')$. An adversary can surely compute $h(x)$ for each of the $2^n + 1$ elements in X , and find at least one such collision $h(x) = h(x')$. Of course, this attack requires $2^n + 1$ computations of the hash function, i.e., its runtime is exponential in n . Hence, the definitions require the adversary against a CRHF to run in time polynomial in n .

The birthday paradox and attack on collision resistance. The argument above required the adversary to compute $2^n + 1$ hash values, i.e., $O(2^n)$. We next show that actually, an adversary can find a collision, in any hash function, with only $O(2^{n/2}) = O(\sqrt{2^n})$ expected number of hash-computations, rather than $O(2^n)$.

This attack is often called the *birthday attack* since it is due to the so-called *birthday paradox*. Consider a room containing 23 persons. What is the probability of a *collision*, i.e., two people having birthday on the same day of the year? Many people expect this probability to be quite small, but in reality, the probability is about half. To understand why this is true, notice that when a person is added to a room currently containing i persons (with no collisions), the probability of a collision with *some* person in the room is $\frac{i}{356}$, not $\frac{1}{356}$.

More precisely, the expected number q of messages $\{m_1, m_2, \dots, m_q\}$ which should be hashed before finding a collision $h(m_i) = h(m_j)$ is approximately:

$$q \lesssim 2^{n/2} \cdot \sqrt{\frac{\pi}{2}} \lesssim 1.254 \cdot 2^{n/2} \quad (3.4)$$

Hence, to ensure collision-resistance against adversary who can do 2^q computations, e.g., $q = 80$ hash calculations, we need the digest length n to be

roughly twice that size, e.g., 160 bits. Namely, the *effective key length* of a CRHF is only $q = n/2$. This motivates the fact that hash functions often have digest length twice the key length of shared-key cryptosystems used in the same system. Using longer digest length and/or longer key length does not harm security, but may have performance implications.

Note that the birthday attack applies to both *keyed CRHF* and *keyless CRHF*; however, it does *not* apply to *Target Collision Resistant (TCR)* hash. Can you see why? Carefully compare Definition 3.2 vs. Definition 3.4, and you'll find out!

3.2.5 CRHF Applications (1): File Integrity

Collision resistance is a great tool for ensuring integrity. One common application is to distribute a (large) object m , e.g., a file containing the executable code of a program. Suppose the file m is distributed from its producer in LA, to a user or repository in Washington DC (step 1 in Fig. 3.8). Next, a user in NY is downloading the file from the repository (or peer user) in DC (step 2), receiving m' (which should be the same as m , of course). To validate the integrity of the received file m' , the user also downloads the digest $h(m)$ of the file, directly from the producer in LA (step 3), and then confirms that $h(m) = h(m')$ (by computing $h(m')$ locally). By downloading the large file m from the nearby DC rather than from LA, the transmission costs are reduced; by checking integrity using the digest $h(m)$, we avoid the concern that the file was modified in DC, or modified in transit between LA to DC or DC to NY.

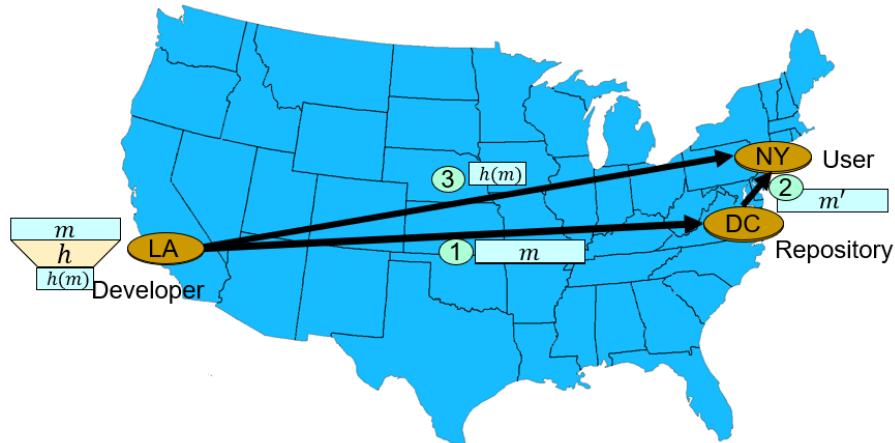


Figure 3.8: Example of use of hash function h to validate integrity of file m downloaded by a user in NY, from an untrusted repository in DC. To validate integrity, the user downloads the (short) digest directly from the website of the producer, in LA. This reduces network overhead - and load on the producer's website - compared to downloading the entire file from the producer's website.

This method of download validation is deployed manually, by savvy users, or in an automated way, by operating systems, applications or a script running within a browser [136].

A potential remaining concern is modification of the digest $h(m)$ received directly from producer in LA, by a *Man-in-the-Middle (MitM)*⁵ attacker. This may be addressed in different ways, including the use of a secure web connection for retrieving $h(m)$, as discussed in Chapter 7, receiving the digest from multiple independent sources, or receiving a signed digest. This last method is basically the same as the Hash-then-Sign methods that we discuss next.

3.2.6 CRHF Applications (2): Hash-then-Sign (HtS)

Collision-resistance is a powerful property; in particular, it facilitates one of the most important applications of cryptographic hash functions - the *Hash-then-Sign (HtS) paradigm*. The Hash-then-Sign paradigm is essential for efficient deployment of public-key digital signatures, which we introduced in subsection 1.4.1. We present constructions for signature schemes based on public key cryptosystems in Section 6.6; and in subsection 3.4.3 we discuss *one-time signatures* and present their constructions, based on one-way functions (OWFs). However, both approaches result in signatures for *limited-length inputs*; furthermore, extending the input length would significantly further increase the already high overhead of signature computation and validation (Table 6.1). Real applications always use, instead, the *Hash-then-Sign (HtS)* construction, which we discuss here. It can be applied to either keyed or keyless hash; we mostly focus on the keyless-hash variant (or ‘keyless HtS’).

The Hash-then-Sign solution applies a hash function $h(\cdot)$ to the ‘long’ message m , and signs the (short) output $h(m)$. Namely, given a signature scheme \mathcal{S} defined for domain $\{0, 1\}^n$ and a hash function h with domain $\{0, 1\}^*$ and range $\{0, 1\}^n$ (i.e., $h : \{0, 1\}^* \rightarrow \{0, 1\}^n$), we define the HtS scheme \mathcal{S}_{HtS}^h as follows:

$$\mathcal{S}_{HtS}^h.\mathcal{KG}(1^n) \equiv \mathcal{S}.\mathcal{KG}(1^n) \quad (3.5)$$

$$\mathcal{S}_{HtS}^h.\mathcal{Sign}_s(m) \equiv \mathcal{S}.\mathcal{Sign}_s(h(m)) \quad (3.6)$$

$$\mathcal{S}_{HtS}^h.\mathcal{Verify}_v(m, \sigma) \equiv \mathcal{S}.\mathcal{Verify}_v(h(m), \sigma) \quad (3.7)$$

The HtS scheme \mathcal{S}_{HtS}^h may be applied to any binary string, i.e., its domain is $\{0, 1\}^*$. The reader may confirm that it is a correct signature scheme over $\{0, 1\}^*$ (Exercise 4.21). Theorem 3.1 shows that if $h : \{0, 1\}^* \rightarrow \{0, 1\}^n$ is a *collision-resistant hash function (CRHF)* $h(\cdot)$, as in Definition 3.1, and \mathcal{S} is *existentially unforgeable signature* scheme over $\{0, 1\}^n$, then the HtS scheme \mathcal{S}_{HtS}^h is *existentially unforgeable signature* scheme over $\{0, 1\}^*$, i.e., applicable to arbitrary-length binary strings. Of course, the HtS method may fail if using an insecure hash function h ; see Exercise 4.24.

⁵Also called Monster-in-the-Middle

Given a keyless CRHF h , HtS would be secure. The keyless *Hash-then-Sign* construction uses a keyless CRHF $h : \{0, 1\}^* \rightarrow \{0, 1\}^n$. Given signature scheme (KG, S, V) whose input domain is (or includes) n -bit strings, i.e., $\{0, 1\}^n$, the keyless Hash-then-Sign signature of any message $m \in \{0, 1\}^*$ is defined as $S_s^h(m) \equiv S_s(h(m))$, where we use s for the private signing key of S (and of S_s^h).

We next show that if h is a Collision-Resistant Hash Function (CRHF), and assuming that the signature scheme (S, V) is secure (existentially unforgeable, see Definition 1.6), then (S^h, V^h) is also secure. Notice that Lemma 3.1 showed keyless CRHFs do not exist at all, which makes this theorem useless as a basis for proofs of security of applications using a keyless hash function. However, the theorem is still useful, for two reasons. First, it is similar (and simpler) to the similar theorems for *keyed* CRHFs. Second, it justifies, at least intuitively, the common use of the Hash-then-Sign paradigm applied to a keyless hash function h .

Theorem 3.1 (Keyless Hash-then-Sign would be secure (existentially unforgeable)). *Let (KG, S, V) be an existentially unforgeable signature scheme over the domain $\{0, 1\}^n$, and let $h : \{0, 1\}^* \rightarrow \{0, 1\}^n$ be a keyless CRHF function. Let \mathcal{S}_{HtS}^h be the Hash-then-Sign signature as defined in Equation 3.6. Then \mathcal{S}_{HtS}^h is an existentially unforgeable signature scheme over domain $\{0, 1\}^*$.*

Proof: Assume that the claim does not hold, i.e., that there is an efficient adversary $\mathcal{A} \in PPT$ s.t. $\varepsilon_{\mathcal{S}_{HtS}^h, \mathcal{A}}^{eu-Sign}(n) \notin NEGL(n)$, as defined in Equation 1.2. Namely, with significant probability, \mathcal{A} outputs a pair (m, σ) s.t. \mathcal{A} didn't provide m as input to the $\mathcal{S}_{HtS}^h.Sign_s(\cdot)$ oracle, yet $\mathcal{S}_{HtS}^h.Verify_v(m, \sigma)$. From Equation 3.7, $\mathcal{S}_{HtS}^h.Verify_v(m, \sigma) = \mathcal{S}.Verify_v(h(m))$. Let $\phi \equiv h(m)$; now, either there was another message m' s.t. $\phi = h(m')$ which \mathcal{A} did provide as input to the $\mathcal{S}_{HtS}^h.Sign_s(\cdot)$ oracle, or not. Let us consider both cases; at least one of the two must occur with significant probability.

If there is another message m' s.t. $\phi = h(m')$ which \mathcal{A} provided as input to $\mathcal{S}_{HtS}^h.Sign_s(\cdot)$, then the pair (m, m') , both of which produced by \mathcal{A} , is a collision for h . This should be impossible to efficiently find, since h is assumed to be a CRHF.

If there was no such m' , then we can use \mathcal{A} to construct an adversary \mathcal{A}' that will find forgery for the original signature scheme (KG, S, V) . The adversary \mathcal{A}' runs \mathcal{A} , and whenever \mathcal{A} makes an oracle query m , then \mathcal{A}' computes $\phi = h(m)$, makes query for $S_s(\phi)$ and returns the result to \mathcal{A} - obviously, this would be the expected value. Finally, when \mathcal{A} returns the forgery (m, σ) for \mathcal{S}_{HtS}^h , then \mathcal{A}' computes $h(m)$ and returns $(h(m), \sigma)$, which is the corresponding forgery for \mathcal{S} . The existence of such forgery contradicts the assumption that (KG, S, V) is an *existentially unforgeable signature* scheme. Hence, there is no such efficient adversary \mathcal{A} that ‘wins’ against \mathcal{S}_{HtS}^h , i.e., \mathcal{S}_{HtS}^h is an *existentially unforgeable signature* scheme over domain $\{0, 1\}^*$, as claimed. \square

Two Keyed-Hash HtS Constructions: *Keyed-HtS* and *TCR-HtS*. We now present *two Hash-then-Sign (HtS)* constructions from keyed-hash function. We begin with a very simple construction by Damgård [93], which we refer to as *Keyed-HtS* as it is based on the use of a *keyed-CRHF*. This construction is identical to the keyless Hash-then-Sign construction, except for the use of a keyed hash $h_k(\cdot) : \{0, 1\}^n \times \{0, 1\}^* \rightarrow \{0, 1\}^n$, defined for arbitrary key and digest length $n \in \mathbb{N}$. The hash key is selected once, during the key-generation process, and becomes part of the public verification key *and* of the private signing key. We define the *Keyed-HtS* construction \mathcal{S}_{HtS}^h as follows.

Definition 3.5 (The Keyed-HtS construction). *Given a signature scheme \mathcal{S} with domain $\{0, 1\}^n$ and a keyed hash (CRHF) $h_k(\cdot) : \{0, 1\}^* \times \{0, 1\}^* \rightarrow \{0, 1\}^*$, the Keyed-HtS signature using signature \mathcal{S} and keyed-hash h is defined as follows:*

$$\mathcal{S}_{HtS}^h.\mathcal{KG}(1^n) \equiv \left[\begin{array}{l} (s, v) \xleftarrow{\$} \mathcal{S}.\mathcal{KG}(1^n) ; \\ k \xleftarrow{\$} \{0, 1\}^n ; \\ \text{return } ((s, k), (v, k)) \end{array} \right] \quad (3.8)$$

$$\mathcal{S}_{HtS}^h.\mathcal{Sign}_{(s, k)}(m) \equiv \mathcal{S}.\mathcal{Sign}_s(h_k(m)) \quad (3.9)$$

$$\mathcal{S}_{HtS}^h.\mathcal{Verify}_{(v, k)}(m, \sigma) \equiv \mathcal{S}.\mathcal{Verify}_v(h_k(m), \sigma) \quad (3.10)$$

The *Keyed-HtS* construction requires the underlying keyed hash function to satisfy a relatively-strong requirement, namely, to make it infeasible to find *any* collision (what we referred to as ACR or keyed-CRHF hash).

Bellare and Rogaway show, in [38], the almost-as-simple *TCR-HtS* construction. The *TCR-HtS* construction requires only the weaker *target-collision resistant (TCR)* keyed-hash function. For discussion and comparison on these two definitions, see subsection 3.2.3; in particular, the generic *birthday attack* is applicable against the keyed-CRHF property, but not against the TCR property, hence a hash function may be a secure TCR hash, even when it uses significantly shorter digests (about half of the bits required for a secure keyed-CRHF).

The *TCR-HtS* construction is similar to the keyless Hash-then-Sign construction (and to the keyed-HtS construction); there are basically two differences. The first difference is obvious: we use a keyed hash $h_k(\cdot) : \{0, 1\}^n \times \{0, 1\}^* \rightarrow \{0, 1\}^n$, i.e., a hash function which receives two inputs binary strings, a key k and a message, and outputs a binary string. The second difference is in the construction; we select and transmit the hash-key *with each signature*. Namely, the hash key is selected, randomly, as part of each signing operation, and is sent together with the output of the underlying signature function. We define the *TCR-HtS* construction $\mathcal{S}_{TCR-HtS}^h$ as follows.

Definition 3.6 (The TCR-HtS construction). *Given a signature scheme \mathcal{S} with domain $\{0, 1\}^n$, defined for any integer n , and a keyed hash $h_k(\cdot) : \{0, 1\}^n \times \{0, 1\}^* \rightarrow \{0, 1\}^n$, the TCR-HtS signature using signature \mathcal{S} and keyed-hash h*

is defined as follows:

$$\mathcal{S}_{TCR-HtS}^h \cdot \mathcal{KG}(1^n) \equiv \mathcal{S} \cdot \mathcal{KG}(1^n) \quad (3.11)$$

$$\mathcal{S}_{TCR-HtS}^h \cdot \mathcal{Sign}_s(m) \equiv \left[\begin{array}{l} k \xleftarrow{\$} \{0,1\}^n; \\ \sigma \leftarrow \mathcal{S} \cdot \mathcal{Sign}_s(k \# h_k(m)) \\ \text{return } (k, \sigma) \end{array} \right] \quad (3.12)$$

$$\mathcal{S}_{TCR-HtS}^h \cdot \mathcal{Verify}_v(m, (k, \sigma)) \equiv \mathcal{S} \cdot \mathcal{Verify}_v(k \# h_k(m), \sigma) \quad (3.13)$$

Both the keyed-HtS and the TCR-HtS constructions, result in a secure, existentially-unforgeable signature scheme for unbounded input length - provided that the underlying hash function satisfies the required property. This property, however, is different between the two constructions. The keyed-HtS construction requires a keyed-CRHF, a relatively strong requirement, while the TCR-HtS construction only makes the (weaker) requirement of a TCR hash function.

Theorem 3.2 (Keyed Hash-then-Sign is secure (existentially unforgeable)). *Let (KG, S, V) be an existentially unforgeable signature scheme over the domain $\{0,1\}^n$, and let $h_k(\cdot) : \{0,1\}^n \times \{0,1\}^* \rightarrow \{0,1\}^n$ be a keyed hash function. Let \mathcal{S}_{HtS}^h be the Keyed-HtS signature scheme, as defined in Equation 3.10, and let $\mathcal{S}_{TCR-HtS}^h$ be the TCR-HtS signature scheme, as defined in Equation 3.13; both schemes are defined for arbitrary-length input strings. Then:*

1. *If h is a keyed collision-resistant hash function (CRHF), then \mathcal{S}_{HtS}^h is an existentially unforgeable signature scheme.*
2. *If h is a Target Collision-Resistant (TCR) hash function, then $\mathcal{S}_{TCR-HtS}^h$ is an existentially unforgeable signature scheme.*

Proof: see [38, 93]. \square

Unfortunately, both of the constructions involve challenges for deployment, namely, may necessitate significant changes in existing systems designed for Hash-then-Sign using keyless hash. The keyed-HtS construction requires distribution of a longer public key, since the random hash key becomes part of the public key; and the TCR-HtS construction requires a longer *signature* (k, σ) , i.e., the signature has to include also the random key chosen for the hash function. Several papers propose alternative HtS constructions with provable security properties but without such deployment challenges, some of them also based on keyless hash functions; for example, see [258].

3.3 Second-preimage resistance (SPR) Hash Functions

The second property we introduce is *second-preimage resistance (SPR)*. We define SPR only for keyless hash functions, although it can also be defined for keyed hash [282].

Intuitively, a *Second-Preimage Resistance (SPR)* hash function h accepts one input, an arbitrary-length binary string $m \in \{0,1\}^*$, and outputs an n -bit

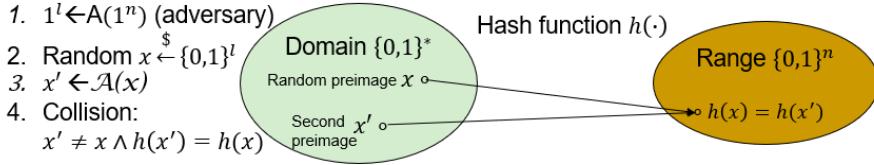


Figure 3.9: Second-preimage resistance (SPR): given keyless hash function $h : \{0,1\}^* \rightarrow \{0,1\}^n$, for any input length $l \geq n$, given a random *first preimage* $x \in \{0,1\}^l$, for $l \leftarrow \mathcal{A}(1^n)$, it is hard to find a collision with x , i.e., a *second preimage* $x' \in \{0,1\}^*$ s.t. $x' \neq x$ yet $h(x) = h(x')$.

long binary string $h(m) \in \{0,1\}^n$, and satisfies the *SPR property*. The SPR property means, intuitively, that an efficient (PPT) adversary \mathcal{A} has negligible probability, when given a random binary string x , to find a *collision* to x , i.e., a different string $x' \neq x$ which has the same hash: $h(x') = h(x)$. We refer to x' as the *second preimage*, and to x as the random (first) preimage, since they are both preimages of $h(x') = h(x)$.

We illustrate the SPR property in Figure 3.9. The reader will notice that we let the adversary select the length l of the random (and first) preimage x . By selecting the length of x first, we allow x to be a uniformly-random string from the (finite) set $\{0,1\}^l$; note that we can not select a uniformly-random element from an infinite set, i.e., we can't select a uniformly-random string from $\{0,1\}^*$ (see Section A.3). The set $\{0,1\}^l$ is a natural choice of a finite set, since it contains all binary strings of length l ; we can select a random string by flipping l fair coins - giving probability $\frac{1}{2^l}$ to each of the 2^l strings in the set $\{0,1\}^l$.

We let the adversary select l , which seems prudent - why limit the adversary to preimage of specific length? However, we must prevent the adversary from choosing l which would be ‘too long’, since the adversary, as an efficient (PPT) algorithm, is allowed runtime which is polynomial in the length of its inputs, which includes the l bit $x \xleftarrow{\$} \{0,1\}^l$. For example, if we let the adversary choose $l = 2^n$, then x would have 2^n bits, and, when given x and asked to choose x' , the adversary will be allowed runtime polynomial in 2^n , i.e., exponential in n , and to find, with high probability, a collision, e.g., by computing $h(x')$ for all values of $x' \in \{0,1\}^{n+1}$.

To ensure that the entire adversary’s runtime is polynomial in n , and in particular that l , the length of x , is polynomial in n , we require the adversary to output l in *unary*, i.e., as a string 1^l consisting of l bits whose value is 1. The definition follows.

Definition 3.7 (Second-preimage resistance (SPR) Hash Function). *A (keyless) hash function $h : \{0,1\}^* \rightarrow \{0,1\}^n$ is second-preimage resistant (SPR) if for every efficient (PPT) algorithm \mathcal{A} , the advantage $\varepsilon_{h,\mathcal{A}}^{SPR}(n)$ is negligible in n , i.e., smaller than any positive polynomial for sufficiently large n (as $n \rightarrow \infty$),*

where:

$$\varepsilon_{h,\mathcal{A}}^{SPR}(n) \equiv \Pr \left[\begin{array}{l} 1^l \leftarrow \mathcal{A}(1^n) \\ x \overset{\$}{\leftarrow} \{0,1\}^l \\ x' \leftarrow \mathcal{A}(x) \\ x \neq x' \wedge h(x) = h(x') \end{array} \right] \quad (3.14)$$

Where the probability is taken over the choice of x and the random coin tosses of \mathcal{A} .

SPR is sometimes referred to as *weak collision resistance*, and indeed, as the reader can prove, every CRHF is also an SPR hash function. However, Exercise 3.5 shows that there may be an SPR which is *not* a CRHF. Indeed, while subsection 3.2.2 shows that there is no keyless CRHF, it is possible, and commonly believed, that (keyless) SPR hash functions do exist. In practice, collision attacks, but *not second-preimage attacks*, are known against the SHA1 and MD5 standard hash functions.

Exercise 3.5. Let h be an SPR hash function. Use h to construct another hash function, h' , which you will show to be (1) an SPR (like h), but (2) not a CRHF.

Therefore, whenever possible, we should design protocols and systems to require only an SPR hash function, rather than a CRHF. Indeed, the SPR property suffices for some protocols and applications. For example, the SPR property suffices to authenticate the integrity of a file downloaded from an untrusted repository, whose hash is signed by the (trusted) developer, as in Figure 3.8.

However, other important applications *require* collision resistance and would be insecure if using a hash function which is only second-preimage resistant (SPR). Importantly, the SPR property is not sufficient for Hash-then-Sign (*Hts*), as we discuss in the next subsection.

Exercise 3.6. Explain, intuitively, why the SPR property suffices to authenticate the integrity of a file downloaded from an untrusted repository, whose hash is signed by the (trusted) developer; and why your explanation does not apply to the use of such hash for the Hash-then-Sign construction.

Although the SPR property suffices for some applications, and we know that there is no keyless hash function which is collision-resistant, it is still better to avoid any use of cryptographic hash functions for which collisions has been found. This protects against the common case, where a designer incorrectly believes that an SPR hash suffices, while the system is actually vulnerable to non-SPR collision attacks; and also protects against implementation errors which use the SPR hash function where the design called for a CRHF.

3.3.1 The Chosen-Prefix Collisions Vulnerability

Theorem 3.1 shows that Hash-then-Sign is secure when used with a CRHF. But would the weaker SPR property suffice to ensure security using the *Hash-then-Sign (HtS)* paradigm, i.e., using the (keyless) HtS construction of Equation 3.7? In this subsection we show that such *SPR-HtS*, i.e., use of a hash function which is second-preimage resistant but not collision resistant, may result in significant vulnerability.

Let us begin with an arguably less-significant vulnerability, showing that SPR-HtS would not ensure existential unforgeability. This requires only the following observation. Consider a (keyless) hash function, which is SPR but not CRHF. Namely, an adversary \mathcal{A} may know a collision $h(m) = h(m')$, although \mathcal{A} cannot efficiently find a collision to a randomly-chosen message m_R . Now, consider the HtS signature (Equation 3.6) over m :

$$\begin{aligned}\mathcal{S}_{HtS}^h.\mathcal{S}ign_s(m) &= \mathcal{S}.\mathcal{S}ign_s(h(m)) \\ &= \mathcal{S}.\mathcal{S}ign_s(h(m')) \\ &= \mathcal{S}_{HtS}^h.\mathcal{S}ign_s(m')\end{aligned}$$

We conclude that \mathcal{S}_{HtS}^h is not an existentially-unforgeable signature scheme (Definition 1.6).

However, could it be that \mathcal{S}_{HtS}^h is ‘secure enough’ for practical applications? Even if \mathcal{A} can find *some* collision $h(m) = h(m')$, for some ‘random’ strings m, m' , how would the attacker convince the signer to sign m , and why should the alternative message m' be of (significant) value to the attacker? In short: is there a clearly *realistic* attack, which may be possible against an SPR hash h (although this attack fails against a CRHF)? We next show that this is indeed the case by presenting such an attack, exploiting a realistic vulnerability: the *chosen-prefix vulnerability*, which we next define.

Definition 3.8 (The chosen-prefix collisions vulnerability). *Hash function h is said to have the chosen-prefix vulnerability if there is an efficient (PPT) collision-finding algorithm \mathcal{CF} , s.t. given a (prefix) string $p \in \{0, 1\}^*$, the algorithm \mathcal{CF} efficiently outputs, with high probability, a collision, i.e., a pair of strings $x, x' \in \{0, 1\}^*$, s.t. for any (suffix) string $s \in \{0, 1\}^*$ holds $h(p \# x \# s) = h(p \# x' \# s)$. Namely:*

$$(\forall p) \text{ w.h.p.: } \left\{ \begin{array}{l} (x, x') \leftarrow \mathcal{CF}(p) \text{ s.t. } (x \neq x') \wedge \\ (\forall s)(h(p \# x \# s) = h(p \# x' \# s)) \end{array} \right\} \quad (3.15)$$

Note that the fact that the collisions hold for any common suffix s is due to the fact that many keyless hashing functions have an iterative design. Due to this design, if there is a collision between prefix p between x and x' , i.e., $x \neq x'$ yet $h(p \# x) = h(p \# x')$, then there is also a collision for any suffix s between $p \# x \# s$ and $p \# x' \# s$. Namely, $(\forall s)h(p \# x \# s) = h(p \# x' \# s)$. In particular, the *Merkle-Damgård* construction, which is used by many hash functions, has this iterative design, and hence has this property. We discuss this construction in subsection 3.8.2.

Chosen-prefix attacks are practical. The chosen-prefix vulnerability is a realistic concern; in fact, such vulnerabilities were found for widely-used (at the time) standard hash functions including RIPEMD, MD4, and MD5 in [301], and later also for SHA1 [216], which all use the Merkle-Damgård construction. Due to this vulnerability, these hash functions are considered insecure and replaced, in new designs and, when possible, in existing systems, with cryptographic hash functions that are not known to have this or other vulnerabilities, such as SHA-2 and SHA-3 [95].

We next show how the chosen-prefix collisions vulnerability facilitates a realistic attack on the Hash-then-Sign paradigm. This attack allows an attacker to trick users into signing what appears to a third party to be a statement (e.g., money transfer) that the user never intended to sign. For more elaborate attacks, which allow also forgery of *public key certificates*, see Chapter 8 and [216, 301, 302].

Chosen-prefix attack on Hash-then-Sign: simplified version. We begin by presenting a simplified version of the chosen-prefix attack on the Hash-then-Sign paradigm. In this version, an attacker, say Mal, uses the chosen-prefix attack to find a pair of strings (x, x') , which would collide when appended to the (chosen) prefix ‘Pay \$’. Namely, the pair (x, x') would satisfy $x \neq x'$ and $h(\text{Pay \$} \# x \# s) = h(\text{Pay \$} \# x' \# s)$ (for any suffix s).

The main simplification we make is to assume that Mal can deposit a ‘payment order’ of the form ‘Pay \$’ $\#$ x $\#$ to Mal’, where x is a binary string interpreted as an integer. Obviously, reality is more complex, e.g., if the payment order is a document, the amount x should be also encoded in printable characters, typically in ASCII. Another simplification is to assume that, as binary numbers, $x << x'$.

Mal now sets-up an online shop and offers for sale an item whose market value is \$ y where $x < y << x'$. Mal offers the item for only \$ x - a real bargain! But what is *really* going to happen?

Alice comes along and happily buys the item, by sending to Mal a signed *payment order* to her bank, ready to be deposited at the bank. Namely, Alice sends to Mal the pair (PO, σ) where $\sigma = \text{Sign}_{A.s}(h(PO))$ and $PO = \text{‘Pay \$}x \text{ to Mal’}$.

Alice expect to be charged \$ x after Mal will deposit this signed payment order (PO, σ) . However, to her chagrin, she finds that she was charged $x' >> x$ - a significantly higher amount. Mal has tricked Alice, by depositing the *forged payment order* $PO' = \text{‘Pay \$}x' \text{ to Mal’}$, together with σ , Alice’s signature on PO ($\sigma = \text{Sign}_{A.s}(h(PO))$). The Bank will honor this transactions, and charge Alice by \$ x' , since σ is also a valid signature for PO' , as:

$$h(PO) = h(\text{Pay \$}x \text{ to Mal}) = h(\text{Pay \$}x' \text{ to Mal}) = h(PO')$$

Hence, the same signature generated by Alice, appears to the bank to be a valid signature over the ‘fake’ payment order PO' - and the bank transfers to Mal the larger amount $x' >> x$!

Recall now the simplifications we made in this description of the attack. In particular, banks will not accept a payment order where amounts are indicated as binary numbers; typically, the entire payment order should be encoded using printable characters, e.g., using ASCII code; more readable formats, such as PDF or HTML, are even more likely. We next discuss a more realistic variant of the attack, which works for payment orders encoded using PDF or HTML.

A more realistic Chosen-prefix Attack: Signing PDF documents

We now improve the chosen-prefix attack to allow forgery of signatures over documents formatted in ‘rich’ markup languages like PDF, postscript, and HTML. The attacker, Mal, exploits the fact that these (and similar) languages allow documents to contain conditional rendering statements, allowing the document to display different content depending on different conditions.

In the attack, Mal uses the conditional rendering capability, to create two documents D_1, D_M that have the same hash value, $h(D_1) = h(D_M)$, but when rendered by the correct viewer, e.g., PDF viewer, the two documents are rendered very differently. Namely, viewing D_1 , the reader displays text $t_1 = \text{‘Pay \$1 to Amazon’}$, while viewing D_M , the reader displays text $t_M = \text{‘Pay \$1,000,000 to Mal’}$. The rest of the contents, and even the details of the markup language used, do not materially change the attack, so we ignore them.

Mal creates these two documents as follows. First, the documents share common prefix and suffix: $D_1 = p \# x_1 \# s$, $D_M = p \# x_M \# s$.

The prefix p consists of headers and preliminaries as required by the markup language, e.g., `%PDF` for PDF, or `<!DOCTYPE html>` for HTML, followed by the ‘if’ statement in the appropriate syntax. Simplifying, let’s say that $p = \text{‘if } \cdot \cdot \cdot$.

Mal next applies the collision-finding algorithm \mathcal{CF} (Definition 3.8), to find collision for prefix p , namely: $(x_1, x_M) \leftarrow \mathcal{CF}(p)$. For every suffix s holds: $h(p \# x_1 \# s) = h(p \# x_M \# s)$.

To complete D_1 and D_M , Mal sets the suffix s to the string:

$$s \leftarrow \text{‘=} \# x_1 \# \text{‘then display’} \# t_1 \# \text{‘, else display’} \# t_M$$

Mal is now ready to launch the attack on Alice, similarly to the simplified attack above. Namely, Mal first sends D_1 to Alice, who views it and sees the rendering t_1 . Let us assume that Alice agrees to pay \$1 to Amazon, and hence signs D_1 , i.e., computes $\sigma = S_{A,s}(h(D_1))$ and sends (D_1, σ) back to Mal.

Mal forwards to the bank the modified message (D_M, σ) . The bank validates the signature, which would be Ok since $h(D_1) = h(p \# x_1 \# s) = h(p \# x_M \# s) = h(D_M)$. The bank then views D_M , and sees:

$$t_M = \text{‘Pay \$1,000,000 to Mal’}$$

As a result, the bank transfers one million dollars from Alice to Mal.

Of course, some work is required to actually deploy the above attack; in particular, it isn’t trivial to handle PDF files. The following exercise challenges the readers to find a similar attack against HTML files; this should not be too

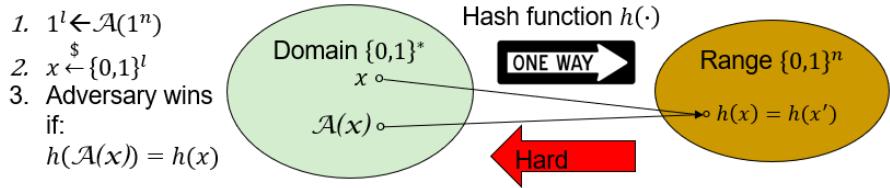


Figure 3.10: Intuition of the One-Way Function property, aka Preimage-Resistant hash function: given $h(x)$ for a (sufficiently long) random preimage x , it is infeasible to find x , or any other preimage x' of h , i.e., s.t.. $h(x') = h(x)$. Details in Definition 3.9.

difficult, and we hope would be fun and instructive. Your solution may assume that the browser displaying the HTML file supports JavaScript applets.

Exercise 3.7. Consider the hash function $h(x_1 \# x_2 \# \dots \# x_l) = \sum_{i=1}^l x_i \bmod p$, where each x_i is 64 bits and p is a 64-bit prime. (a) Is h an SPR hash function? CRHF? (b) Present a collision-finding algorithm \mathcal{CF} for h . (c) Create two HTML files D_1, D_M as above, i.e., s.t.. $h(D_1) = h(D_M)$, yet when they are viewed in a browser, they display texts t_1, t_M as above.

3.4 One-Way Functions, aka Preimage Resistance

The third security property we discuss for cryptographic hash functions is called *Preimage resistance* or *One-Way Function (OWF)*. Intuitively, h is a one-way function, if, given $h(x)$ for a (sufficiently long) random preimage x , it is infeasible to find either x or any other preimage x' of h , i.e., s.t.. $h(x') = h(x)$. This is illustrated in Fig. 3.10.

We prefer the term *One-Way Function (OWF)* to the term *preimage-resistant hash*, since OWF emphasizes the ‘one-way’ property: computing $h(x)$ is easy, but ‘inverting’ it to find x , or a colliding preimage x' s.t. $h(x') = h(x)$, is hard.

The definition follows. Note that the input x is selected as an $l > n$ bits string, where l is selected by the adversary (in unary). This selection process is similar to the one used in the definition of second-preimage resistant (SPR) hash, and for the same reasons: to ensure that the adversary is limited to runtime polynomial in n , and to allow random choice of x (from a finite domain).

Definition 3.9 (One-Way Function (OWF), aka Preimage resistance). *An efficient function h , with range $\{0,1\}^n$, is called preimage resistant, or a one-way function, if for every efficient algorithm $\mathcal{A} \in PPT$, the advantage $\varepsilon_{h,\mathcal{A}}^{OWF}(n)$ is negligible in n , i.e., smaller than any positive polynomial for sufficiently large n (as $n \rightarrow \infty$), where:*

$$\varepsilon_{h,\mathcal{A}}^{OWF}(n) \equiv \Pr \left[\begin{array}{c} \left(1^l \xleftarrow{\$} \mathcal{A}(1^n) ; x \xleftarrow{\$} \{0,1\}^l \right) \\ h(\mathcal{A}(x)) = h(x) \end{array} \right] \quad (3.16)$$

Where the probability is taken over the random coin tosses of \mathcal{A} and over the choice of $x \xleftarrow{\$} \{0, 1\}^l$.

The definition *requires* the input x to be selected as a *random* l bits string. This raises the following question: supposed h is a OWF (as in Definition 3.9), but we select input x as a random member of some other set (not $\{0, 1\}^l$), e.g., a random text from a collection of numerous texts. Is it possible that the attacker will be able to find x , or another preimage x' s.t. $h(x) = h(x')$?

We next present two important applications of one-way functions: *one-time passwords* and *one-time signatures*.

3.4.1 Using OWF for One Time Passwords (OTPw) Authentication

A one-way function can be used to facilitate *one time password* (*OTPw* or *OTP*)⁶ authentication. A one time password is a pair of two n -bit values for each user, say Alice: a *non-secret validation token* v_A , known to and used by the validating server, say Bob; and a *secret one time password*, p_A , sent by Alice to Bob when Alice wishes to authenticate herself to Bob.

The *one time password* design offers an important advantage over the use of ‘regular’ passwords: the authentication is secure, even if an attacker is able to obtain the value of the validation token v_A . This differs from conventional passwords, which must be kept secret. In Section 9.1 we discuss different password-based and other *login ceremonies*, several of which build upon the one time password design.

To implement one-time passwords, Alice selects a random secret n -bit one time password $p_A \xleftarrow{\$} \{0, 1\}^n$, and computes the non-secret validation token as $v_A = h(p_A)$. Alice shares the validation token v_A with Bob. Later, to prove her identity, or to make some other agreed-upon signal, Alice sends p_A to Bob. Bob can check if $v_A = h(p_A)$; if so, then Bob considers that Alice has authenticated successfully, i.e., that p_A is the correct one time password.

If h is a one way function (OWF), then, from Definition 3.9, an efficient (PPT) adversary cannot find the random preimage p_A , even if given the validation token v_A . Hence, the one time password scheme is secure: if Bob finds that $v_A = h(p_A)$, this should mean that p_A is the OTPw. If Alice kept the OWPw securely and only discloses it to authenticate herself to Bob, then Bob knows this also means that Alice has initiated the authentication (by disclosing p_A). Of course, an attacker which obtains the one time password p_A could try to abuse it; therefore, Bob should only accept p_A once, i.e., ‘one time’.

⁶Unfortunately, the acronym OTP is commonly and confusingly used for the two different notions of *one time pad* and *one time password*. We will use *OTPw* for one time passwords; we hope this may reduce confusion.

3.4.2 Hash Chain Authentication

The fact that each OTPw can only be used once, makes their use quite inconvenient: either Alice has to either establish a new validation token after using the previous OTPw, or setup multiple tokens in advance (and replenish them before they are all used up).

To improve the convenience of one time passwords, Lamport proposed the *hash chain* [203] scheme. A hash chain allows Bob to authenticate Alice multiple times, say m times, yet requires Bob to maintain only a single validation token, and requires Alice to remember only one secret.

Alice selects the secret, which we denote by x_0 , to be a random n -bits string. Then, Alice computes the a ‘chain’ of m values, $\{x_1, \dots, x_m\}$, by repeated hashing: $x_1 = h(x_0), \dots, x_m = h(x_{m-1})$.

Alice sends to Bob only the last value, x_m . To authenticate herself for the first time, Alice sense x_{m-1} ; Bob validates the incoming value x_{m-1} by hashing it and validating that $x_m = h(x_{m-1})$. Similarly, to authenticate herself for the i^{th} time, Alice sends x_{m-i} , and Bob validates that $x_{m-i} = h(x_{m-(i-1)}) = \dots = h^i(x_m)$, either using the previously-validated x_{i-1} or using the initial validation token x_m .

Many people like the simplicity and elegance of the hash chain design, leading to adoption and influencing other designs, some of which we discuss in Section 9.1. One well-known and widely used implementation is *S/Key*, which has a precise specification [157].

However, is hash chain authentication secure?

Intuitively, it may seem that the security of hash-chain follows from the same argument for the security of the one time password design, namely, since h is one-way, the preimage cannot be found. However, the one way function property is with respect to *uniformly-random preimage*, and in the hash chain, Alice selects, randomly, only the very first preimage (x_m). The other inputs (x_i for $i < m$) are the result of applying h , and therefore, may not be uniformly distributed. We give a concrete example in the following exercise.

Exercise 3.8. [*hash chain using OWF may be vulnerable*] Let $h : \{0, 1\}^* \rightarrow \{0, 1\}^n$, and define $g : \{0, 1\}^* \rightarrow \{0, 1\}^{2n}$ as:

$$g(x) = \begin{cases} 0^{2n} & \text{if } x = 0 \mod 2^n, \text{ otherwise } h(x) \# 0^n \end{cases} \quad (3.17)$$

Show that if h is a OWF, then g is also a OWF; yet, show that $f(x) = g(g(x))$ is not a OWF, and in particular, that a hash chain using g is completely insecure.

The counterexample function g of Exercise 3.8 is obviously a bad choice for a cryptographic hash function; in particular, it is trivial to find collisions in g . Specifically, $g(x) = 0^{2n}$ for every $x \in \{0, 1\}^n 0^n$; hence, g is not a CRHF.

More generally, the vulnerability in Exercise 3.8 exploited the fact that even when the input to a hash function is distributed randomly, the output may not be random. Suppose, however, that we use a hash function h that also has this property, i.e., if the input to h is selected randomly, then the output is also

random (or pseudo-random); for example, this holds if h is a permutation over $\{0, 1\}^n$, in addition to being a OWF. In this case, Exercise 3.9 shows that the hash-chain construction is secure.

Exercise 3.9. *Show that if h is a OWF and a permutation when applied to the domain $\{0, 1\}^n$, then a hash chain using h is secure. For simplicity, it suffices to prove this for a chain of length two (like in Exercise 3.8).*

However, even if h is a OWF and a permutation, the hash chain construction has another disadvantage: it is vulnerable to the *birthday attack*. Namely, the attacker computes the hash of some random m values, e.g., selecting a random x'_0 and computing the chain $x'_1 = h(x'_0), \dots, x'_m = h(x'_{m-1})$. As discussed in subsection 3.2.4, when m is about $2^{n/2}$, there is a high probability of a collision, i.e., $x'_i = x_j$ (for $(i, j \in [1, \dots, m])$). When the attacker detects the collision, i.e., when Alice authenticates for the $m - j$ time, and x_j becomes known to the attacker, then the attacker can forge the next authentication by sending the preimage known to it, x'_{i-1} . This will pass the validation, since $x_j = x'_i = h(x'_{i-1})$. Not only would the attacker succeed in authenticating as Alice; this would also prevent Alice from authenticating, since now Bob expects to receive the preimage of x'_{i-1} , which is not even known to Alice.

There are two ways to handle this concern. The first is to use digest length n which is large enough, and/or hash chains whose length m is not very large, so that $m << n/2$, making the birthday attack ineffective. The second is to change slightly the design, and use, in each link of the chain, a different hash function. One simple method, attributed to Micali and Leighton, is to compute x_i as: $x_i = h(x_{i-1} || i)$.

Note that in this modified design, the preimages are longer than the digest, i.e., h is not a permutation. Exercise 3.27 asks you to identify an alternative requirement on h which makes the modified design secure. See also Exercise 3.26 which extends Exercise 3.8 to investigate the case of functions which are both OWF and SPR, or both OWF and CRHF.

However, hash chain *would be secure* - if the function is not only one-way, but also a length-preserving *permutation*⁷. See Exercise 3.9.

3.4.3 Using OWF for One-Time Signatures

We next show how to use one-way functions to implement public key signatures - limited to signing only a single (one) time, i.e., a *one-time signature scheme*; this can be extended to a scheme allowing a limited number of signature operations. Note that our definitions of signature schemes (subsection 1.4.1) did not allow a restriction on the number of applications of the signature scheme, therefore, our discussion in this subsection will be informal; the interested reader may

⁷Note that h is definitely *not* a pseudorandom permutation (Definition 2.8); in fact, h does not even receive an input key! The adversary *knows* h , and would trivially be able to distinguish between oracle access to h , and oracle access to a random permutation.

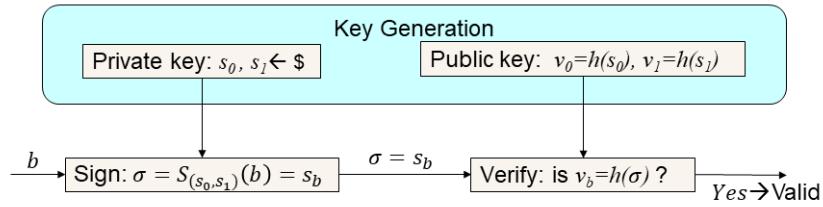


Figure 3.11: A one-time signature scheme, limited to a single bit (as well as to a single signature).

try to extend the definitions in subsection 1.4.1 to support limited number of applications.

In spite of their obvious limitation, one-time signatures have also important advantages in *security* and *performance*, making them a good choice for many applications. These advantages are in comparison to the widely-used public-key signatures schemes such as RSA and DSA, whose security is based on the hardness of factoring (RSA) or of computing discrete logarithms (DSA), which we discuss in Chapter 6.

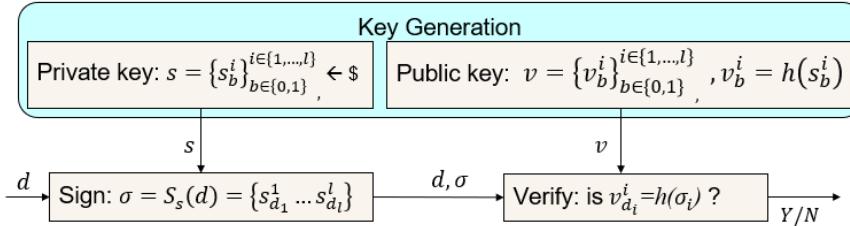
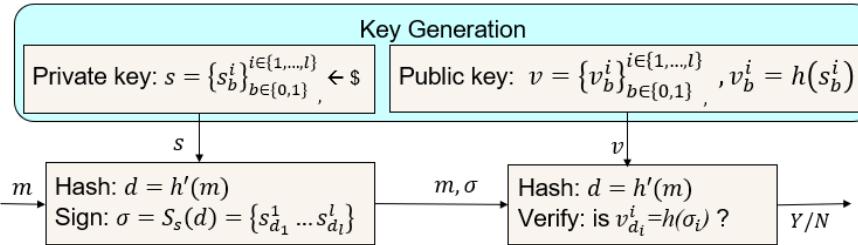
The main *security advantage* of one-time signatures is that they are not (too) vulnerable to the potential availability to an attacker of a *quantum computing* device. This is in contrast to factoring, discrete-logarithms and many other computational problems, which may be efficiently solved (broken) using an appropriate quantum computer. See Section 10.4. Signature schemes based on computational assumptions (factoring, discrete logarithm) are also subject to other possible improvements in the algorithms to solve these problems.

The *efficiency advantage* of one-time signatures is their low computational overhead. This advantage can further increased, if longer key size are used to ensure security against a future quantum computer (or algorithmic advances in solving the underlying ‘hard’ problem).

Note, however, that one-time signatures require rather long public keys and signatures. Of course, the overhead depends on the scheme used. We present a simple scheme; some other schemes require shorter public keys and signatures.

We present the construction in three steps, gradually improving performance. Figure 3.11 presents a one-time signature scheme which is defined only for the case of a single-bit message. This is a simple extension of one-time passwords. The private key s simply consists of two random strings s_0, s_1 , while the public key consists of the hashes of these strings: $v_0 = h(s_0), v_1 = h(s_1)$. To sign a bit b , we simply send $\sigma = s_b$; to validate incoming bit b and signature σ , we validate that $v_b = h(\sigma)$. Note that $v = (v_0, v_1)$ is not secret; only $s = (s_0, s_1)$ is secret - and we can disclose s_b upon signing bit b .

We next extend the scheme, to allow one-time signature of an l -bit string d , as illustrated in Figure 3.12. (We use the symbol d for ‘digest’, as this value will become digest of a longer message in the next extension.) This is simply application of the one-bit signature scheme of Figure 3.11, over each bit d_i of d ,

Figure 3.12: A one-time signature scheme, for l -bit string (denoted d).Figure 3.13: A one-time signature scheme, for variable-length message m , using 'Hash-then-Sign'.

for $i = 1, \dots, l$. We use now multiple key pairs (s_b^i, v_b^i) , where $b \in \{0, 1\}$ is the bit value (as before) and i is the index of the bit within d , and $v_b^i = h(s_b^i)$.

Finally, we extend the scheme further, to efficiently sign arbitrary-length inputs (VIL) string m . This extension, illustrated in Figure 3.13, simply applies the *Hash-then-Sign* paradigm. Namely, we first use a CRHF, which we denote by h' (since it does not have to be the same as h), to compute the l -bit digest d of the message: $d = h'(m)$. Then, we simply apply the scheme of Figure 3.12 to sign the digest d .

3.5 Randomness Extraction and Key Derivation Functions

Security and cryptography use randomness for many mechanisms, including encryption, key-generation, and challenge-response authentication. A source of true, perfect randomness is often unavailable; systems often rely on imperfect sources of randomness, such as measurements of delays of different physical actions. In this section, we discuss two related cryptographic tools to deal with this challenge: *randomness extractors* hash functions and *key derivation functions* (*KDFs*).

Intuitively, a *randomness extractor* (or simply an *extractor*) is a (keyless or keyed) hash function h , whose output $h(x)$ is pseudo-random⁸, provided that

⁸This definition only requires pseudorandom output, rather than true randomness; such

its input x has ‘sufficient randomness’, i.e., satisfies a specified *randomness assumption*. A keyed extractor also receives as input a non-secret random *salt*; if its input x has ‘sufficient randomness’, then its output $h_{\text{salt}}(x)$ is pseudo-random.

The randomness extraction property is more subtle and harder to define than the collision resistance, SPR and one-way properties, and not listed as a goal of standard hash functions. In subsection 3.5.1, we discuss the simple, keyless *Biased-Coin Extractor* proposed by Von Neumann, which ensure uniformly random output, provided that its input is a sequence of independently-sampled bits from some biased distribution. The cryptographic-theory literature mostly deals with the (significantly more complex) case of *generic extractors*, which only require the input to have sufficiently-high *min-entropy*, but this is beyond our scope; see [103, 235]. These works also focus on keyed extractors, which use random non-secret keys; this is since *keyless generic extractors do not exist* [103, 235].

Practical systems usually extract randomness using either keyless hash functions $h(x)$ or (keyed) *Key Derivation Functions (KDFs)* schemes, which we discuss in subsection 3.5.3.

3.5.1 Von Neumann’s Biased-Coin Extractor

We first discuss the classical *biased-coin* model proposed already in 1951 by Von Neumann [315], one of the pioneers of computer science. In the Von Neumann model, each of the input bits is the result of an independent toss of a coin with fixed bias. Namely, for every bit generated, the value 1 is generated with probability $0 < p < 1$ and the value 0 is generated with probability $1 - p$ - with no relation to the value of other bits. We refer to this as the *Von Neumann assumption*.

Von Neumann proposed the following method to extract perfect randomness from these biased bits. First, arrange these sampled bits in pairs $\{(x_i, y_i)\}$. Then, remove pairs where both bits are identical, i.e., leave only pairs of the form $\{(x_i, 1-x_i)\}$. Finally, output the sequence $\{x_i\}$. This simple - if somewhat ‘wasteful’ in input bits - algorithm, is called the *Von Neumann extractor*; and it is not hard to show that if the input satisfies the Von Neumann assumption, then the output is a string of *uniformly random bits*.

Exercise 3.10. [Von Neumann extractor] Show that, if the input of the Von Neumann extractor satisfies the Von Neumann assumption (above), then the output is uniformly random, i.e., each bit x_i is 1 with probability exactly half - independently of all other bits.

The Von Neuman extractor is simple, and the output is proven uniform without any computational assumption. However, the Von Neumann assumption

extractors are referred to in [200] as *computational extractors*. Extractors whose output is random regardless of the adversary’s computational abilities are referred to as *statistical extractors*.

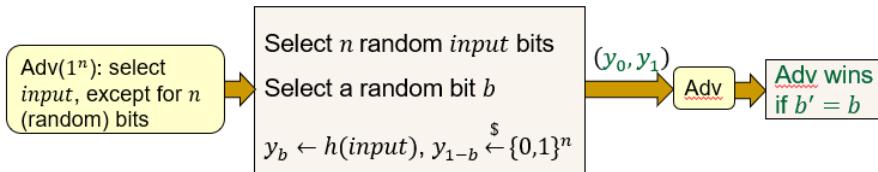


Figure 3.14: Bitwise-Randomness Extractor (BRE) Hash Function.

is hard to justify, for many typical security applications of randomness-extraction. In particular, consider the goal of *key derivation*, where we use some large input x which is ‘fairly random’ but not truly random, such as many measurements (of time, movements, etc.). We cannot use x directly as a key, so we apply a hash and use $h(x)$. However, can we be certain the Von Neumann assumption holds? The following simple exercise show this assumption may not hold - even when every second bit in the input is random!

Exercise 3.11. Consider the following random process for producing bit sequences $\{b_1, b_2, \dots\}$:

$$b_i = \begin{cases} b_i \leftarrow 0 & \text{if } i = 1 \pmod{2} \\ b_i \xleftarrow{\$} \{0, 1\} & \text{otherwise} \end{cases} \quad (3.18)$$

Show that this sequence does not satisfy the Von Neumann assumption, and, in fact, that applying the Von Neumann extractor, will not result in random output string.

3.5.2 The Bitwise Randomness Extractor

We now present another model for randomness extraction, which we call the *bitwise randomness extractor (BRE)* model. We present the BRE model as a simple way to provide readers with an understanding of randomness-extracting hash functions; you will find more advanced and stronger models in the literature, but these are beyond our scope.

Intuitively, a hash function h is a *bitwise-randomness extractor* if its output $h(x)$ is pseudorandom, even if the adversary can *select the input x , except for a ‘sufficient’ number of bits* of the input; these bits are selected randomly. This intuition is illustrated in Fig. 3.14, which defines a ‘game’ where an adversarial algorithm \mathcal{A} tries to defeat the randomness extraction - by selecting some input message, except for n random bits⁹, and then distinguishing between the output and a random n -bit string.

Finally, we define an ‘indistinguishability test’, much like the ones used in Chapter 2 (for IND-CPA encryption (Definition 2.10), PRF, PRG...). Namely,

⁹The choice of requiring exactly n random bits in the input is quite arbitrary - we could have required a larger number of input random bits; we used exactly n just for simplicity.

we select a random bit b , and let y_b be the n -bit output of the hash function h (whose input contains n random bits), and y_{1-b} be n random bits. Notice that $1 - b$ is simply the complement of b , i.e., it is 0 if $b = 1$ and 1 if $b = 0$. The adversary \mathcal{A} ‘wins’ if it correctly guesses the value of b .

Definition 3.10 (Bitwise-Randomness extractor (BRE) hash function). *An efficient hash function $h : \{0, 1\}^* \rightarrow \{0, 1\}^n$ is called bitwise-randomness extractor (BRE) if for every efficient algorithm $\mathcal{A} \in PPT$, the advantage $\varepsilon_{h, \mathcal{A}}^{BRE}(n)$ is negligible in n , i.e., smaller than any positive polynomial for sufficiently large n (as $n \rightarrow \infty$), where:*

$$\varepsilon_{h, \mathcal{A}}^{BRE}(n) \equiv \Pr [1 = BRE_{A, h}(1, n)] - \Pr [1 = BRE_{A, h}(0, n)] \quad (3.19)$$

Where $BRE_{A, h}(\cdot, \cdot)$ is defined in Algorithm 2 and the probability is taken over the random coin tosses of \mathcal{A} and of $BRE_{A, h}(\cdot, \cdot)$.

Algorithm 2 Bitwise-Randomness Extraction Indistinguishability test $BRE_{A, h}(b, n)$.

```
( $m, M$ )  $\leftarrow A(1^n)$ 
If  $|m| \neq |M|$  or  $\sum_{i=1}^{|M|} M(i) < n$  return  $\perp$ 
 $m' \xleftarrow{\$} \{0, 1\}^{|m|}$ 
 $y_b \leftarrow h(m \oplus (m' \wedge M))$ 
 $y_{1-b} \xleftarrow{\$} \{0, 1\}^n$ 
return  $\mathcal{A}(y_0, y_1, m, M)$ 
```

Exercise 3.19 gives a simple example of a hash function which is *not* a BRE.

3.5.3 The Key Derivation Functions (KDF) and the Extract-then-Expand paradigm

One challenge with the use of extractors, is that their output is of limited length (n bits), which may be insufficient for many applications. For example, the TLS protocol (Chapter 7 needs random bits for multiple keys (authentication/encryption, client to server and server to client), for random bits sent in the protocol, and for initialization-vectors for randomized encryption (if used). We could apply an extractor repeatedly to provide all of these bits, but this will be inefficient, in particular, requiring large input (of sufficiently-random bits) to the extractor.

A better alternative is to use the extractor to generate just enough bits, to provide pseudo-random input to a *pseudo-random generator (PRG)*, or to provide the (random) key to a *pseudorandom function (PRF)*. The output of the PRG or PRF can provide as many pseudo-random bits as needed. We now describe this *extract-then-expand* paradigm, proposed by Krawczyk [200] and adopted as an Internet standard [201]. This KDF design consists of *two functions*: an *extract* function and an *expand* function.

	PRF $f_k(x)$	KDF $f_k(x, l, ID)$, Keyed extractor $h_k(x)$	Keyless extractor $h(x)$
Key k	Secret, random	Public, random ('salt')	No key
Input x	Arbitrary	Sufficiently random	
Output	Pseudorandom		

Table 3.2: Comparison: PRF, KDF and (keyed and keyless) randomness extractor hash functions.

The *extract* function is basically a keyed randomness-extracting hash function, which receives a non-secret random *salt* as well as the ‘sufficiently random’ input x , and produces a fixed-length pseudorandom string $k = \text{extract}_{\text{salt}}(x) \in \{0, 1\}^l$. Namely, if x has ‘sufficient randomness’, and if *salt* is chosen randomly, then k is indistinguishable from a truly random string.

The *expand* function has three inputs: (secret, random) key k , bit-length l and an *identifier ID*. The identifier *ID* is optional; it allows deriving multiple independent keys, for different applications (identified by different *IDs*), from the same sufficiently-random input x . The expand function can be implemented using any pseudorandom function *PRF*, as follows. Let L denote the number of bits output by *PRF*; then:

$$\text{expand}_k^{\text{PRF}}(l, ID) = T_1 \# T_2 \# \dots \# T_{\lceil \frac{l}{L} \rceil} \quad (3.20)$$

Where $T(0)$ is a null string, and for every $i > 0$, $T_i = \text{PRF}_k(T_{i-1} \# ID \# i)$.

Intuitively, the output string $\text{expand}_k^{\text{PRF}}(l, ID)$ is pseudorandom, since it is a concatenation of outputs of the PRF with different inputs.

In [201], both the (keyed-hash) *extract* function, and the PRF used in the construction of the *expand* function, are implemented using a keyless hash function h , using the HMAC construction as described in subsection 4.6.3 (and [201]). Namely:

$$\begin{aligned} \text{extract}_{\text{salt}}(x) &= \text{HMAC}_{\text{salt}}^h(x) &= h(\text{salt} \oplus \text{OPAD} \# h(\text{salt} \oplus \text{IPAD} \# x)) \\ \text{PRF}_k(x) &= \text{HMAC}_k^h(x) &= h(k \oplus \text{OPAD} \# h(k \oplus \text{IPAD} \# x)) \end{aligned} \quad (3.21)$$

Where *OPAD* and *IPAD* are fixed strings.

We can also define the KDF as a single function, which combines the *extract* and *expand* functions. In the common case where both the *extract* function and the PRF are implemented using HMAC, as in

$$\text{KDF}_{\text{salt}}(x, l, ID) = T_1 \# T_2 \# \dots \# T_{\lceil \frac{l}{L} \rceil} \quad (3.22)$$

Where $k \equiv \text{HMAC}_{\text{salt}}(x)$, $T(0)$ is a null string, and for every $i > 0$, $T_i = \text{HMAC}_k(T_{i-1} \# ID \# i)$. The

Table 3.2 compares a KDF to a PRF as well as to keyed and keyless extractors.

The following exercise shows that the KDF and PRF definitions are ‘incomparable’: a function can be a PRF but not a KDF and vice versa. Try to solve

it first by yourself, and then see and understand the solution, since once you see the solution it may appear obvious; this is typical for such exercises which clarify the differences between concepts.

Exercise 3.12 (PRF vs. KDF). *Assume that f is a (secure) PRF and g is a (secure) KDF, where for both, the output length is the same as the key length (and denoted n). Derive from f and g a (secure) PRF f' and a (secure) KDF g' , such that f' is not a secure KDF and g' is not a secure PRF.*

Solution:

$$\begin{aligned} f'_k(x) &= \begin{cases} 0^n & \text{if } k = x \pmod{2^n} \\ f_k(x) & \text{otherwise} \end{cases} \\ g'_k(x) &= \begin{cases} k & \text{if } 0^n = x \pmod{2^n} \\ g_k(x) & \text{otherwise} \end{cases} \end{aligned}$$

Complete the solution by explaining or demonstrating why f' is a secure PRF but insecure KDF, and why g' is a secure KDF but insecure PRF. And find another solutions! \square

3.6 The Random Oracle Model

Often, designers use cryptographic hash functions, but without identifying a specific, well-defined requirement of the hash function that suffices to ensure security. However, such constructions are often still ‘secure’ in the practical sense, that no practical attack against them is found - for many years, and often, in spite of considerable motivation for cryptanalysis and efforts.

Of course, there are also plenty of constructions which similarly use hash functions, without a well-defined assumption - but which have been found to be insecure. However, very often, the attacks are *generic*, i.e., do not exploit a weakness of a specific hash function, and apply when the design is implemented with an arbitrary hash function.

Therefore, even when we cannot identify the specific ‘generic’ property of hash function on which security relies, it is useful to identify designs which are vulnerable even for an ‘ideal-security’ hash function. We definitely want to avoid such a design, which would be clearly insecure for *any* hash function! Can we define a property which implies such vulnerable designs, and then *prove* that a candidate design is secure in the (limited) sense of being secure for such ‘ideal’ hash function? Unfortunately, we don’t know how to define an ‘ideal-security’ hash function...

The *Random Oracle Model (ROM)*¹⁰, proposed by Bellare and Rogaway [35], is a common approach to this dilemma. Intuitively, *Random Oracle Model (ROM)* constructions and protocols are secure in the (impractical) case that the parties select $h()$ as a truly random function (for the same domain and range),

¹⁰Often people use the term with ‘methodology’ or ‘method’ instead of ‘model’, but we use ‘model’ since it is more common.

instead of using a concrete, specific hash function as $h()$. The function $h()$ is still assumed to be public, i.e. known to the attacker. Namely, we model an ‘ideal’ hash function as a *random function* (over the same domain and range, i.e., $\{0, 1\}^* \rightarrow \{0, 1\}^n$).

Definition 3.11 (ROM-security). *Let H be the set of all n -bit hash functions, i.e., functions $h : \{0, 1\}^* \rightarrow \{0, 1\}^n$ (for some n). Consider a parameterized scheme \mathcal{S}^h , where h is a given hash function. Also, for any security definition def , let $\varepsilon_{\mathcal{S}^h, \mathcal{A}^h}^{\text{def}} \rightarrow [0, 1]$ be the def -advantage function, defined for a given scheme \mathcal{S}^h and parameterized adversary \mathcal{A}^h , where \mathcal{A}^h is a PPT algorithm, using standard computational model (e.g., Turing machine), with the additional ability to provide an arbitrary input x to h and receive back $h(x)$ (as a single operation).*

We say that the (parameterized) system \mathcal{S}_h is def -ROM-secure, or def -secure under the Random Oracle Model, if the advantage of any PPT adversary \mathcal{A}^h for scheme \mathcal{S}^h , for a random hash function $h \xleftarrow{\$} H$, is negligible, i.e.:

$$\Pr_{h \xleftarrow{\$} H} \varepsilon_{\mathcal{S}^h, \mathcal{A}^h}^{\text{def}} \in \text{NEGL}(n) \quad (3.23)$$

Note about random choice of h (may be skipped). The careful reader may have noticed that it is not well defined how to select an element from an infinite set with uniform probability, hence, $h \xleftarrow{\$} H$ is not well defined. This choice should be interpreted as follows. For any $l > n$, let H^l be the set of all functions from $\{0, 1\}^l$ to $\{0, 1\}^n$, i.e., functions from l -bit binary strings to n -bit binary strings: $H^l = \{h : \{0, 1\}^l \rightarrow \{0, 1\}^n\}$. The set H^l is finite, hence we can define random sampling from it: $h^l \xleftarrow{\$} H^l$. When we write $h \xleftarrow{\$} H$, it should be interpreted as choosing $h^l \xleftarrow{\$} H^l$ for every integer $l > n$; and, for any input string $m \in \{0, 1\}^*$, let $h(m) = h^{|m|}(m)$. \square

From the definition it follows that to ‘prove security under the ROM’, we would analyze the construction/protocol as if the function $h(\cdot)$, used by the protocol, was chosen randomly at the beginning of the execution and then made available to all parties - legitimate parties running the protocols or using the scheme/construction, as well as the adversary.

Security under the ROM vs. Security under the Standard Model. Analysis of security under the ROM model is so widely used, that papers in cryptography often use the term ‘secure in the standard model’ to emphasize that their results are ‘really’ proven secure, rather than only proven under the ROM (or under another idealized model). Of course, these proofs are still based on reductions, i.e., on unproven assumptions; however, these assumptions are ‘standard’ cryptographic assumptions. For many of these standard cryptographic assumptions there are even theoretical results showing that there exist schemes satisfying these assumptions under a ‘standard’ mathematical assumption, most often, assuming $P \neq NP$.

We emphasize again, that ROM-security does not necessarily imply that the design is ‘really’ secure, when implemented with any specific hash function; once the hash function is fixed, there may very well be an adversary that breaks the system. This is true even when the hash function meets the ‘standard’ security specifications, e.g., CRHF. However, ROM-security is definitely a good *indication* of security, since a vulnerability has to use some property of the specific hash function. Indeed, there are many designs which are only proven to be ROM-secure. This motivates many works in cryptography, which provide designs secure in the standard model, as alternatives to known designs which are secure only under the ROM. The following subsection studies one of the most widely used applications of hash functions - the construction of a *Message Authentication Code (MAC)* scheme from a hash function; we show several constructions which are secure under ROM but insecure under the standard model.

3.7 Cryptographic Accumulators

3.8 The Merkle-Damgård Accumulator

In this and the following two sections, we discuss *Digest schemes*, which are a generalization of the Collision-Resistant Hash Function (CRHF). In this section, we begin with the simplest digest scheme, which we refer to as the *Digest-Chain* scheme. The terms Digest schemes and Digest-Chain are not widely used; in particular, the only well-known construction of this type is the Merkle-Damgård construction, which we present, but whose common definition only implements the *digest* function of the *digest-chain* scheme, requiring a minor extension to provide the entire functionality of *digest-chain* schemes. We see this as a demonstration of the importance of distinguishing between an abstract scheme (e.g., block cipher or digest-chain) and constructions of it (e.g., the AES block cipher, or the Merkle-Damgård digest function, or its extension into digest-chain construction).

All digest schemes extend the integrity-goal of CRHF, but instead of hashing a single input, the input to a digest scheme is a *sequence* of input strings (messages), which we usually refer to as a *block*, and typically denote by B . Similarly to hash functions, one can define either a keyless or a keyed Digest-Chain scheme.

Since digest schemes extend the goals of CRHFs, it is easy to see that, for the same arguments presented for CRHF in Section 3.2, there can actually be *no secure digest-chain scheme* (as we define). In spite of that, following the discussion in subsection 3.2.2, we only define the keyless variants, typically constructed from keyless hash functions. All you need to do to have a keyed digest-chain scheme, is to use a keyed CRHF instead, essentially.

Similarly to the case for hash functions and other cryptographic schemes, in order to use asymptotic definitions of security, the functions of the Digest-chain scheme must depend on a *security parameter* which is also the length of the

output of the digest function, i.e., the length of the digest; it is often denoted by n . Recall that, to facilitate asymptotic security definitions, such security parameter (for length of output) was also used for keyless hash functions, but normally not written explicitly; we similarly usually omit the security parameter n of digest schemes.

3.8.1 The collision resistant digest function

In this subsection, we focus on the basic *collision-resistance* property of *digest-chain* schemes. This property is a trivial extension of CRHFs, and requires just a single function, to which we refer as *the digest function* and denote by Δ .

The input to the digest function Δ is a *block* $B = \{m_1, \dots, m_l\}$, i.e., a finite sequence of messages. The output of the digest function, $\Delta(B)$, i.e., the *digest* of B , is an n -bit binary string.

We require the digest function to ensure *collision-resistance*, as defined below. Intuitively, collision resistance means that it is infeasible for a PPT adversary \mathcal{A} to output a *collision*, i.e., two different blocks $B \neq B'$ which have the same digest, i.e., $\Delta(B) = \Delta(B')$. Let us define this a bit more precisely, although we still present the definition for fixed digest length of n bits; recall that the more precise interpretation is that we define a whole sequence of such functions Δ for every integer $n > 0$, we just omit the (n) for brevity and simplicity - and since in practice, we use a specific length anyway.

Definition 3.12. A digest function Δ is an efficiently computable function (in PPT) that maps blocks (finite sequences of binary strings) to n -bit binary strings, i.e., $\Delta : (\{0, 1\}^*)^* \rightarrow \{0, 1\}^*$, where n is the security parameter.

Digest function Δ is collision resistant if the digest collision-resistance advantage $\varepsilon_{\mathcal{A}, \Delta}^{DCR}(n)$ is negligible (in n), for every efficient adversary $\mathcal{A} \in PPT$, where:

$$\varepsilon_{\mathcal{A}, \Delta}^{DCR}(n) \equiv \Pr((B, B') \leftarrow \mathcal{A}(1^n) \text{ s.t. } B \neq B' \wedge \Delta(B) = \Delta(B')) \quad (3.24)$$

Obviously, collision-resistant digest functions are similar to collision-resistant hash functions (CRHFs). It may seem that we can compute the digest of a block merely by concatenating all its inputs into a single string. However, this is incorrect; intuitively, concatenation of all the inputs does not preserve the ‘inter-string boundaries’. For example, surely $(11, 00) \neq (1, 10, 0) \neq (1, 1, 00)$.

3.8.2 The Merkle-Damgård Construction of collision resistant digest function

In this subsection, we present the well-known *Merkle-Damgård* construction, transforming a CRHF into a collision resistant digest function, illustrated in Figure 3.15 and in Equation 3.25.

Given a CRHF h and input block $B = \{m_1, \dots, m_l\}$, the value of

We define $m\mathcal{D}^h.\Delta$, the Merkle-Damgård digest function using CRHF h , as follows:

$$m\mathcal{D}^h.\Delta(\{m_i\}_{i=1}^l) \equiv \begin{cases} \text{For } l > 1 : & h(m\mathcal{D}^h.\Delta(\{m_i\}_{i=1}^{l-1}) \# 1 \# m_l) \\ \text{For } l = 1 : & h(0^{n+1} \# m_1) \end{cases} \quad (3.25)$$

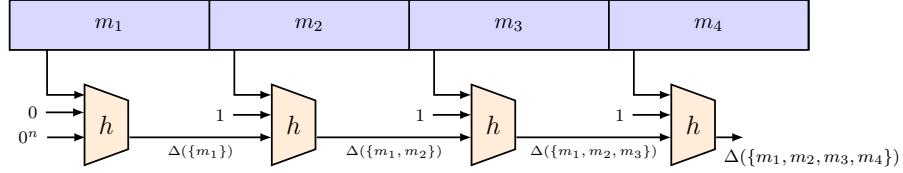


Figure 3.15: The Merkle-Damgård Construction of a collision-resistance digest function from a CRHF. The construction can also be used to construct a CRHF from a compression function, i.e., a CRHF restricted to fixed-length inputs; see details in the text.

Example 3.2. Let $B = \{11, 22, 33\}$ be an input block. Present a formula for $m\mathcal{D}^h.\Delta(B)$, and compute its value $m\mathcal{D}^h.\Delta(B)$ for the case $h = h_{sum}$

Lemma 3.2. If h is a CRHF, then $m\mathcal{D}^h.\Delta$ is a collision-resistant digest function.

Proof: assume, to the contrary, that adversary $\mathcal{A}^{MD} \in PPT$ has non-negligible advantage against $m\mathcal{D}^h.\Delta$, i.e., $\varepsilon_{\mathcal{A}^{MD}, \Delta}^{DCR}(n) \notin NEGL(n)$. We use \mathcal{A}^{MD} to construct adversary \mathcal{A}^h which has non-negligible advantage against h . Specifically, \mathcal{A}^h runs \mathcal{A}^{MD} , and whenever \mathcal{A}^{MD} outputs a collision for $m\mathcal{D}^h.\Delta$, then \mathcal{A}^h outputs a collision for h . Let us explain how.

Let the collision be (B, B') , i.e., $: B, B' \in (\{0, 1\})^*$ and $(B \neq B') \wedge \Delta(B) = \Delta(B')$. Denote the number of messages in B by l and the number of messages in B' by l' , and, without loss of generality, assume $l \geq l'$. The proof is by induction on l .

If $l = 1$ then l' must also be one, hence: $m\mathcal{D}^h.\Delta(B) = h(0^{n+1} \# m_1)$ and $m\mathcal{D}^h.\Delta(B) = h(0^{n+1} \# m'_1)$. Since $B \neq B'$, and in this case $B = \{m_1\}$ and $B' = \{m'_1\}$, it follows that $m_1 \neq m'_1$. Let $\bar{m} = 0^{n+1} \# m_1$, $\bar{m}' = 0^{n+1} \# m'_1$; it follows that $\bar{m} \neq \bar{m}'$. Hence, (\bar{m}, \bar{m}') is a collision that \mathcal{A}^h will output, as claimed.

Assume therefore that the claim holds for $l = i$ and we prove it holds also for $l = i + 1$. First assume $l' > 1$. We have

$$m\mathcal{D}^h.\Delta(\{m_1, \dots, m_{i+1}\}) \equiv h(m\mathcal{D}^h.\Delta(m_1, \dots, m_i) \# 1 \# m_{i+1}) \quad (3.26)$$

and

$$m\mathcal{D}^h.\Delta(\{m'_1, \dots, m'_{l'}\}) \equiv h(m\mathcal{D}^h.\Delta(m'_1, \dots, m'_{l'-1}) \# 1 \# m'_{l'})$$

and of course

$$m\mathcal{D}^h.\Delta(\{m_1, \dots, m_{i+1}\}) = m\mathcal{D}^h.\Delta(\{m'_1, \dots, m'_{l'}\})$$

Now, if the inputs to the hash are identical, than this means that:

$$m\mathcal{D}^h.\Delta(m'_1, \dots, m'_{l'-1}) = m\mathcal{D}^h.\Delta(m_1, \dots, m_i)$$

This would contradict the induction hypothesis. If the inputs to the hash are different, then this is a collision. Hence, in both cases, \mathcal{A}^h can output a collision, as claimed.

It remains to consider the case where $l' = 1$ (and we prove for $l = i + 1$ after we proved for $l = i$). Equation 3.26 still holds, but in this case we have

$$m\mathcal{D}^h.\Delta(\{m'_1\}) \equiv h(0^{n+1} \parallel m'_{l'})$$

and

$$m\mathcal{D}^h.\Delta(\{m_1, \dots, m_{i+1}\}) = m\mathcal{D}^h.\Delta(\{m'_1\}) = h(0^{n+1} \parallel m'_{l'})$$

Since the $(n+1)^{th}$ bit differs between these two inputs to h , but their outputs are the same, it follows that also in this case, \mathcal{A}^h outputs a collision and the claim is complete. \square

Exercise 3.13. Show that collisions may be possible for a variant of the construction where Equation 3.25 is replaced by:

$$m\mathcal{D}^h.\Delta(\{m_i\}_{i=1}^l) \equiv \begin{cases} \text{For } l > 1 : & h\left(m\mathcal{D}^h.\Delta(\{m_i\}_{i=1}^{l-1}) \parallel 0 \parallel m_l\right) \\ \text{For } l = 1 : & h(0^{n+1} \parallel m_1) \end{cases}$$

Note: this is not an easy exercise. *Hint:* Assume some CRHF h' and use it to design a very unlikely CRHF h for which such collision is possible. Reading the proof of Lemma 3.2 carefully may help. \square

The Merkle-Damgård construction was proposed independently by Merkle and by Damgård, in [94, 231], respectively; I personally find Damgård's text easier to follow. Note, however, that in both works, and in most of the extensive work building on this important construction, this construction is presented as a construction of a *CRHF* from a *compression function*, where a compression function¹¹ is defined just like a CRHF, except that its input domain is the set of binary strings of some specific length $m > n$ (where n is still used for the size of the output string). See Figure 3.16. We have added the observation that the same construction, when h is either a compression function or a hash function, results in a digest scheme.

¹¹The term *compression function* may not be the best choice; it may have been clearer to refer to such functions, with Fixed-Input-Length (FIL), as FIL-hash functions. However, the use of ‘compression functions’ is entrenched in the literature; hence, it seems best to use it.

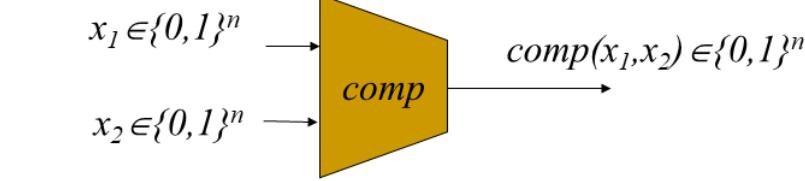


Figure 3.16: A compression function, mapping fixed-length input to shorter-length output. For simplicity, in this example we show input length as twice the output length, i.e., $2n$.

One motivation to build a cryptographic hash function from a compression function, is the ‘cryptographic building blocks’ principle 8: *The security of cryptographic systems should only depend on the security of a few basic building blocks. These blocks should be simple and with well-defined and easy-to-test security properties.*

Cryptographic hash functions are one of these few building blocks of applied cryptography, due to their simplicity and wide range of applications. However, compression functions are even simpler, since their input is restricted to fixed-input length strings. Let us, therefore, explain how the construction can be also be used for this application, i.e., constructing a CRHF from a compression function

The Merkle-Damgård construction, simplified: messages consisting of integral blocks. We first describe the Merkle-Damgård construction, for a simplified case, where the input (binary) strings split into an exact number of $(m - n)$ -bits blocks. Namely, the input m satisfies $|m| = 0 \pmod{m - n}$. In this case, given a collision-resistant compression function h , can simply define the following CRHF h' :

$$h'(m') = m \Delta^h . \Delta \left(\left\{ m'[1 : (m - n)], m'[(m - n + 1) : 2 \cdot (m - n)], \dots, m'[|m'| - (m - n) + 1 : |m'|] \right\} \right) \quad (3.27)$$

Where we use the notation $m'[i : j]$ for the sub-string of m' , from the i^{th} bit to the j^{th} bit.

It is easy to see, from Lemma 3.2, that if h is a collision-resistant compression function, then h' would be collision-resistant.

The Merkle-Damgård construction with MD-strengthening. The Merkle-Damgård construction is usually presented with an additional pre-processing step called *MD-strengthening*¹². This step is necessary to avoid the restriction that the input m satisfies $|m| = 0 \pmod{m - n}$, required by h'

¹²The reader may wonder about this term; ‘MD’ clearly stands for Merkle and Damgård, so that’s fine, but why ‘strengthening’? My guess is that this term is used in relation to naive and vulnerable methods of padding the input to the Merkle-Damgård construction.

(from Equation 3.27). To remove this restriction, we do a further step that allows us to handle any binary string as input.

The basic idea of MD-strengthening is to pad the message m' with $(m - n) - [n' \bmod (m - n)]$ additional bits, e.g. all zeroes, so that its length would become an exact multiple of $m - n$ bits ‘messages’. However, this introduces a small issue: the hash of m' would be the same as that of $m' \# 0$ - an extra zero bit would have the same impact as a pad bit! This can be solved in different ways; MD-strengthening simply appends one additional $m - n$ bits ‘message’, which contains the binary encoding of the number of pad bits, i.e., of: $(m - n) - [n' \bmod (m - n)]$. Let $\text{bin}^l(x)$ denote the l bit string containing the encoding of an integer $x : 0 \leq x \leq l$; then we need to append $\text{bin}^{m-n}((m - n) - [n' \bmod (m - n)])$. Therefore, the complete MD construction of a CRHF h^{MD} from a compression function h , including padding and MD-strengthening, is defined as:

$$\begin{aligned} h^{MD}(m') = h'(m' \# 0^{(m-n)-[n' \bmod (m-n)]} \# \\ \# \text{bin}^{m-n}((m - n) - [n' \bmod (m - n)])) \end{aligned} \quad (3.28)$$

Where $\text{bin}^l(x)$ is defined above and h' is defined in Equation 3.27.

Lemma 3.3. *If h is a collision-resistant compression function, then h^{MD} , defined as in Equation 3.28, is a collision-resistant hash function (CRHF).*

Proof sketch: follows from Lemma 3.2. □

Initialization Vector (IV). The string of 0^n bits that we append to the first message m_1 in Equation 3.25, is often referred to as an Initialization Vector (IV), similarly to the use of these term in some descriptions of the ‘modes of operation’ of block cipher (Section 2.8). The choice of 0^n is completely arbitrary; and n -bit string could have been used, as long as it is a *fixed* string - if we allow the IV to be chosen freely each time, then collisions are possible (see next exercise). Note that it is possible to use a random IV as a way to key the hash function - but, again, only if this IV is used for all messages.

Exercise 3.14. *Assume that $|m_1| = n$, and consider a variant on the MD construction where we change Equation 3.25 so that for $l = 1$, we have: $M\mathcal{D}^h.\Delta(\{m_1\}) \equiv m_1$. This variant ‘saves’ a hash operation; however, show that it may allow collisions.*

3.8.3 The Extend Function and Validation of Entries and Extensions

Digest schemes provide additional integrity mechanisms *beyond* collision resistance. These mechanisms are useful for many applications and situations, in which the sequence of messages is *dynamic*, such as in a log scenario. Clearly, in a log, new messages may be added over time. Furthermore, we may want to

add messages to the log, to validate that a particular message appears in the log, or to validate that a new digest of a log is consistent with a previous digest, all *without* re-using the entire set of messages. There are two motivations for not requiring the entire set of messages: improved efficiency - and allowing validation and log-extension by different parties, who may not even possess all the messages in the log. The reader may already see how this will soon bring us to more elaborate digest schemes, such as Merkle digests and Blockchains - the topics of the following two sections.

However, for now, we still continue to discuss the simpler *digest-chain scheme*. Our discussion so far was limited to the digest function, which can be viewed as a very basic digest-chain scheme; we now extend it, to define a ‘proper’ digest-chain scheme.

The extension involves only one more function, which we actually refer to as the *extend function*, and denote *Extend*. This function receives the ‘current’ digest and a sequence of (one or more) additional (‘new’) messages, and produces the ‘new’ digest. The only additional requirement we need to make is that the extend function is consistent with the digest function, i.e., that for any two sequences of messages M, M' holds:

$$\text{Extend}(\Delta(M), M') = \Delta(M \# M') \quad (3.29)$$

The definition of a digest-chain scheme and its security requirements follows.

Definition 3.13. A *Digest-Chain scheme* is a pair (Δ, Extend) of PPT-computable functions:

Δ is a digest function as defined in Definition 3.12.

Extend is the extend function, whose inputs are a digest Δ and a sequence of ‘additional’ messages M' , and whose output is a ‘new’ digest Δ' .

A digest-chain scheme is correct if for any two sequences of messages M, M' , Equation 3.29 holds.

A digest-chain scheme is secure if it is correct and its digest function is collision-resistant (see Definition 3.12).

In spite of this simple definition and requirements, there are three different ways to use the *Extend* function, for different applications and scenarios:

Extend current digest: this is direct use of Δ to extend the sequence of messages, $M = \{m_1, \dots, m_l\}$, with additional messages $M' = \{m_{l+1}, \dots, m_L\}$. The digest function will receive as input the current digest $\Delta(M_l)$, and the sequence of additional messages M' , and produce the new digest; the correctness property ensures that this would be the same as $\Delta(M \# M')$.

Validate digest consistency: in this use-case, the current digest $\Delta(M)$ and the new digest $\Delta(M \# M')$ are computed by one entity, e.g., a bank, and received by a different entity, *Val*, e.g., a customer. *Val* may want to validate that $\Delta(M \# M')$ is consistent with a previously received

$\Delta(M)$ and with a given set of new messages M' . Notice that in some applications, Val may not even be interested in the specific additional transactions in the set $M_{l+1,l'}$, but they must be used for validation of consistency, when using a digest-chain scheme; this will be avoided in the Merkle-digest scheme and Blockchain scheme, presented in the following sections.

Validate inclusion: in this use-case, Val is an entity who has a digest $\Delta(M)$ for some sequence M , but without knowing M . Val receives a particular message m and wants to validate that m is one of the messages in M , possibly also with its sequence number. To this end, Val is provided with m , the digest $\Delta(M_{pre})$ of a sequence M which is the prefix of M up to message m , and the sequence M_{suf} which is the suffix of M following m . Then Val can validate that $\Delta(M) = \text{Extend}(\Delta(M_{pre}, \{m\} \# M_{suf}))$.

The Merkle-Damgård extend function. The Merkle-Damgård construction allows extension, a required feature for a digest-chain scheme. We define the Merkle-Damgård extend function $\text{MD}^h.\text{Extend}$, based on a hash function h :

$$\text{MD}^h.\text{Extend}(\Delta, \{m_1, \dots, m_l\}) \equiv \begin{cases} \text{Let } \Delta_1 \leftarrow h(\Delta + 1 + m_1) \\ \text{For } l > 1: \\ \quad \text{MD}^h.\text{Extend}(\Delta_1, \{m_2, \dots, m_l\}) \\ \text{For } l = 1: \Delta_1 \end{cases} \quad (3.30)$$

Lemma 3.4. If h is a CRHF, then $(\text{MD}^h.\Delta, \text{MD}^h.\text{Extend})$ is a secure digest-chain scheme.

Proof: Correctness follows by substituting using the definitions of the functions, and collision resistance of $\text{MD}^h.\Delta$ was proven in 3.2. \square

Note that we can extend a digest Δ without knowing the messages which were input to the digest function, resulting in digest Δ . This ability, to extend the digest without knowing the original messages, ‘breaks’ some naive methods for constructing a message authentication code (MAC) from a hash function, when the hash functions are based on the Merkle-Damgård construction. See subsection 4.6.3.

3.9 The Merkle Accumulator

Like the digest-chain scheme discussed in Section 3.8, the Merkle digest scheme extends the integrity-goal of CRHF, mainly by using a *digest function* $\Delta : \{m_i \in \{0, 1\}^*\}_i \rightarrow \{0, 1\}^*$. Specifically, instead of hashing a single input, the input to the Merkle digest function is a set of objects, often referred to as *files* or *messages*. For example, a typical software distribution may consist of multiple messages, e.g., (m_1, m_2, m_3, m_4) , such as when using large libraries. The Merkle-digest Δ

function also has the same collision-resistance requirement as of the digest-chain Δ function.

There are two main differences between the Merkle digest scheme and the digest-chain scheme. The first difference is that in the Merkle digest scheme, *validation of inclusion* is a mandatory, important mechanism, whose *efficiency* is often a main design goal of constructions of Merkle digest scheme. In typical applications of Merkle digest schemes, it is desirable to allow recipients to efficiently validate integrity (inclusion) of one or few messages that they need, using only limited, concise information, for efficiency (and sometimes also for privacy). To support validation of inclusion, Merkle schemes (always) include the *PoI* function, that computes a *Proof-of-Inclusion*, and the *VerPoI* function, that validates a given *PoI*.

The other main difference between the digest-chain scheme and the Merkle-digest scheme, is in the mechanisms and requirements related to *sequence extension and validation*. Recall that the digest-chain scheme handles extensions of the message-sequence, as well as validation of consistency and of inclusion, all using the same - optional - *Extend* function, as described in subsection 3.8.3. In the Merkle-digest scheme, validation of inclusion is done by the (mandatory) *PoI* and *VerPoI* functions, as explained; while sequence extension and validation are handled with the (optional) *Proof-of-Consistency (PoC)* and *validation of PoC (VerPoC)* functions. These functions are often more complex; in fact, while we present two Merkle scheme constructions, we describe the extension and validation of consistency mechanisms only for one of the two (with a simple - but inefficient - design).

We next describe the Merkle digest scheme and two constructions; however, let us first say few words on our naming of the scheme and of the constructions.

On naming: Merkle digest scheme vs. the Merkle tree construction. The term *Merkle tree* is widely used in the literature, often in reference to the ‘classical’ construction proposed by Ralph C. Merkle, one of the pioneers of modern cryptography, in [230]. Merkle’s construction, and variants of it, are usually based on a CRHF and on the graph-theory concept of a *tree*¹³, and are widely used in cryptography. We present below two of the simpler, classical constructions, and mention few of their many applications.

However, while different constructions were proposed, they are usually all referred to as *Merkle tree* or some variant of that name; the term is also used to refer to this type of cryptographic scheme (rather than a specific construction). However, the author believes that it is important to distinguish between an abstract scheme (e.g., block cipher) and constructions of it (e.g., AES and DES).

We think it is better to use different terms for the abstract scheme and for each specific construction. Specifically, we use the term *Merkle digest* to refer to

¹³Readers not familiar with graph theory and specifically trees, are encouraged to learn this fascinating and useful topic, e.g. see [120, 326]. However, this is not be really critical for understanding the constructions.

the abstract scheme. We use the terms *Merkle tree* ($M\mathcal{T}$) and *Flat Merkle tree* ($2lM\mathcal{T}$) to refer to two specific *constructions* we present. The $M\mathcal{T}$ construction is a minor variant of the one by Merkle (in [230]), and the $2lM\mathcal{T}$ construction is a simpler, ‘folklore’ construction. We also discuss the *extensible* Merkle digest scheme, which adds additional functionality to the Merkle digest scheme, and constructions for it.

Note, also, that our definition of a Merkle digest scheme is *not* a widely-adopted definition; the author is not aware of any existing widely-deployed definition for such schemes, and believes that this definition is a bit simpler than some of the few definitions that the author did find. Comments on these points (and other aspects of this manuscript) would be highly appreciated.

3.9.1 The Merkle digest scheme: Definitions

Let us begin with a slightly informal description of the Merkle digest scheme. Similarly to hash functions, one can define either a keyless or a keyed Merkle digest scheme. Following the discussion in subsection 3.2.2, we only define the keyless variants. Most practical protocols using Merkle digest schemes (or the specific Merkle tree construction), use the keyless versions, constructed from keyless hash functions. Let us finally get to the definition of the Merkle digest scheme, as well as its correctness and security requirements.

A Merkle digest scheme consists of three functions: a *digest* function Δ , a *Proof-of-Inclusion* function PoI and a *Proof-of-Inclusion Verification* predicate $VerPoI$. Similarly to keyless hash functions, the three functions of the Merkle digest scheme are defined for a different security parameter n , which is also the length of the digest, but we usually do not explicitly write it as a parameter. If we do need to explicitly write it, we will use the same notation as for keyless hash functions, e.g., Δ . Intuitively:

- The Δ function generalizes the digest functionality of a hash function; the difference is that a hash function receives, as input, a single string, while the Merkle digest scheme’s Δ function receives a set of strings. Namely, given as input a sequence $B = \{m_i \in \{0,1\}^*\}_i$ of messages¹⁴, the output (digest) of $\Delta(B)$ is n -bits long, i.e., $\Delta(B) \in \{0,1\}^n$.
- The PoI function produces a sequence of strings which provides a *proof of inclusion* $PoI(B, i)$ for the contents of the i^{th} message in the set B (i.e., m_i).
- The $VerPoI(d, m, i, p)$ function uses the proof p to verify that m is the i^{th} message in some set B , whose digest is d .

The definition follows.

Definition 3.14 (Merkle digest scheme). *A Merkle digest scheme \mathcal{M} is a tuple of three PPT functions $(\mathcal{M}.\Delta, \mathcal{M}.PoI, \mathcal{M}.VerPoI)$, where:*

¹⁴In some implementations, each message is of fixed length, e.g., $m_i \in \{0,1\}^n$.

$m.\Delta$ is the Merkle tree digest function, whose input is a sequence of messages $B = \{m_i \in \{0, 1\}^*\}_i$ and whose output is an n -bit digest: $m.\Delta : (\{0, 1\}^*)^* \rightarrow \{0, 1\}^n$.

$m.PoI$ is the Proof-of-Inclusion function, whose input is a sequence of messages $B = \{m_i \in \{0, 1\}^*\}_i$, an integer $i \in [1, |B|]$ (the index of one message in B), and whose output is a Proof-of-Inclusion (PoI): $m.PoI : (\{0, 1\}^*)^* \times \mathbb{N} \rightarrow \{0, 1\}^*$.

$m.VerPoI$ is the Verify-Proof-of-Inclusion predicate, whose inputs are digest $d \in \{0, 1\}^n$, message $m \in \{0, 1\}^*$, index $i \in \mathbb{N}$, proof $p \in \{0, 1\}^*$, and whose output is a bit (1 for ‘true’ or 0 for ‘false’): $m.VerPoI : \{0, 1\}^n \times \{0, 1\}^* \times \mathbb{N} \times \{0, 1\}^* \rightarrow \{0, 1\}$.

Let us now define the correctness and security requirements of Merkle digest schemes.

Definition 3.15. • A Merkle digest scheme is collision resistant if no efficient (PPT) adversary \mathcal{A} can find two colliding sequences. Namely, for every efficient (PPT) adversary \mathcal{A} , the collision advantage $\varepsilon_{m,\mathcal{A}}^{Coll}(n)$ is negligible in n ($\varepsilon_{m,\mathcal{A}}^{Coll}(n) \in NEGL(n)$), where:

$$\varepsilon_{m,\mathcal{A}}^{Coll}(n) \equiv \Pr \left[\begin{array}{l} (x, x') \leftarrow \mathcal{A}(1^n) \text{ s.t. } (x \neq x') \\ \wedge (m.\Delta(x) = m.\Delta(x')) \end{array} \right] \quad (3.31)$$

• A Merkle digest scheme m is correct if a correctly-generated proof of inclusion is always verified correctly. Namely, for every sequence of messages $B = \{m_i \in \{0, 1\}^*\}_i$ and every index $i \in [1, |B|]$, the Proof-of-Inclusion verifies correctly, i.e.:

$$m.VerPoI(m.\Delta(B), m_i, i, m.PoI(B, i)) = \text{TRUE} \quad (3.32)$$

• The Merkle digest scheme verifies inclusion if no efficient (PPT) adversary \mathcal{A} can forge a proof of inclusion. Namely, for every efficient (PPT) algorithm \mathcal{A} , the PoI advantage $\varepsilon_{m,\mathcal{A}}^{PoI}(n)$ is negligible in n , ($\varepsilon_{m,\mathcal{A}}^{PoI}(n) \in NEGL(n)$), where:

$$\varepsilon_{m,\mathcal{A}}^{PoI}(n) \equiv \Pr \left[\begin{array}{l} (d, m, i, p) \leftarrow \mathcal{A}(1^n) \text{ s.t.:} \\ VerPoI(d, m, i, p) = \text{TRUE} \wedge \\ (\exists B \in \{\{0, 1\}^*\}^*) \text{ s.t.} \\ d = \Delta(B) \wedge (m \notin B) \end{array} \right] \quad (3.33)$$

Where the probabilities are taken over the random coin tosses of \mathcal{A} .

A Merkle digest scheme m is secure if it is correct, collision-resistant and verifies inclusion.

Merkle digest constructions are optimized not only for computation time, but also to have succinct proofs. The length of the PoI in the simpler *Flat*

Merkle Tree ($2lM\mathcal{T}$) construction of subsection 3.9.4 is $n \cdot l$ bits, and this is reduced to $n \cdot \lceil \log(l) \rceil$ bits in the better-optimized (and widely used) Merkle tree ($M\mathcal{T}$) construction of subsection 3.9.5. The length of the digest is typically n - the same as the length of the output of the underlying hash function h .

3.9.2 Extending the sequence: Proofs of Consistency

In many applications of the Merkle digest scheme, the digest is always computed and validated by applying the Δ function to the entire sequence of messages. However, there are also applications of Merkle digest schemes where entries may be added to the sequence over time, motivating the use of special operations to extend and validate the consistency of the digest of the extended sequence - like the *Extend* function of the digest-chain scheme (subsection 3.8.3). For example, this may be desirable in order to allow a recipient to validate new entries and the corresponding new digest, even if the recipient did not maintain all the entries which were included in the sequence so far, e.g., for maintaining a log or ledger of transactions.

The Merkle digest scheme, as defined in Definition 3.14, allows verification of the integrity of the sequence (e.g., log) using just the digest, and verification of a particular entry, using the *PoI* mechanism. However, to allow the sequence of events to be extended over time, we need an additional mechanism: *Proof of Consistency* (*PoC*).

Let B_C be a block of l_C messages, $B_C = \{m_{C,1}, \dots, m_{C,l_C}\}$. Assume Bob receives from Alice, securely, the digest of B_C , which we denote Δ_C , i.e., $\Delta_C \equiv \Delta(B_C)$.

Let B_N be another block, of l_N messages: $B_N = \{m_{N,1}, \dots, m_{N,l_N}\}$. Intuitively, block B_N is an *extension* of block B_C . Namely, Alice computes and sends to Bob the digest of the concatenation of the two blocks, $B_C \# B_N$; Alice may also send B_N to Bob - but not necessarily. Let B_{CN} denote this concatenation, i.e.:

$$B_{CN} \equiv B_C \# B_N = \{m_{C,1}, \dots, m_{C,l_C}, m_{N,1}, \dots, m_{N,l_N}\}$$

Let Δ_{CN} denote the digest of the concatenation B_{CN} , i.e., $\Delta_{CN} \equiv \Delta(B_C \# B_N)$.

The goal of the *PoC* mechanism, is to allow Bob, using the *VerPoC* function, to validate that Δ_{CN} is consistent with Δ_C , i.e., that Δ_{CN} is a digest of a block whose prefix has the digest Δ_C .

Definition 3.16 (Proof-of-Consistency). *We say that Merkle digest scheme m supports Proof-of-Consistency given two additional functions, $m.PoC$ for extending a digest, and $m.VerPoC$ for validating an extension, where:*

$m.PoC(B_C, B_N)$ is the Extend and Proof-of-Consistency function PoC , whose input are two sequences, B_C and B_N , and whose output $p_{CN} = m.PoC(B_C, B_N)$ is a binary string which we call the Proof-of-Consistency from $\Delta_C \equiv m.\Delta(B_C)$ to $\Delta_{CN} \equiv m.\Delta(B_C \# B_N)$.

$m.\text{VerPoC}(\Delta_C, \Delta_{CN}, l_C, l_N, p) \in \{\text{True}, \text{False}\}$ is the Verify-Proof-of-Consistency predicate, whose inputs are the two digests Δ_C, Δ_{CN} , the numbers of entries (l_C and l_N), and the purported Proof-of-Consistency (PoC) p .

We use M to refer to the entire Merkle digest scheme, including the PoC , VerPoC functions.

We say that the Merkle digest scheme M has a correct PoC if for every two sequences of messages $B_C = \{m_{C,1}, \dots, m_{C,l_C}\}$, $B_N = \{m_{N,1}, \dots, m_{N,l_N}\}$, the Proof-of-Consistency verifies correctly, i.e.:

$$M.\text{VerPoC}(M.\Delta(B_C), M.\Delta(B_C \# B_N), l_C, l_N, M.PoC(B_C, B_N)) = \text{TRUE} \quad (3.34)$$

We say that M has a secure PoC , if for every efficient (PPT) algorithm \mathcal{A} , the PoC -advantage $\varepsilon_{M,\mathcal{A}}^{PoC}(n)$ is negligible in n , where:

$$\varepsilon_{M,\mathcal{A}}^{PoC}(n) \equiv \Pr \left[\begin{array}{l} (B_C, B_A, l_C, l_A, p) \leftarrow \mathcal{A}(1^n) \text{ s.t.} \\ M.\text{VerPoC}(M.\Delta(B_C), M.\Delta(B_A), l_C, l_A, p) = \text{TRUE} \wedge \\ \wedge B_C \text{ is not a prefix of } B_A \end{array} \right]$$

Where the probability is taken over the random coin tosses of \mathcal{A} .

Merkle Digest PoC vs. Blockchain PoC Merkle Digest and digest-chains are both used in many *Blockchain* systems. Blockchains systems allow parties to agree on a sequence of *blocks* of events (entries). Blockchains, like Merkle digests, provide PoC and VerPoC mechanisms to ensure *consistency*. However, this does not necessarily mean that they use the Proof-of-Consistency of the Merkle scheme; often, blockchains use only the PoC of the digest chain (using the *Extend* function). We discuss the blockchain scheme and constructions in Section 3.10.

3.9.3 Merkle Digest scheme and Privacy

Using Merkle Digest schemes for privacy. A Merkle Digest scheme may also be useful for privacy, when some recipients should have access only to some files, e.g., if each file m_i contains data which is private to user i . Note, however, that CRHFs - and Merkle digests - may not ensure confidentiality. Collision-resistance does not ensure that the value of $h(m)$ will not expose some information about m . Namely, for such privacy properties, the hash function h should also have additional properties, such as the other properties we discuss later on for general-purpose cryptographic hash functions.

Exercise 3.15. Let h be a (keyed or keyless) CRHF. Use h to design another hash function g , s.t. (1) g is also a CRHF, yet (2) g exposes one or more bits of its input. Explain why this implies that the two Merkle constructions we discussed, do not guarantee privacy. In particular, explain why the PoI of one message may expose information about other messages.

3.9.4 $2lMT$: the Flat Merkle Tree construction

In this and the next subsection, we present two ‘classical’ constructions of a Merkle Tree, from an underlying CRHF. In this subsection, we begin with the simple *Flat Merkle Tree* ($2lMT$) construction. This construction results in long proofs of inclusion; for input which is a sequence of l messages, the resulting Proof-of-Inclusion requires $l \cdot n$ bits. Nevertheless, for some applications, this construction suffices; and it is surely a good stepping stone to the more complex and efficient Merkle tree (MT) construction.

As the name *Flat Merkle Tree* ($2lMT$) implies, the construction uses a ‘flat’ tree with only two layers. We first apply the hash function to each of the input messages, and then apply it again - to the concatenated digest of all messages. We illustrate the keyless and keyed variants of the *Flat Merkle Tree* ($2lMT$) construction, in Figure 3.17 and Figure 3.18, respectively. The definition for the keyless $2lMT$ follows; we leave the definitions of the keyed versions, with ACR-hash and with TCR-hash, to the reader.

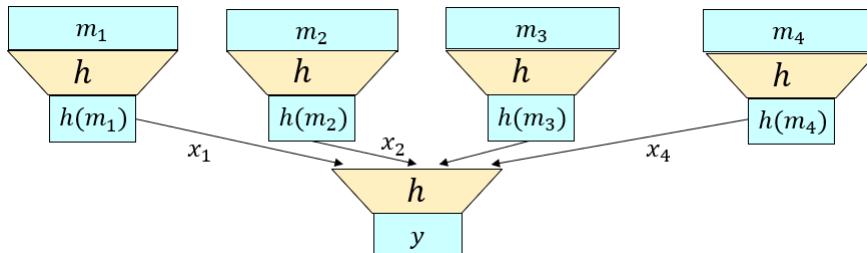


Figure 3.17: The keyless *Flat Merkle Tree* ($2lMT$) construction, applied to input set of four-messages $B \equiv \{m_1, m_2, m_3, m_4\}$. Let $x_i = h(m_i)$ denote the hash of a single message m_i (for $i \in \{1, 2, 3, 4\}$). The digest of the set B is the hash of the concatenation of the hashes of each message, i.e., $y = h[h(m_1) \# h(m_2) \# h(m_3) \# h(m_4)]$. We denote the digest by $2lMT.\Delta(B) = y$.

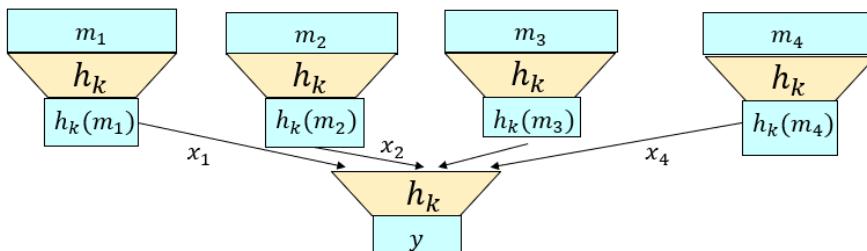


Figure 3.18: The *keyed* Flat Merkle Tree ($2lMT$) construction, applied to input set of four-messages $B \equiv \{m_1, m_2, m_3, m_4\}$. This keyed construction is a simple adaptation of the keyless construction of Figure 3.17.

Definition 3.17 (The *Flat Merkle Tree (2lMT)* construction). *Given a keyless function $h : \{0, 1\}^* \rightarrow \{0, 1\}^n$, we define the Flat Merkle Tree (2lMT) construction as follows, for input containing l messages:*

$$2lMT.\Delta(m_1, \dots, m_l) \equiv h[h(m_1) \mathbin{\#} \dots \mathbin{\#} h(m_l)] \quad (3.35)$$

$$2lMT.PoI((m_1, \dots, m_l), j) \equiv \{h(m_i)\}_{i=1}^l \quad (3.36)$$

$$\begin{aligned} 2lMT.VerPoI(d, m, j, p) &\equiv \\ &\equiv \left[\begin{array}{l} \text{TRUE if } \left(\exists l \in \mathbb{N}, \{p_i \in \{0, 1\}^n\}_{i=1}^l \right) \text{ s.t.} \\ p = (p_1, \dots, p_l) \wedge p_j = h(m) \wedge \\ d = h(p_1 \mathbin{\#} \dots \mathbin{\#} p_l) \end{array} \right] \end{aligned} \quad (3.37)$$

The reader may notice that the $2lMT.PoI$ function returns the same proof for all messages, i.e., regardless of the selection j .

Theorem 3.3. *The Flat Merkle Tree (2lMT) construction is a correct and secure Merkle digest scheme.*

Proof: correctness follows from the definition, and applying Equation 3.36 and Equation 3.37.

To prove security, let us first show negligible advantage for finding collisions $\varepsilon_{2lMT, \mathcal{A}}^{Coll}(n)$. Specifically, we show that every collision of $2lMT.\Delta$, gives a collision of h .

Let $x \equiv \{m_1, \dots, m_l\}$ and $x' \equiv \{m'_1, \dots, m'_{l'}\}$ be a collision for $2lMT.\Delta$, i.e., $x \neq x'$ yet:

$$2lMT.\Delta(x) = 2lMT.\Delta(x') \quad (3.38)$$

For every $i \in [1, l]$, let $h_i = h(m_i)$; similarly, for every $i \in [1, l']$, let $h'_i = h(m'_i)$. Directly from Equation 3.35:

$$\begin{aligned} 2lMT.\Delta(x) &= h(h_1 \dots h_l) \\ 2lMT.\Delta(x') &= h(h'_1 \dots h'_{l'}) \end{aligned}$$

Hence, from Equation 3.38, $h(h_1 \dots h_l) = h(h'_1 \dots h'_{l'})$. If $h'_1 \dots h'_{l'} \neq h'_1 \dots h'_{l'}$, then this is a collision to h , and we are done (with the collision part). Assume, therefore, that $h_1 \dots h_l = h'_1 \dots h'_{l'}$; since the output of h is always the same number of bits (n), i.e., $(\forall i)|h_i| = |h'_i| = n$, then clearly, $l = l'$. Furthermore, for every i holds $h_i = h'_i$.

Now, since $x \neq x'$ and $l = l'$, it follows there is some i s.t. $m_i \neq m'_i$. However, since $h_i = h'_i$, i.e., $h(m_i) = h(m'_i)$, we found a collision for h .

It ‘only’ remains to show negligible advantage for the PoI requirement, i.e., to show that $\varepsilon_{2lMT, \mathcal{A}}^{PoI}(n)$, defined in Equation 3.36, is negligible - or that we can find a collision for h . Namely, this is the probability that the adversary

outputs $(\{m_1, \dots, m_l\}, d, m, i, p)$ s.t. $m_i \neq m$ yet $d = 2lM\mathcal{T}.\Delta(\{m_1, \dots, m_l\})$ and $2lM\mathcal{T}.VerPoI(d, m, i, p) = \text{TRUE}$.

From Equation 3.35, since $d = 2lM\mathcal{T}.\Delta(\{m_1, \dots, m_l\})$, we have that $d = h[h(m_1) + \dots + h(m_l)]$. From Equation 3.37, since $2lM\mathcal{T}.VerPoI(d, m, i, p) = \text{TRUE}$, then p is a tuple $p = (p_1, \dots, p_{l'})$ of some number l' of n -bit strings, s.t. $p_i = h(m_i)$ and $d = h(p_1 + \dots + p_{l'})$.

Combining these two equations for d , we have that: $h[h(m_1) + \dots + h(m_l)] = h(p_1 + \dots + p_{l'})$. If the inputs differ, we have a collision for h , as required. Hence, assume that the inputs are equal, i.e., $h(m_1) + \dots + h(m_l) = p_1 + \dots + p_{l'}$. Since the output of the hash function is always n bits, and the strings p_i are also n bits, it follows that $l = l'$, and, more significantly, $h(m_i) = p_i$. Since $p_i = h(m_i)$, we have $h(m_i) = h(m)$. Since $m_i \neq m$, this is a collision for h . \square

The obvious disadvantage of the $2lM\mathcal{T}$ construction is that the Proof-of-Inclusions are of length $l \cdot n$ bits, which is inefficient; this is the main motivation for the use of the $M\mathcal{T}$ construction, which we show in the next subsection, whose PoI requires only $n \cdot \lceil \log(l) \rceil$ bits. Actually, the following exercise shows that a minor reduction is possible - changing only the PoI and $VerPoI$ functions.

Exercise 3.16. Show a (simple) change to the PoI and $VerPoI$ functions of the $2lM\mathcal{T}$ scheme, which reduce the length of the PoI to ‘only’ $n \cdot (l - 1)$ bits.

We next show the PoC for the $2lM\mathcal{T}$ construction. Again, it is very simple - which is our main motivation for showing it - but definitely suboptimal, with long Proof-of-Consistency. (The PoC of the $M\mathcal{T}$ construction is only $n \cdot \lceil \log l \rceil$ bits long - but we do not describe this PoC function.)

Definition 3.18 (Proof-of-Consistency for the Flat Merkle Tree). *Given a keyless function $h : \{0, 1\}^* \rightarrow \{0, 1\}^n$, We define the $2lM\mathcal{T}.PoC$ and $2lM\mathcal{T}.VerPoC$ functions as:*

$$\begin{aligned} 2lM\mathcal{T}.PoC(\{m_{C,1}, \dots, m_{C,l_C}\}, \{m_{N,1}, \dots, m_{N,l_N}\}) &\equiv \\ &\equiv h(m_{C,1}) + \dots + h(m_{C,l_C}) + h(m_{N,1}) + \dots + h(m_{N,l_N}) \end{aligned} \quad (3.39)$$

$$\begin{aligned} 2lM\mathcal{T}.VerPoC(\Delta_C, \Delta_{CN}, l_C, l_N, p) &\equiv \\ &\equiv (\Delta_C = h(p[1 : l_C \cdot n]) \wedge \Delta_{CN} = h(p)) \end{aligned} \quad (3.40)$$

Theorem 3.4. The Flat Merkle Tree PoC is correct and secure. ($2lM\mathcal{T}$) construction, with $2lM\mathcal{T}.PoC$ and $2lM\mathcal{T}.VerPoC$, has correct and secure PoC .

Proof: omitted. \square

When l , the number of messages/files, is large, then both proof-of-inclusion and proof-of-consistency may become long. We next present the ‘original’ Merkle tree ($M\mathcal{T}$) construction, which improves efficiency for this case (large l).

3.9.5 The Merkle tree \mathcal{MT} construction

We now present \mathcal{MT} , the ‘classical’ Merkle tree construction, first presented in Merkle’s seminal paper [230]. The \mathcal{MT} can be seen as the next logical step from the $2l\mathcal{MT}$ construction: it is a bit more complex: instead of using two layers, it uses $\lceil \log(l) \rceil$ layers, where l is the number of input strings. This allows better efficiency: the *PoI* contains only $\log(l)$ digests ($n \cdot \log(l)$ bits), and verifying it requires only $\log(l)$ hash operations.

We illustrate the computation of the digest in the \mathcal{MT} construction in Figure 3.19, and the computation of the *Proof-of-Inclusion* (*PoI*) in Figure 3.20, both for the simple case of a digest of a set of four messages ($m = 4$). As usual, for simplicity, we show and discuss only the keyless variant; it is not difficult to modify the construction for the case of keyed hash (assuming *ACR* or *TCR*, see subsection 3.2.3).

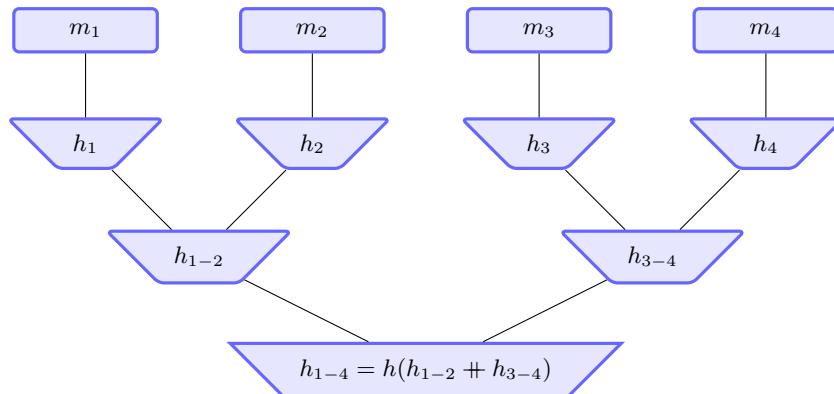


Figure 3.19: The *Merkle Tree* (\mathcal{MT}) construction, applied to input block of four messages $B \equiv \{m_1, m_2, m_3, m_4\}$. Let $h_i = h(m_i)$ denote the hash of a single message m_i , and h_{i-j} denote the \mathcal{MT} digest for messages m_i, \dots, m_j ; the complete digest is $\mathcal{MT}.\Delta(B) = h_{1-4} = h(h_{1-2} + h_{3-4})$. The figure does not indicate the use of the key, as for keyless \mathcal{MT} ; however the only change required for keyed version would be that all hashing is done using a common, random key, known to the attacker.

Except for the very bottom ‘layer’ hashing the input messages, all other hash applications have $2n$ -bit inputs to each hash function. In fact, Merkle’s original construction in [230] is the same, except that it did not have this ‘bottom’ layer, and simply assumed that the input messages are each only n bits - or, that the input is one long string broken down to chunks of n bits (providing another way to transform a compression function to a CRHF). In any case, let us focus on our variant; in the special case where inputs are strings of n bits each, one can simply eliminate one layer of hashing (i.e., use Merkle’s original construction).

For simplicity, assume that the number l of inputs is a power of two, say $l = 2^L$. It is easy to extend the construction to support any l , e.g., by padding.

This construction has $L + 1$ ‘layers’, which we number from $i = 0$ to $i = L$. The first layer, $i = 0$, is unique: it applies the hash function to each of the 2^L inputs. The output are 2^L strings of n bits each. The next layer hashes each consecutive pair of output of the previous layer, i.e., $2n$ -bits, resulting in 2^{L-1} string of n bits each. The process then repeats, with each layer i (from $i = 1$ to $i = L$) receiving $2^{L-(i-1)}$ strings from previous layer ($i - 1$), and outputting 2^{L-i} digests (n bits each). This continues, each time halving the total number of bits, until we remain with only the final n -bits digest.

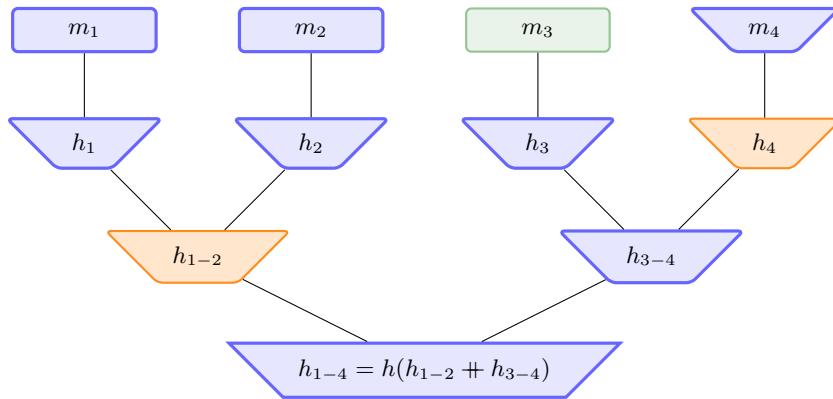


Figure 3.20: Example of Proof-of-Inclusion (PoI) in the *Merkle Tree* (\mathcal{MT}). The PoI for m_3 consists of $\{h_{1-2}, h(m_4)\}$, where $h_{1-2} = h(h(m_1) \# h(m_2))$. For simplicity, the figure (and the PoI equations) use a keyless hash; the only change required for keyed \mathcal{MT} would be to use a keyed hash instead.

The definition for the keyless \mathcal{MT} construction follows; we leave the definition of the keyed versions, using ACR and using TCR hash, to the reader. The definition is a bit ‘hairy’; Figure 3.19, Figure 3.20 and Exercise 3.17 may be helpful.

Definition 3.19 (The (keyless) *Merkle Tree* (\mathcal{MT}) construction). *Let $h : \{0, 1\}^* \rightarrow \{0, 1\}^n$ be a keyless hash function. We define the Merkle Tree (\mathcal{MT}) construction ($\mathcal{MT}.\Delta, \mathcal{MT}.PoI, \mathcal{MT}.VerPoI$) by Equation 3.41, Equation 3.42 and Equation 3.43 below. For simplicity, the number of messages l given to $\mathcal{MT}.\Delta$ has to be a power of two, i.e., $l = 2^L$ for some integer L .*

$$\mathcal{MT}.\Delta(m_1, \dots, m_l) \equiv \begin{cases} \text{If } l = 1 : & h(m_1) \\ \text{Else} & h(\mathcal{MT}.\Delta(m_1, \dots, m_{2^{L-1}}) \# \\ & \quad \# \mathcal{MT}.\Delta(m_{2^{L-1}+1}, \dots, m_{2^L})) \end{cases} \quad (3.41)$$

$$m\mathcal{T}.PoI(m_1, \dots, m_l, j)_{1 \leq j \leq l} \equiv$$

$$\begin{cases} \text{If } l = 1 : & (\text{empty string}) \\ \text{If } j > l/2 : & m\mathcal{T}.\Delta(m_1, \dots, m_{l/2}) \# \\ & \quad \# m\mathcal{T}.PoI[(m_{l/2+1}, \dots, m_l), j - l/2] \\ \text{Else :} & m\mathcal{T}.\Delta(m_{l/2+1}, \dots, m_l) \# \\ & \quad \# m\mathcal{T}.PoI[(m_1, \dots, m_{l/2}), j] \end{cases} \quad (3.42)$$

$$m\mathcal{T}.VerPoI(d, m, j, \{x_i\}_{i=1}^L) \equiv RVerPoI(d, h(m), j, \{x_i\}_{i=1}^L) \text{ (below)} \quad (3.43)$$

Where $RVerPoI$ is the following (recursive) function:

$$RVerPoI(d, x', j, \{x_i\}_{i=1}^L) \equiv$$

$$\begin{cases} \text{If } L = 1 \wedge j = 0 : & d = h(x_1 + x') \\ \text{If } L = 1 \wedge j = 1 : & d = h(x' \# x_1) \\ \text{If } j > 2^{L-1} : & RVerPoI(d, h(x_L + x'), \lfloor j/2 \rfloor, \{x_i\}_{i=1}^{L-1}) \\ \text{Else :} & RVerPoI(d, h(x' \# x_L), \lfloor j/2 \rfloor, \{x_i\}_{i=1}^{L-1}) \end{cases} \quad (3.44)$$

Exercise 3.17. This question is about digest and PoI for the Merkle tree scheme. For concreteness, we will also refer to the trivial (and insecure) h_{sum} function (Example 3.1).

1. Compute the Merkle-tree digest, for the input sequence/block $B_1 = \{10, 20\}$, as a formula for an arbitrary hash function h , and as a value for h_{sum} .

Solution:

$$m\mathcal{T}.\Delta(B_1) = h(h(10) + h(20)) \quad (3.45)$$

When using h_{sum} , we have $m\mathcal{T}.\Delta(B_1) = 3$.

2. Compute the Merkle-tree digest, for the inputs $B_2 = \{30, 40, 50, 60, 70, 80, 90, 100\}$. Present the digest as a formula for an arbitrary hash function h , and as a value for h_{sum} .

Solution:

$$\begin{aligned} m\mathcal{T}.\Delta(B_2) = & h[h(h(h(30) + h(40)) + h(h(50) + h(60))) + \\ & h(h(h(70) + h(80)) + h(h(90) + h(100)))] \end{aligned}$$

When using h_{sum} , we have $m\mathcal{T}.\Delta(B_2) = 7$.

3. Compute the PoI for the input value 50 in B_2 . Present the PoI as a formula for an arbitrary hash function h , and as a value for h_{sum} .

value 50 was the third input in B_2 so, using Equation 3.19, Figure 3.20, the PoI is:

$$\begin{aligned} \mathcal{MT}.PoI(B_2, 3) &= \mathcal{MT}.\Delta(\{70, 80, 90, 100\}) \# \mathcal{MT}.PoI(\{30, 40, 50, 60\}, 3) \\ &= h[h(h(70) \# h(80)) \# h(h(90) \# h(100))] \# \\ &\quad h(h(30) \# h(40)) \# h(60) \end{aligned}$$

For the special case of h_{sum} , we have $\mathcal{MT}.PoI(B_2, 3) = 7 \# 7 \# 6$.

As expected, the \mathcal{MT} construction results in a correct and secure \mathcal{MT} scheme.

Theorem 3.5. *The (keyless) Merkle Tree (\mathcal{MT}) construction (Fig. 3.19) is a correct and secure \mathcal{MT} scheme.*

Proof sketch: Similar to Theorem 3.3. □

\mathcal{MT} extensions: arbitrary l and PoC . We simplified the \mathcal{MT} construction by assuming that the number of messages in the input, l , is a power of two ($l = 2^L$). Under this assumption, it is also easy to define PoC and $VerPoC$ functions; we leave this as an exercise. We don't do it, since, with this assumption, the tree always doubles - which is rarely practical.

It is a bit more challenging to extend the \mathcal{MT} construction for arbitrary number of messages l (not a power of two), and to define the appropriate PoC , $VerPoC$ functions, and we will not do it here. It is not *very* difficult, so readers are encouraged to work out the details, and/or see published constructions, e.g., in [90, 106, 210].

Storage concerns. Computation of the Merkle-tree hash requires at least storage of one hash value for each layer of the tree, which is particularly inconvenient in hardware. This is one of the reason that this construction is not used to construct CRHFs from compression functions.

Security concern: fragile security for collisions. There is another concern with the use of Merkle tree, in particular for hashing: it is vulnerable to the discovery of a collision in the underlying hash (or compression) function. Given *any* collision of the underlying hash/compression function, we can find collisions for the entire Merkle-tree - for *any* given prefix and suffix.

3.10 Blockchains, Proof-of-Work (PoW) and Bitcoin

In this section, we discuss one of the most exciting applications of cryptographic hash functions: the *blockchain*. A *blockchain* is another integrity mechanism; this time, the input is organized as a *sequence of blocks*, where each block is a sequence of records. Indeed, blockchains are mostly based on a combination of

a *digest-chain* scheme and a *Merkle digest* scheme. However, there are many variants.

The basic integrity property of blockchains is *collision-resistance*; it is infeasible to find two different sequences of blocks, which will result in the same digest. However, blockchains usually have additional integrity properties, referred to as *immutability*, which ensures that all users will see the sequence of blocks in the blockchain - possibly except for some number of last blocks. Of course, this implies some control mechanism governing the addition of blocks to the blockchain; such blockchains are called *controlled blockchains*.

The entries in the blockchain may contain different data, e.g., transactions. In many applications, entries are signed, although this is not always the case. Often, controlled blockchains are used to track transfer of objects (or payments) between entities; such log of transfer of objects or funds between parties, is often referred to as a *ledger*. Blockchains allow maintenance of *public ledgers*, allowing everyone to audit and validate the ledger, and ensuring their integrity and immutability.

Anonymity? Many blockchain applications, e.g. Bitcoin, maintain some level of *anonymity*. In Bitcoin, the anonymity is provided by identifying the recipient of an object (or payment) using only their public validation key, without exposing their name or other identifier. For example, Alice, whose (validation, signing) key pair is $(A.v, A.s)$, may transfer object or payment x to an entity identified only by their public validation key v , by adding entry $Sign_{A.s}(v + x)$.

3.10.1 The blockchain digest scheme

Like the digest-chain and the Merkle digest schemes, the blockchain maintains a *digest* of the sequence of entries, in a way which ensures *integrity* of the sequence of entries, given the value of the digest. Namely, it is infeasible to add, remove, reorder or modify the entries, without changing the digest. We refer to this integrity property as *collision-resistance* of the blockchain.

Later in this subsection, we discuss additional blockchain integrity properties, in particular *Proof-of-Consistency*, which prevents an attacker from modifying the sequence of entries, except for appending new entries; and in subsection 3.10.2 we discuss *controlled blockchains*, which also control the process of adding blocks and entries to the blockchain, and can ensure immutability.

We focus on the most common construction of a blockchain digest scheme, which combines a digest-chain scheme with a Merkle digest scheme. Most properties - and definitely the collision-resistance property - follow from the corresponding properties of the digest-chain and Merkle-digest schemes, which, in turn, are due to the collision-resistance of the underlying hash function.

Blocks. In a blockchain, as the name implies, entries are added in groups called *blocks*, which are added to the blockchain sequentially (hence, a *chain*).

The mapping of entries to blocks is also protected by the digest: it is infeasible to move an entry from one block to another, while retaining the same digest. Even the last entry of one block cannot be ‘moved’ to become the first entry of the next block, or vice versa. This property can be useful, e.g., to map all operations in the block to a particular period.

Often, blockchains use a Merkle scheme - often, the Merkle tree - to compute the digest within each block, and then use a digest-chain scheme, typically the Merkle-Damgård scheme, to compute the digest of multiple blocks. See Figure 3.21.

Blockchain digest schemes focus on ensuring the *integrity* of the blockchain, and of particular entries in it. Like for CRHF and Merkle digest schemes, integrity is *validated using the digest*, i.e., integrity is assured with respect to a *blockchain digest*.

However, the blockchain digest scheme, like the other digest schemes (and like a CRHF), does not include mechanisms to *validate the digest* itself. Suppose that currently, the blockchain contains the sequence $\{B_1, B_2, \dots, B_l\}$ of blocks, resulting in digest $\Delta_l \equiv \mathcal{B}.\Delta(B_1 + \dots + B_l)$ (Figure 3.21). An attacker wishing to make a change in the blocks, say to blocks $\{B'_1, B'_2, \dots, B'_{l'}\}$, can simply replace the current digest, Δ_l , with the fake digest $\Delta'_{l'} \equiv \mathcal{B}.\Delta(B'_1 + \dots + B'_{l'})$. We next discuss additional mechanisms, which ensure the integrity of the digest itself, thereby ensuring immutability of the sequence of blocks.

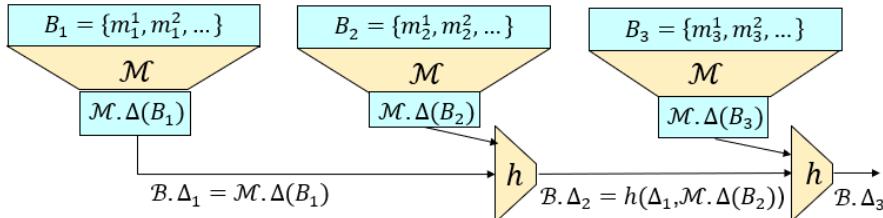


Figure 3.21: A basic blockchain: sequence of digests $\Delta_1, \Delta_2, \dots$ of a sequence of blocks $B_1 = \{m_1^1, m_1^2, \dots\}, B_2 = \{m_2^1, \dots\}, \dots$

Blockchain digest schemes, like Merkle digest schemes, support two additional integrity mechanisms: *Proof-of-Inclusion (PoI)* and *Proof-of-Consistency (PoC)*.

Proof-of-Inclusion (PoI). A PoI proves that a particular entry is included in the blockchain, and identifies its block and sequence number within the block. However, there is a significant difference in the inputs and outputs required to create the PoI and to verify it, as defined in the Merkle digest scheme and in the blockchain digest scheme:

Merkle scheme PoI is produced based on the entire sequence of messages, although Validation requires only the specific message and its sequence number, rather than the entire set of messages. In \mathcal{MT} and most other

efficient constructions, the PoI is of size $O(n \cdot \log l)$ bits and validation requires $O(\log l)$ hash operations.

Blockchain PoI is produced based only on the messages in the same block, and on the digest Δ_{pre} of the blockchain *before* this block was added. Verification requires only the specific message, its sequence number within the block, and the previous digest Δ_{pre} .

Proof-of-Consistency (PoC). A PoC is used to validate that a newly received digest, is of a sequence whose prefix is a previous digest. Again, there are important differences in the inputs and outputs of the PoC, between blockchains and Merkle schemes:

Merkle scheme PoC is produced based on both the original sequence of entries, and the entries extending it. In \mathcal{MT} and most other efficient constructions, the PoC is of size $O(n \cdot \log l)$ bits and validation requires $O(\log l)$ hash operations.

Blockchain PoC requires only the digest of the of the blockchain before this block, and the entries in the new block. The PoC is typically only n bits (one digest, essentially), and validation requires only *one* hash operation.

There is one more concern for the integrity of the blockchain: can we prevent an attacker from *appending* fraudulent data onto the blockchain? Blockchains that defend against fraudulent appending are called *controlled blockchains*, and are the subject of the next subsection.

3.10.2 Controlled blockchains: permissioned and permissionless

In the rest of this section, we present controlled blockchains - *permissioned* and *permissionless*, the *Bitcoin* blockchain-based *cryptocurrency*, and finally, *Proof-of-Work (PoW)*, a crucial component of Bitcoin and many other blockchains, usually implemented using hash functions.

Our discussion of blockchains so far did not involve any restrictions or controls over the blocks added to a blockchain. However, many applications, e.g. Bitcoin and other crypto-currencies, require a mechanism that controls the addition of new blocks to the blockchain. In fact, when people refer to ‘blockchains’, they usually mean a controlled blockchain. Different control mechanisms were proposed; they mostly fall into one of the following two categories:

Permissioned blockchains, where only specific, authorized parties can add a block to the blockchain. In a typical permissioned blockchain, every blockchain digest must be signed using the private signing key of an *authority*. The relying parties should know the corresponding authority signature-validation key $v_{Authority}$. Such blockchain may support multiple

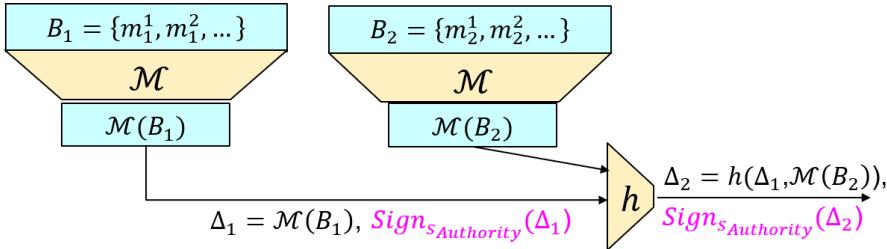


Figure 3.22: Permissioned Blockchain: blocks added only with *permission*-signature which is validated with known *authority* signature-validation key $v_{Authority}$. If there are multiple authorities, use *consensus* to avoid conflicts.

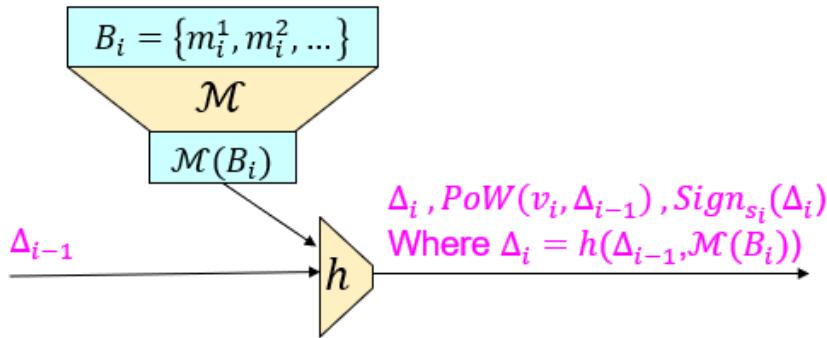


Figure 3.23: The permissionless Bitcoin Blockchain uses Proof-of-Work (PoW) to control the addition of new blocks, in a process called *mining*.

authorities, by using a *consensus* protocol to determine the authority to sign each block. See, e.g., [2].

Permissionless blockchains, where *any* party may, in principle, add a block to the blockchain, as long as it ‘wins’ in a *fair game*, which limits the issuing of new blocks. To ensure that parties will ‘compete’ in these games and issue new blocks, such schemes include some *reward* to the party that ‘wins the game’ and issues a new block, such as by a payment (e.g., in Bitcoin); hence, the process of competing in the game and issuing new blocks is referred to as *mining*. The two main control mechanisms are *Proof-of-Work (PoW)* and *Proof-of-Stake*. The *Bitcoin* blockchain uses PoW, as shown in Figure 3.23, and therefore we focus on PoW.

3.10.3 Proof-of-Work (PoW) schemes

In Bitcoin, issuing a new block requires a solution to a difficult computational problem, namely, a *Proof-of-Work (PoW)*. While this may be the only application of PoW that we will explore in this volume, there are many more

applications of PoW in Cybersecurity, and we will discuss some of them in the next volume [161], mainly with respect to Denial-of-Service attacks.

Intuitively, a PoW allows one party, the *worker*, to *solve a challenge*, with an *approximately known amount of computational resources*, resulting in a *proof* of this success, which can be efficiently verified by anyone.

Proofs-of-Work schemes belong in this chapter of hash functions, both due to their use in permissionless blockchains and in particular Bitcoin, but also since their most well known implementation - and the one used in Bitcoin - is based on a hash function.

Notice that we used the general term ‘computational resources’ and not a more specific term such as *computation-time*; the reason is that some PoW proposals focus on other resources, e.g., storage, or on a combination of resources, e.g., time and storage. However, from this point, let us focus on the case where provision of the PoW requires an approximately known amount of computation time. This is the most widely used form of PoW - and in particular, the one used in Bitcoin, which, as we mentioned (and will show), is based on the use of a cryptographic hash function.

Like our definitions of hash functions and of the schemes in this chapter, PoW schemes may be keyed or keyless ; and like for the other schemes, we focus on the keyless variant.

We define a PoW scheme with respect to a specific computational model \mathcal{M} and ‘challenge domain’ C_D . As mentioned above, our definition is limited to keyless PoW based on computational time.

Definition 3.20 (Proof of Work (PoW)). *A PoW scheme $\mathcal{P}oW$ consists of two efficient algorithms: $\mathcal{P}oW.solve$, and $\mathcal{P}oW.validate$.*

$\mathcal{P}oW.solve$: The $\mathcal{P}oW.solve$ algorithm receives three inputs: a challenge $c \in C_D$, a randomizer $r \in \{0, 1\}^n$, and a work-amount $w \in \mathbb{N}$ s.t. $1 \leq w \leq n$. The $\mathcal{P}oW.solve$ function outputs an n -bit binary string, to which we refer as the solution.

$\mathcal{P}oW.Validate$: The $\mathcal{P}oW.Validate$ algorithm has four inputs: the challenge c , the randomizer r , the work-amount w (all as described for $\mathcal{P}oW.Solve$), and a purported Proof-of-Work $\pi \in \{0, 1\}^$. The $\mathcal{P}oW.Validate$ algorithm returns TRUE or FALSE.*

A PoW scheme $\mathcal{P}oW$ is correct, if the runtime of $\mathcal{P}oW.solve(c, r, w)$, using computational model \mathcal{M} , is always at most w , and if for every challenge $c \in C_D$ and randomizer $r \in \{0, 1\}^n$ holds:

$$\mathcal{P}oW.Validate(c, r, w, \mathcal{P}oW.solve(c, r, w)) = \text{TRUE} \quad (3.46)$$

A PoW scheme $\mathcal{P}oW$ is α -secure, for $\alpha \in (0, 1]$ if for any algorithm \mathcal{A} whose average runtime (using computational model \mathcal{M}) on input (r, w) , where

$r \xleftarrow{\$} \{0,1\}^n$, is less than $\alpha \cdot w$, holds: $\varepsilon_{\mathcal{A}, \mathcal{P}oW}^{PoW}(n) \in NEGL(n)$, where:

$$\varepsilon_{\mathcal{A}, \mathcal{P}oW}^{PoW}(n) \equiv \Pr_{r \xleftarrow{\$} \{0,1\}^n} \{(\exists c \in C_D) \mathcal{P}oW.Validate(c, r, w, \mathcal{A}(c, r, w)) = \text{TRUE}\} \quad (3.47)$$

Proof-of-work mechanisms are often implemented using a cryptographic hash function. A typical implementation, which is a simplification of the one in Bitcoin, is the PoW scheme $\mathcal{P}oW^B$, defined as:

$\mathcal{P}oW^B.Validate$: On input (c, r, w, π) , return TRUE if $h(c + r + \pi) \leq \frac{2^{n-1}}{w}$. Otherwise, return FALSE.

$\mathcal{P}oW^B.solve$: On input (c, r, w) , repeatedly compute $x \equiv h(c + r + \pi)$ for different $\pi \xleftarrow{\$} \{0,1\}^n$, aborting and returning π when $x \leq \frac{2^{n-1}}{w}$.

We leave the properties of $\mathcal{P}oW^B$ as an exercise to the reader.

Exercise 3.18. 1. Show if $\mathcal{P}oW^B$ correct or not.

2. (harder) Present an example of a hash function h , which is collision resistant, and yet for which $\mathcal{P}oW^B$ is not α -secure, for any constant $\alpha > 0$.
3. (harder) Show if $\mathcal{P}oW^B$ is secure under the Random Oracle Model.

3.10.4 The Bitcoin Blockchain and Cryptocurrency

One of the important, interesting and controversial applications of blockchains is for *cryptocurrency*. Bitcoin is the most well-known cryptocurrency, based on a permissionless blockchain. In particular, Bitcoin is probably the most important application of PoW, which are key to the operation - and economics - of Bitcoin, and in particular, of the Bitcoin *mining process*, which is the main Bitcoin mechanism we discuss in this subsection.

Bitcoin is using the ‘classical’ combination of a Merkle-tree digest construction for computing a digest of each block, with the Merkle-Damgård digest-chain construction for appending new blocks in a controlled manner, as illustrated in Figure 3.23. Bitcoin is a *cryptocurrency*; the entries in the blockchain represent movement of funds. Funds, in Bitcoin, are associated with a *public (signature validation) key*, rather than with an account number or other identifier; in that sense, Bitcoin provides some level of *anonymity*, or, more precisely, *pseudonymity*, since this mechanism does not prevent *linkage* between different transactions using the same key.

The Bitcoin control mechanism: Mining. One of the innovations of Bitcoin, is its permissionless mechanism for adding blocks to the blockchain - which is also the mechanism used to gradually increase the total number of available Bitcoins.

BitCoin mining uses the $\mathcal{P}oW^B$ Proof-of-Work design, presented in subsection 3.10.3. As shown in Figure 3.23, if the previous (current) digest is Δ_{i-1} , then to add a block, the new digest Δ_i must be signed by private key s_i , where the corresponding public key is included in a PoW solved by the entity adding the new block. The PoW is computed over both the public key v_i , and the previous digest Δ_{i-1} . The inclusion of the previous digest, ensures that the ‘competition’ for solving the PoW (and mining) may begin only once the previous digest is known.

Of course, solving PoW requires considerable computational resources, mainly energy. This use of energy for mining is one of the criticisms against Bit-coin, motivating the use of blockchains which are more energy-efficient, including permissioned blockchains and blockchains using other mining mechanisms, e.g., *proof of stake*.

To motivate miners to invest resources and solve the PoW, Bitcoin offers two incentives. The first incentive is a *reward*: every time a miner succeeds in adding a new block to the chain, she is ‘rewarded’ with a number of Bitcoins. The number of Bitcoins awarded to the lucky miner, is computed by a clever ‘reward rule’, designed to motivate miners sufficiently, but not excessively. This is also the (only) process of adding new Bitcoins, which is one reason that this process is referred to as *mining*, maybe with a gold-mine in mind: many miners work hard, only few lucky ones find gold. This reward is cut in a half once per 210,000 blocks mined.

The second incentive to miners is *transaction fees*. It is hoped that transactions fees would provide sufficient incentive to motivate miners, even when rewards are cut in half, as the usage of bitcoin grows. Every transaction indicates a fee, which is an amount paid to the miner of the block in the chain which includes this transaction; this is in addition to the amount of bitcoins transferred to the payee. Different transactions offer different fees, and miners may prefer transactions with higher fees; each block has limited size, so miners may not be able to include all available transactions in their blocks. Both reward and fees are ad to the balance associated with the public key of the miner, which is part of the new block, and controlled by the corresponding private key, known to the miner.

The work-amount parameter w of the PoW (not shown in Figure 3.23), is adjusted automatically by a feedback mechanism in Bitcoin, whose goal is to ensure that new blocks will be added at a *reasonable*, but not excessive, rate. Namely, if blocks are added more quickly, then the work amount is increased - making it harder to mine new blocks, i.e., slowing down the rate of adding blocks. This should maintain a stable rate of mining new blocks, and therefore, balancing between the overhead of block creation, and the delay until a new transaction appears on the chain. The mining rates can be impacted by multiple factors, including the energy costs of mining and the value of the award and fees obtained by a lucky miner, the likelihood of mining a block added to the chain (which depends on competition), and the costs and efficiency of mining hardware.

Tracing funds in Bitcoin. One element of Bitcoin which is less advanced, is the method of tracing funds. Namely, in order to know the number of Bitcoins owned by a specific public key at a given time, as required in order to confirm a payment, it is necessary to trace-back all the transactions, identifying any operations that involve this key. Namely, Bitcoin does not have an efficient way to validate the amounts ‘owned’ by each public key, using an efficient process such as a Proof-of-Inclusion of an entry in a digest.

There are many additional concerns and mechanisms of Bitcoin that we will refrain from discussing, such as the fact that multiple ‘first blocks’ may exist for the chain at the same time - or how Bitcoin attempts to manage this concern. Indeed, obviously, our discussion of Bitcoin, cryptocurrencies and blockchains is superficial. There are many publications, including books, that should be used to study these areas further.

3.11 Lab and additional exercises

Lab 3 (CRC-hash Collisions). *In this lab, we try to use the CRC-32 error-detection code as a hash function. Namely, to hash input x , we compute $\text{CRC}(x)$, where CRC is the CRC-32 function. We show that CRC is not a secure hash function, specifically, we find collisions, i.e., inputs $x \neq x'$ such that $\text{CRC}(x) = \text{CRC}(x')$. Finally, we create a partially-chosen collision, i.e., a collision between the CRC-hash over two given (and very different) documents, by ‘filling in’ a designated area left undefined in each of the two documents. Note that this implies universal forgery of signatures computed using the Hash-then-Sign construction with CRC-32 as the hash function.*

As for the other labs in this textbook, we will provide Python scripts for generating and grading this lab (`LabGen.py` and `LabGrade.py`). If not yet posted online, professors may contact the author to receive the scripts. The lab-generation script generates random challenges for each student (or team), as well as solutions which will be used by the grading script. We recommend to make the scripts available to the students, as example of how to use the cryptographic functions. It is easy and permitted to modify these scripts to use other languages/libraries or to modify and customize them as desired.

1. Write a program to compute the CRC-32. In you lab-input folder, find files $f1a.txt$, $f1b.txt$ and $h1a.txt$. File $h1a.txt$ is the CRC-32 hash of file $f1a.txt$; use it to test your program. Then, compute the CRC-32 hash of $f1b.txt$; name the resulting file $h1b.txt$ and place it in the lab-solutions folder.
2. Find and submit a collision for CRC-32. Namely, place in the lab-solutions folder three files, $f2a.txt$, $f2b.txt$ and $h2.txt$, such that $h2.txt$ is the CRC-32 hash of *both* files ($f2a.txt$ and $f2b.txt$).
3. In the lab-input folder, find files $f3a.html$ and $f3b.html$. Open both files in the browser; you will see that the contents of these pages are very

different. Open new the files in a text editor; you will see a designates ‘allowed to change’ area in each of the two files. Find how to change *only* the designated ‘allowed to change’ areas in these files, so that the resulting files, denoted *f3as.html* and *f3bs.html* will be identical with the original files except for the designated areas, will display the same content as the original files, and yet, these two files will have *the same CRC-32 hash*. Upload *f3as.html* and *f3bs.html* to the lab-solutions folder.

Exercise 3.19 (XOR-hash). *Consider the following hash function, defined for input messages consisting of number l of n bits ‘blocks’, i.e., total of $l \cdot n$ bits, where n is the length of the digest ($n = |h(m)|$). Given such message m containing $l \cdot n$ bits, let us denote the i^{th} block (of n bits) by m_i , i.e., $m = m_1 \# m_2 \# \dots \# m_l$ and $(\forall i)|m_i| = n$.*

Define hash function h for such $l \cdot n$ bit messages, as: $h(m_1 \dots m_l) = \bigoplus_{i=1}^l m_i$. Show that h does not have each of the following properties, or present a convincing argument why it does:

1. Collision-resistance (CRHF), see Section 3.2.
2. Second-preimage resistance (SPR), see Section 3.3.
3. Preimage resistance, i.e., h is not a one-way function (OWF), see Section 3.4.
4. Bitwise randomness extraction (BRE), see subsection 3.5.2.
5. Secure MAC, when h is used in the HMAC construction, see subsection 4.6.3.

Solution to part 4 (randomness extraction): For simplicity, we present the solution for even n , i.e., $n = 2\mu$ where μ is an integer. Adversary \mathcal{A} selects input message $m = 0^{2n}$ and mask $M = 0^\mu \# 1^\mu \# 0^\mu \# 1^\mu$. Let y_b and y_{1-b} be the values computed by $\text{BRE}_{A,h}(b, n)$; the reader can confirm that y_{1-b} is random, while y_b is of the form $0^\mu \# r$, where r is a random string of μ bits. On input (y_0, y_1, m, M) , the adversary \mathcal{A} returns:

$$\mathcal{A}(y_0, y_1, m, M) = \begin{cases} 0 & \text{if } m \bmod 2^\mu \neq 0^\mu \\ 1 & \text{otherwise} \end{cases} \quad (3.48)$$

The reader should confirm that the adversary is correct with overwhelming probability. Hence, h is not a bitwise randomness extractor (BRE). \square

Exercise 3.20. *Let h be a ‘compression function’, i.e., a cryptographic hash function whose input is of length $2l$ and output is of length l . Let $h' : \{0,1\}^{2l} \rightarrow \{0,1\}^l$ extend h to inputs of length $2l \cdot n$, as follows: $h'(m_1 \# \dots \# m_n) = \bigoplus_{i=1}^n h(m_i)$, where $(\forall i = 1, \dots, n)|m_i| = 2l$. For each of the following properties, assume h has the property, and show that h' may not have the same property. Or, if you believe h' does retain the property, argue why it does. The properties are:*

1. Collision-resistance.
2. Second-preimage resistance.
3. One-wayness (preimage resistance)
4. Randomness extraction.

Would any of your answers change, if h and h' have a random public key as an additional input?

Exercise 3.21. Consider messages of $2n$ blocks of l bits each, denoted $m_1 \dots m_n$, and let h_c be a secure compression function, i.e., a cryptographic hash function from $2n$ bits to l bits. Define hash function h for such $2n$ blocks of l bits messages, as: $h(m_1 \dots m_{2n}) = \bigoplus_{i=1}^n h_c(m_{2i}, m_{2i-1})$. Show that h does not have each of the following properties, although h_c has the corresponding property, or present a convincing argument why it does:

1. Collision-resistance.
2. Second-preimage resistance.
3. One-wayness (preimage resistance)
4. Bitwise randomness extraction.
5. Secure MAC, when h is used in the HMAC construction.

Exercise 3.22. It is proposed to combine two hash functions by cascade, i.e., given hash functions h_1, h_2 we define $h_{12}(m) = h_1(h_2(m))$ and $h_{21}(m) = h_2(h_1(m))$. Suppose collision are known for h_1 ; what does this imply for collisions in h_{12} and h_{21} ?

Exercise 3.23. Recently, weaknesses were found in few cryptographic hash functions such as h_{MD5} and h_{SHA1} , and as a result, there were many proposals for new functions. Dr. Simpleton suggests to combine the two into a new function, $h_c(m) = h_{SHA1}(h_{MD5}(m))$, whose output length is 160 bits. Prof. Deville objects; she argued that hash functions should have longer outputs, and suggest a complex function, h_{666} , whose output size is 666 bits. A committee setup to decide between these two, proposes, instead, to XOR them into a new function: $f_X(m) = [0^{506} \# h_c(m)] \oplus h_{666}(m)$.

1. Present counterexamples showing that each of these may not be collision-resistant.
2. Present a design where we can be sure that finding a collision is definitely not easier than finding one in h_{SHA1} and in h_c .
3. Repeat the first part for bitwise randomness-extraction.

Exercise 3.24. Let h be the result of a Merkle hash tree, using a compression function $\text{comp} : \{0,1\}^{2n} \rightarrow \{0,1\}^n$, and let (KG, S, V) be a secure (FIL) signature scheme. Let $S_s^h(m) = S_s(h(m))$ follow the ‘hash then sign’ paradigm, to turn (KG, S, V) into a VIL signature scheme, i.e., allow signatures over messages of arbitrary number of blocks. Show that S_s^h is not a secure signature scheme, by presenting an efficient adversary (program) that outputs a forged signature.

Exercise 3.25. Consider the following slight simplification of the popular HMAC construction: $h'_k(m) = h(k \# h(k \# m))$, where $h : \{0,1\}^* \rightarrow \{0,1\}^n$ is a hash function, k is a random, public n -bit key, and $m \in \{0,1\}^*$ is a message.

1. Assume h is a CRHF. Is h'_k also a CRHF?

Yes. Suppose h'_k is not a CRHF, i.e., there is some adversary \mathcal{A}' that finds a collision (m'_1, m'_2) for h' , i.e., $h'_k(m'_1) = h'_k(m'_2)$. Then at least one of the following pairs of messages $(m_{1,1}, m_{2,1})$, $(m_{1,2}, m_{2,2})$ is a collision for h , i.e., either $h(m_{1,1}) = h(m_{2,1})$ or $h(m_{1,2}) = h(m_{2,2})$ (or both). The strings are: $m_{1,1} = \underline{\hspace{1cm}}$, $m_{1,2} = \underline{\hspace{1cm}}$, $m_{2,1} = \underline{\hspace{1cm}}$, $m_{2,2} = \underline{\hspace{1cm}}$.

No. Let \hat{h} be some CRHF, and define $h(m) = \underline{\hspace{1cm}}$. Note that h is also a CRHF (you do not have to prove this, just to design h so this would be true and easy to see). Yet, h'_k is not a CRHF. Specifically, the following two messages $m'_1 = \underline{\hspace{1cm}}$, $m'_2 = \underline{\hspace{1cm}}$ are a collision for h'_k , i.e., $h'_k(m_1) = h'_k(m_2)$.

2. Assume h is an SPR hash function. Is h'_k also SPR?

Yes. Suppose h'_k is not SPR, i.e., for some l , there is some algorithm \mathcal{A}' which, given a (random, sufficiently-long) message m' , outputs a collision, i.e., $m'_1 \neq m'$ s.t. $h'_k(m') = h'_k(m'_1)$. Then we define algorithm \mathcal{A} which, given a (random, sufficiently long) message m , outputs a collision, i.e., $m_1 \neq m$ s.t. $h_k(m) = h_k(m_1)$. The algorithm \mathcal{A} is:

Algorithm $\mathcal{A}(m)$:

```
{
Let  $m' = \underline{\hspace{1cm}}$ 
Let  $m'_1 = \mathcal{A}'(m')$ 
Output  $\underline{\hspace{1cm}}$ 
}
```

No. Let \hat{h} be some SPR, and define $h(m) = \underline{\hspace{1cm}}$. Note that h is also an SPR (you do not have to prove this, just to design h so this would be true and easy to see). Yet, h'_k is not an SPR. Specifically, given a random message m' , then $m'_1 = \underline{\hspace{1cm}}$ is a collision, i.e., $m' \neq m'_1$ yet $h'_k(m'_1) = h'_k(m'_2)$.

3. Assume h is a OWF. Is h'_k also a OWF?

Yes. Suppose h'_k is not OWF, i.e., for some l , there is some algorithm \mathcal{A}' which, given $h'_k(m')$ for a (random, sufficiently-long) message m' , outputs a preimage, i.e., $m'_1 \neq m'$ s.t. $h'_k(m') = h'_k(m'_1)$. Then we define

algorithm \mathcal{A} which, given $h(m)$ for a (random, sufficiently long) message m , outputs a preimage, i.e., m_1 s.t. $h_k(m) = h_k(m_1)$. The algorithm \mathcal{A} is:

Algorithm $\mathcal{A}(m)$:

{

Let $m' = \underline{\hspace{2cm}}$

Let $m'_1 = \mathcal{A}'(m')$

Output $\underline{\hspace{2cm}}$

}

\square No. Let \hat{h} be some OWF, and define $h(m) = \underline{\hspace{2cm}}$. Note that h is also an OWF (you do not have to prove this, just to design h so this would be true and easy to see). Yet, h'_k is not an OWF. Specifically, given a random message m' , then $m'_1 = \underline{\hspace{2cm}}$ is a collision, i.e., $m' \neq m'_1$ yet $h'_k(m'_1) = h'_k(m')$.

4. Repeat similarly for bitwise randomness extraction.

Exercise 3.26 (Hash chain). In Exercise 3.8 we presented a hash function g which is a OWF (assuming an underlying function h is a OWF), yet, a hash chain using g would be vulnerable.

1. Assume that h is also known to be a SPR hash function. Would g , constructed from h as in Equation 3.17 be a SPR hash function too?
2. Extend the solution to Exercise 3.8 to show that the hash chain construction may be vulnerable, even when using a hash function which is both a OWF and a SPR.
3. Recall the improved hash chain design proposed by Micali and Leighton, i.e., to compute x_i as: $x_i = h(x_{i-1}||i)$. Obviously, h cannot be a permutation over its inputs; present an alternative assumption which suffices to make this design secure.
4. Challenge: is true that the hash chain construction may be vulnerable, even when using a hash function which is both a OWF and a CRHF? Prove it or present a counterexample.

Exercise 3.27 (Improved hash chain). Recall the improved hash chain design proposed by Micali and Leighton, i.e., to compute x_i as: $x_i = h(x_{i-1}||i)$. Obviously, h cannot be a permutation over its inputs; present an alternative assumption which suffices to make this design secure.

Exercise 3.28. Consider the following construction: $h'_k(m) = h(k + m)$, where $h : \{0, 1\}^* \rightarrow \{0, 1\}^n$ is a hash function, k is a secret n -bit key, and $m \in \{0, 1\}^*$ is a message. Assume you are given some SPR hash function $\hat{h} : \{0, 1\}^* \rightarrow \{0, 1\}^{\hat{n}}$; you can use \hat{n} which is smaller than n . Using \hat{h} , construct hash function h , so that (1) it is ‘obvious’ that h is also SPR (no need to prove), yet (2) $h'_k(m) = h(k + m)$ is (trivially) not a secure MAC. Hint: design h s.t. it becomes trivial to find k from $h'_k(m)$ (for any m).

1. $h(x) = \underline{\hspace{2cm}}$.
2. (Justification) h is an SPR, since $\underline{\hspace{2cm}}$.
3. (Justification) $h'_k(m) = h(k \# m)$ is not a secure MAC, since: $\underline{\hspace{2cm}}$.

Exercise 3.29 (HMAC is secure under ROM). *Show that the HMAC construction is secure under the Random Oracle Model (ROM), when used as a PRF, MAC and KDF.*

Exercise 3.30 (HMAC is insecure using CRHF). *Show counterexamples showing that even if the underlying hash function h is collision-resistant, its (simplified) HMAC construction $hmac_k(x) = h(k \# h(k \# m))$ is insecure when used as any of PRF, MAC and KDF.*

Exercise 3.31 (Hash-tree with efficient proof of non-inclusion). *The Merkle hash-tree allows efficient proof of inclusion of a leaf (data item) in the tree. Present a variant of this tree which allows efficient proof of either inclusion or of non-inclusion of an item with given ‘key’ value. In this tree, each item consists of two strings, a key and data. Assume all data items are given together, sorted by their key values; no need to build the tree dynamically or extend it. Your solution may ‘expose’ one or two additional data items beyond the one queried. Note: try to provide solution which is efficient in number of hash operations required for verification (the number should be about one more than in the regular Merkle tree).*

Hint: You can see example of proof of non-inclusion and its application in the NSEC3 record of DNSSEC (RFC 5155 [212]), and a graphical illustration in Figure 8.19.

Exercise 3.32 (Hash-tree with efficient proof of consistency). *In this exercise, we extend the $M\mathcal{T}$ construction (Merkle hash-tree) to provide efficient proof of consistency.*

1. Define the appropriate PoC, VerPoC functions.
2. Show that the pair of functions you defined ensures correctness.
3. Show that the pair of functions you defined ensures secure proof of consistency (secure PoC).
4. Show that the functions are efficient.

Hint: You may see published constructions, e.g., in [90, 106, 210]. However, you may also just do it yourself, it is not that hard and may be more fun. \square

Chapter 4

Authentication: Message Authentication Code (MAC), Blockchain and Signature Schemes

Cybersecurity and cryptography address different goals related to threats to information and communication. The most well-known goals are *confidentiality*, *integrity*, *authentication* and *availability*. In cryptography, the terms *integrity* and *authentication* are used mostly as synonyms, both meaning the validation that communication and information comes from a specific entity, or from one of a specific set of entities, leaving us with the sassy acronym CIA, often referred to as the CIA triad, for *confidentiality*, *integrity*, *authentication* and *availability*¹

So far, we have mostly focused on confidentiality, to which we dedicated all of Chapter 2. We also briefly introduced, in subsection 1.4.1, *signature schemes*, which are *asymmetric (public key)* authentication schemes. In this chapter, we discuss *symmetric (shared key)* authentication schemes, called *Message Authentication Code (MAC)*. MAC schemes efficient - much more than comparably-secure signature schemes.

People often expect that encryption will ensure authenticity as a side-product of ensuring confidentiality. Therefore, let us begin this chapter by discussing the use of encryption for authentication, and show that this can be vulnerable - although, later, in Section 4.7, we also discuss *authenticated encryption* schemes, designed to ensure, indeed, both confidentiality and authenticity.

¹Originally, the ‘A’ in *CIA* referred to *authentication*; indeed, in computer security, ‘integrity’ has a different meaning: protection of a computer or other system from corruption. ‘Availability’ was later identified as another basic goal, with the less-sassy acronym *CIAA*. Both acronyms are also used sometimes with *accountability* replacing *availability* and/or *authentication*. Oh well, all worthy goals!

4.1 Encryption for Authentication?

As we discussed in the previous chapter, encryption schemes ensure *confidentiality*, i.e., an attacker observing an encrypted message (ciphertext) cannot learn anything about the plaintext (except its length). Sometimes, people expect encryption to also be useful for *authentication* and *integrity*.

Some encryption schemes indeed have some integrity properties. One important properties is *non-malleable* encryption. Intuitively, a non-malleable encryption scheme prevents the attacker from *modifying* the message in a ‘meaningful way’. See definition and secure constructions of non-malleable encryption schemes in [104]. However, be warned: achieving, and even defining, non-malleability is not as easy as it may seem!

In fact, many ciphers are *malleable*; often, an attacker can easily modify a known ciphertext c , to $c' \neq c$ s.t. $m' = D_k(c') \neq m$ (and also $m' \neq \text{ERROR}$). Furthermore, often the attacker can ensure useful relations between m' and m . An obvious example is when using the (unconditionally-secure) one-time-pad (OTP), as well as using Output-Feedback (OFB) mode.

Exercise 4.1. *Mal is a Man-in-the-Middle, able to intercept and modify messages from Alice to her bank. In this question we explore the ability of Mal to modify ciphertext (encrypted message) message which Alice sends to her bank. Alice’s message is composed of the following fields, in the given order, each consisting of eight bytes: operation, reason, amount, payee, payer, password, date. When operation= 1 and password contains the correct (four-bytes) password for payer, the bank transfers amount from payer to payee, listing it in the bank ledger with the given (four-bytes) reason. Use identifiers 1 for Alice, 2 for Bob, and 3 for Mal.*

1. Suppose the parties use One-Time-Pad (OTP) encryption, and Mal intercept the ciphertext c sent from Alice to her bank, which is encryption of a request to transfer \$ 3 to Bob. Explain how Mal can modify the ciphertext, causing the bank to transfer (preferably, a larger amount) to Mal rather than Bob. Assume Mal knows all the details (amount, payee, etc.).
2. How would your response change when using, instead of OTP, the following modes of operation of DES, with block size of 64 bits (8 bytes): (a) OFB, (b) CFB, (c) CBC, (d) CTR, (e) ECB. Note: You may not be able to find a successful attack for some of the modes - but don’t give up too easily!

Solution for first part: with OTP, the i^{th} ciphertext bit is computed by $c_i = m_i \oplus k_i$, and decrypted by $m_i = c_i \oplus k_i$. Therefore, i is encrypted by c_i

We conclude that encryption schemes may not suffice to ensure authentication. This motivates us to introduce, in the next section, another symmetric-key cryptographic scheme, which is designed explicitly to ensure authentication and integrity: the *Message Authentication Code (MAC)*. Later, in Section 4.7, we also discuss how to achieve confidentiality together with authenticity.

4.2 Message Authentication Code (MAC) schemes

Message Authentication Code (MAC) schemes are a simple, symmetric key cryptographic functions, designed to verify the authenticity and integrity of information (messages), namely, to detect that a message was not sent by an ‘allowed sender’ (or, was modified after it was sent). A MAC function $MAC_k(m)$ has two inputs, a (secret) *n-bit secret (symmetric) key* k , and a message m . As illustrated in Figure 4.1, MAC schemes use the same key k to *generate* the authenticator (tag), and to *validate* the authenticator (tag); usually, upon receiving a message m with a purported authenticator σ , the recipient computes $\sigma' \leftarrow MAC_k(m)$ and verifies m by confirming that $\sigma = \sigma'$. Notice that this implies that MAC schemes, and authenticators, are usually *deterministic*. Intuitively, given m , $MAC_k(m)$ for a secret, random key k , it is infeasible for a (computationally-bounded) attacker to find another message $m' \neq m$ together with the value of $MAC_k(m')$.

Typically, as shown in Fig. 4.1, a secret, symmetric MAC key k is shared between two (or more) parties. Each party can use the key to authenticate a message m , by computing an *authentication tag* $MAC_k(m)$. Given a message m together with a previously-computed tag T , a party verifies the authenticity of the message m by re-computing $MAC_k(m)$ and comparing it to the tag T ; if equal, the message is valid, i.e., the tag must have been previously computed by the same party or another party, using the same secret key k .

In a typical use, one party, say Alice, sends a message m to a peer, say Bob, authenticating m by computing and attaching the tag $T = MAC_k(m)$. Bob confirms that $T = MAC_k(m)$, thereby validating that Alice sent the message, since he shares k only with Alice. See Fig. 4.1.

MAC schemes are related to *signature schemes*, an *asymmetric (public key)* authentication mechanism which we introduced in subsection 1.4.1. With signature schemes, each party, e.g., Alice, generates a *private signing key* $A.s$ and a corresponding *public verification key* $A.v$. In Definition 1.6, we define an *existentially unforgeable* signature scheme; intuitively, an adversary who is given the public key $A.v$, and can choose messages m_1, m_2, \dots and receive the corresponding signatures $\sigma_1 = \mathcal{S}.Sign_{A.s}(m_1), \sigma_2 = \mathcal{S}.Sign_{A.s}(m_2), \dots$, cannot find a different message $m' \notin \{m_1, m_2, \dots\}$ with a corresponding signature σ' such that $\mathcal{S}.Verify_{A.v}(m', \sigma') = \text{TRUE}$.

Note that we usually use the term *authenticator* to refer to the output $MAC_k(m)$ of the MAC function, i.e., if $\sigma = MAC_k(m)$, then σ is the *authenticator* of m using shared key k . Other terms for the authenticator are a *tag*, or a *signature*; we warn that this last term (‘signature’) may cause confusion between MAC schemes and signature schemes, and therefore, we recommend (and try) to avoid it.

Repudiation vs. deniability While both signature schemes and MAC schemes are used to authenticate messages, there is a critical difference: a valid MAC can be computed by *any* entity that knows the shared key. Consider the scenario in Figure 4.1; even after Carl successfully validates \hat{m} using $\hat{\sigma}$, he

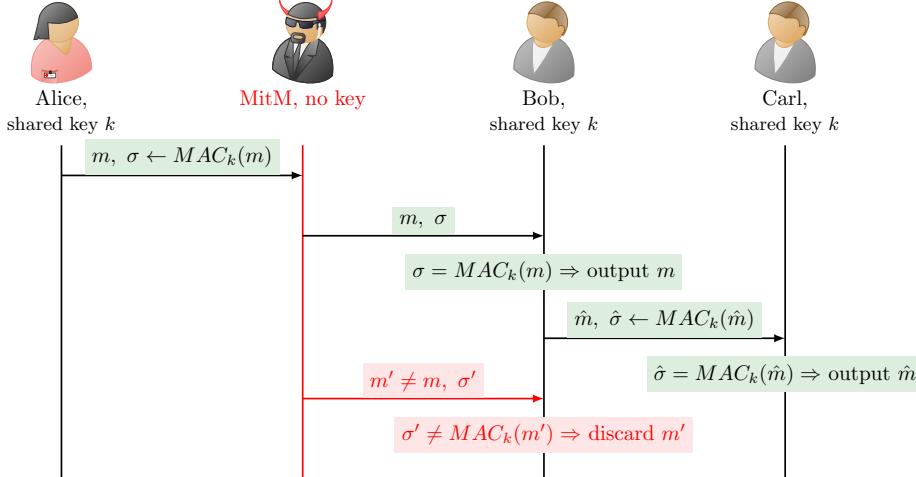


Figure 4.1: Using a *Message Authentication Code (MAC)* scheme, and a shared key k , to authenticate messages. The Man-in-the-Middle (MitM) adversary can observe message m and its authenticator $\sigma = MAC_k(m)$, but cannot *forgery* the MAC, i.e., generate a pair m', σ' such that $\sigma' = MAC_k(m')$, for $m' \neq m$. Note, however, that if a key k is shared among more than two entities, e.g., Alice, Bob and Carl, then each entity can authenticate messages using k ; e.g., when Carl receives \hat{m} , it cannot know if \hat{m} was sent by Alice or Bob (or even by Carl himself), except by using some indication within m , e.g., if m includes sender identification. Due to this property, we say that MAC schemes allow *repudiation*; for *non-repudiation*, use signatures instead of MAC.

cannot know if \hat{m} was sent by Alice or Bob. Of course, the sender identity may be indicated as part of the message m ; however, there is nothing preventing an entity knowing the shared key k , from putting a different identity in the message and computing the MAC. This is in contrast to signature schemes, where the private signing key $A.s$ must be used to produce a valid signature σ for a given message m , namely, knowing the public validation key $A.v$ does not allow forgery of a message as if it was signed using $A.s$. This property of signature schemes is often referred to as *non-repudiation*, as it prevents the sender of a (signed) message from repudiating (denying) having sent it. Note that in some situations, e.g., for a whistle-blower, we may have the opposite goal, i.e., of *preventing* the recipient from proving the identity of the sender to a third person; this goal is usually referred to as *deniability*.

To validate that a given tag T correctly validates a message m , i.e., $T = MAC_k(m)$, requires the ability to compute $MAC_k(\cdot)$, i.e., knowledge of the *shared* secret key k . However, this implies the ability to compute (valid) tags from any other message. This allows the entity that computed the tag to later deny having done it, since it could have been computed also by other entities. Therefore, MAC schemes do not ensure non-repudiation - and, exactly because

of that, allow *deniability*.

Namely, we should use a signature scheme when we need to ensure non-repudiation, i.e., to ensure that after validating a message with the key associated with Alice, we should be able to assume that Alice indeed sent the message. When we do not need non-repudiation, and especially when *deniability* is important, then we should use a MAC scheme.

4.3 Message Authentication Code (MAC): Definitions

A MAC scheme is a function F , with the following *unforgeability* property: an attacker, which does not know the key k and is not given $F_k(m)$ for any given message m , is unable to find the value of $F_k(m)$, with better chance than a random guess. The definition has a lot in common with the definition of signature schemes and their existential-unforgeability requirement, see subsection 1.4.1; in particular, we allow the adversary to obtain the MAC values for *any other* message. The definition follows. For concreteness, we will focus on MAC whose output is an l -bit binary string.

Definition 4.1 (MAC). *An l -bit Message Authentication Code (MAC) over domain D , is a function $F : \{0,1\}^* \times D \rightarrow \{0,1\}^l$, such that for all PPT algorithms \mathcal{A} , the advantage $\varepsilon_{F,\mathcal{A}}^{MAC}(n)$ is negligible in n , i.e., smaller than any positive polynomial for sufficiently large n (as $n \rightarrow \infty$), where:*

$$\varepsilon_{F,\mathcal{A}}^{MAC}(n) \equiv \Pr_{k \xleftarrow{\$} \{0,1\}^n} \left[(m, F_k(m)) \leftarrow \mathcal{A}^{F_k(\cdot|\text{except } m)}(1^n) \right] - \frac{1}{2^l} \quad (4.1)$$

Where the probability is taken over the random choice of an n bit key, $k \xleftarrow{\$} \{0,1\}^n$, as well as over the coin tosses of \mathcal{A} .

Oracle. The expression $\mathcal{A}^{F_k(\cdot|\text{except } m)}$ refers to the output of the adversary \mathcal{A} , where during its run, the adversary can give arbitrary inputs $x \neq m$ and receive the corresponding values of the function, $F_k(x)$. We say that the adversary \mathcal{A} has an *oracle* to the MAC function $F_K(\cdot)$ (excluding the message m). See Definition 1.3.

The advantage function $\varepsilon_{F,\mathcal{A}}^{MAC}(n)$ and key length n . The definition is for l -bit MAC, i.e., the output is always a binary string of length l . Hence, a random guess at the MAC of any input message m would be correct with probability 2^{-l} . Therefore, we defined the *advantage* $\varepsilon_{F,\mathcal{A}}^{MAC}(n)$ as the probability that the adversary finds a correct MAC value for a message m (not input to the oracle), minus the ‘base success probability’ of 2^{-l} . The function F is a (secure) MAC, if this advantage $\varepsilon_{F,\mathcal{A}}^{MAC}(n)$ is negligible.

The key length is denoted n , and is not bounded. The ‘advantage’ of the adversary over random guess, should be negligible in n , i.e., converge to zero as n grows. In practice, MAC functions are used with specific key length, which

is believed to be ‘long enough’ to foil attacks (by attackers with reasonable resources and time).

Output length - fixed (l) or as key length (n). In some other definitions of MAC schemes, the output length is also n , i.e., same as the key. In this case, the $\frac{1}{2^l}$ element becomes $\frac{1}{2^n}$, which is negligible in n , and hence can be ignored.

Input domain. Notice that the definition allows an arbitrary input domain D to the MAC function. The two most commonly used domains are $D = \{0, 1\}^*$, i.e., the set of all binary string (of unbounded length), and $D = \{0, 1\}^{l_{in}}$, i.e., the set of all binary strings of some fixed length l_{in} . Of course, l_{in} may also be the same as l . A MAC function whose input is the set of binary strings of fixed length, is called *FIL-MAC*, i.e., *Fixed Input Length MAC*. In contrast, a MAC function whose input is the set of all binary strings is called *VIL-MAC*, i.e., *Variable Input Length MAC*.

To ‘warm up’, let us show two examples of *insecure* MAC design. Our examples follow the definition, i.e., the attacker is allowed to ask for the MAC for some messages, and then has to come up with a *different* message and a correct MAC for that message. Notice that the definition does not require the ‘forged’ message to be ‘meaningful’; this means that it isn’t always trivial to *exploit* a vulnerable MAC. Following the ‘conservative design principle’ (Principle 3), the definition does not attempt to predict which forgeries will be meaningful, instead forbidding *any* forgery.

Our first example is a very simple FIL-MAC construction which we denote XOR^E ; the construction is defined for a given n -bit block cipher E . The XOR^E construction is defined for inputs which are exactly *two blocks*. Its output is a single block, which is the result of XORing the ‘encryption’ of each block. Namely, for given key k :

$$XOR_k^E(m) \equiv E_k(m[1 : n]) \oplus E_k(m[n + 1 : 2n])$$

The following example shows that XOR^E is not a secure MAC. We recommend you try to show it yourself before reading the solution.

Example 4.1. *To show that XOR^E is not a secure MAC, observe that:*

$$\begin{aligned} XOR_k^E(m) &= E_k(m[1 : n]) \oplus E_k(m[n + 1 : 2n]) \\ &= E_k(m[n + 1 : 2n]) \oplus E_k(m[1 : n]) \\ &= XOR_k^E(m[n + 1 : 2n] \oplus E_k(m[1 : n])) \\ &= XOR_k^E(\bar{m}) \text{ where } \bar{m} = m[n + 1 : 2n] \oplus E_k(m[1 : n]) \end{aligned}$$

Therefore, for every $2n$ -bits input message m , we have $XOR_k^E(m) = XOR_k^E(\bar{m})$, where \bar{m} is simply the message with the two blocks switched. This suffices to conclude that XOR^E does not satisfy the definition for secure MAC.

Let us present a specific adversary \mathcal{A} that ‘breaks’ XOR^E , i.e., shows it does not meet the definition of secure MAC. First, \mathcal{A} asks for the MAC of

$m_{01} = 0^n \parallel 1^n$ (a block of zeros followed by a block of 1's); we could have used almost any $2n$ -bit message, the choice of m_{01} is just for simplicity.

As per the definition, \mathcal{A} receives the MAC of m_{01} , i.e., $XOR_k^E(m_{01}) = E_k(0^n) \oplus E_k(1^n)$. Then \mathcal{A} returns the pair $m_{10}||XOR_k^E(m_{01})$, where $m_{10} = 1^n \parallel 0^n$. This is a successful forgery, since:

$$XOR_k^E(m_{10}) = E_k(1^n) \oplus E_k(0^n) = E_k(0^n) \oplus E_k(1^n) = XOR_k^E(m_{01})$$

Our second example is of a ‘hairy’ function f_k , defined ‘from scratch’, i.e., not using an underlying block cipher. Such ‘hairy’ designs may appear to be good candidates for MAC - but all too often, they are vulnerable. Showing the vulnerability can be tricky, which motivates (1) the use of a strong definition and (2) the use of standard, secure constructions from basic building blocks, following the ‘building blocks principle’ (Principle 8).

Example 4.2. Consider $f_k(m) = (k^3 \cdot m + k^2 \cdot m^2 + k \cdot m^3) \bmod p$, where p is a known number. Let us show, in a simple yet detailed way, that this hairy expression is not a secure MAC. Notice that our solution does not involve any attempt to find k !

The idea of the solution is simple. Recall the most basic properties of modular arithmetic (Section A.2). From these it follows that $f_k(m) = f_k(m + i \cdot p)$, for any integer i . In particular, $f_k(m) = f_k(m + p)$. If this isn’t clear, try to substitute some small integers for m , k and p ; and then read again Section A.2 to see why these equations hold.

This is the crux of the solution, but let us complete the details, by presenting an adversary \mathcal{A} which s.t. $\varepsilon_{F,\mathcal{A}}^{MAC}(n)$ is non-negligible; in fact, we’ll show that $\varepsilon_{F,\mathcal{A}}^{MAC}(n) = 1 - 2^{-l}$, for every n . Actually, the fact that we show advantage of (almost) 1 is quite typical of these exercises, although, of course, it suffices to show any non-negligible advantage.

The adversary \mathcal{A} is the following simple algorithm:

1. Let m' be some arbitrary value, e.g. p , or 1, or 0, or whatever you like.
2. Let $x \leftarrow F_k(m')$, i.e., call the oracle on m' .
3. Let $m = m' + p$.
4. Output (m, x) .

Let us explain why $\varepsilon_{F,\mathcal{A}}^{MAC}(n) = 1 - 2^{-l}$. Given oracle access to $f_k(\cdot)$ (for some random k), the adversary gave some input m' , and received $x \equiv f_k(m')$, i.e., in our case, $x = (k^3 \cdot m' + k^2 \cdot m'^2 + k \cdot m'^3) \bmod p$. Then the adversary outputs (m, x) , where $m = m' + p$. Obviously, $m \neq m'$, so the condition of the use of the oracle is satisfied; on the other hand, $x = f_k(m') = f_k(m + p) = f_k(m)$. Therefore, the expression is true for any k and we have: $\Pr_{k \in \{0,1\}^n} [(m, F_k(m)) \leftarrow \mathcal{A}^{F_k(\cdot \mid \text{except } m)}(1^n)] = 1$, proving that $\varepsilon_{F,\mathcal{A}}^{MAC}(n) = 1 - 2^{-l}$.

4.4 Applying MAC Schemes

A MAC function is a simple cryptographic mechanism, which is quite easy to use; however, it should be applied correctly - with an understanding of its properties and without expecting it to provide other properties. We now discuss a few aspects of the usage of MAC schemes, and give a few examples of common mistakes.

Confidentiality A MAC function is a great tool to ensure integrity and authenticity; however, MAC *may not ensure confidentiality*. Namely, $MAC_k(m)$ may expose information about the message m . This is sometimes overlooked by system designers; for example, early versions of the SSH protocol used the so-called ‘Encrypt and Authenticate’ method, where to protect message m , the system sent $E_k(m) + MAC_k(m)$; one problem with this design is that $MAC_k(m)$ may expose information about m .

Notice that while obviously confidentiality is not a goal of MAC schemes, one may hope that it is derived from the authentication property. To refute such false hopes, it is best to construct a counterexample - a very useful technique to prove that claims about cryptographic schemes are incorrect. The counter-examples are often very simple - and often involve ‘stupid’ or ‘strange’ designs, which are specially designed to meet the requirements of the cryptographic definitions - while demonstrating the falseness of the false assumptions. Here is an example showing that MAC schemes may expose the message.

Example 4.3 (MAC does not ensure confidentiality.). *To show that MAC may not ensure confidentiality, we construct such a Non-confidential MAC function F^{NcM} (where NcM stands for ‘Non-confidential MAC’). Our construction uses an arbitrary secure MAC scheme F (which may or may not ensure confidentiality). Specifically:*

$$F_k^{NcM}(m) = F_k(m) + LSb(m)$$

where $LSb(m)$ is the least-significant bit of m . Surely, F^{NcM} does not ensure confidentiality, since it exposes a bit of the message (we could have obviously exposed more bits - even all bits!).

On the other hand, we now show that F^{NcM} is a secure MAC. Assume, to the contrary, that there is some adversary \mathcal{A}^{NcM} that succeeds (with significant probability) against F^{NcM} . We use \mathcal{A}^{NcM} to construct an attacker \mathcal{A} that succeeds with the same probability against F . Attacker \mathcal{A} works as follows:

1. When \mathcal{A}^{NcM} makes a query q to F^{NcM} , then \mathcal{A} makes the same query to F , receiving $F_k(q)$; it then returns $F_k^{NcM}(q) = F_k(q) + LSb(q)$, as expected by \mathcal{A}^{NcM} .
2. When \mathcal{A}^{NcM} outputs its guess m, T , where T is its guess x for $F_k^{NcM}(m) = F_k(m) + LSb(m)$, and m was not used in any of \mathcal{A}^{NcM} ’s queries, then \mathcal{A} outputs x except for its least-significant bit; namely, if $x = F_k^{NcM}(m) = F_k(m) + LSb(m)$, then \mathcal{A} outputs $F_k^{NcM}(m) = F_k(m)$.

It follows that F^{NcM} is a secure MAC if and only if F is a secure MAC. \square

We show, later on (subsection 4.5.1), that every PRF is a MAC. The following exercise shows that the reverse is not true: a MAC is not necessarily a PRF. This exercise is similar to the example above.

Exercise 4.2 (Non-PRF MAC). *Show that a MAC function is not necessarily a pseudorandom function (PRF).*

Solution outline: Let F be an arbitrary secure MAC scheme that outputs n -bit tags. Construct a MAC scheme F' , which outputs $2n$ -bit tags, as follows.

$$F'_k(m) = F_k(m) \# 0^n$$

Clearly, F' is not a PRF, because \mathcal{A} has a significant chance of distinguishing between an output of F' and a random $2n$ -bit string (since the second half of the output of F' is all zeros). Yet, you can show that F' is a secure MAC if and only if F is a secure MAC, using a similar method to the one in Example 4.3. Therefore, a MAC function is not necessarily a PRF. \square

Key separation Another problem with the SSH ‘Encrypt and Authenticate’ design, $E_k(m) \# MAC_k(m)$, is the fact that the same key is used for both encryption and MAC. This can cause further vulnerability; an example is shown in the following simple exercise.

Exercise 4.3 (Separate keys for separate functions). *Show that the use of the same key for encryption and MAC in $E_k(m) \# MAC_k(m)$ can allow an attacker to succeed in forgery of messages - in addition to the potential loss of confidentiality shown above - even when E and MAC are secure (encryption and MAC, respectively).*

Solution outline: Let E', MAC' be secure encryption and MAC functions, respectively. Define $E_{k_E, k_M}(m) = E'_{k_E}(m) \# k_M$ and $MAC_{k_E, k_M}(m) = k_E \# MAC'_{k_M}(m)$. Obviously, the use of $E_{k_E, k_M}(m) \# MAC_{k_E, k_M}(m)$ exposes both keys and is therefore insecure. However, using the method of Example 4.3, you can show that E, MAC are also secure encryption and MAC functions, respectively. See also Exercise 4.18. \square

Another motivation to separate between the keys used by a given cryptographic function/scheme, is to reduce the quantity of plaintext available to the cryptanalyst, and especially the amount of known and chosen plaintext. These considerations result in the *principle of key separation*.

Principle 10 (Key Separation). *Use separate, independently-pseudorandom keys for: (1) each different cryptographic scheme/function, (2) different types and/or different sources of plaintext, (3) different periods, and (4) different versions of the protocol or scheme.*

Freshness, replay-prevention and sender authentication A valid MAC received with a message shows that the message was properly authenticated by an entity holding the secret key, which we refer to as *message authentication*. Message authentication is a useful property, and can facilitate additional important properties. Let us discuss three such important properties, which can be facilitated using message authentication, and are even sometimes (incorrectly) assumed to be implied directly by message authentication: *sender authentication*, *freshness* and *no-replay*.

Sender authentication is the ability to identify the identity of the party originating the message. We commented above that MAC does not ensure *sender authentication*, unless the design ensures that only the specific sender will compute the MAC using the specific key over the given message. A simple way to ensure this is by including the sender identity as part of the payload being signed. Another way to ensure this is for each sender to use its own authentication key. Of course, both methods do not prevent one entity holding a shared key, from impersonating as another entity using the same key; to prevent this, use signatures.

Freshness is the ability to confirm that a message was sent ‘recently’; a related property, *replay-prevention*, ensures that the message was not already handled previously. We can use MAC to ensure these properties, by including in the authenticated data appropriate fields, such as a timestamp, a counter or a random number (‘nonce’) selected by the party validating freshness. Each of these options has its corresponding drawback: the need for synchronized clocks, the need to keep a state, or the need for the sender to receive the nonce from the recipient (additional interaction).

4.5 Constructing MAC from a Block Cipher

In this section we discuss constructions of a MAC function from a *block cipher*. More precisely, the constructions are of a MAC from a *pseudorandom function (PRF)*, mainly the *CBC-MAC* construction. Since a PRF is often not included in cryptographic libraries, it may be tempting to use instead a *block cipher*, which is part of most cryptographic libraries; recall that a block cipher is modeled by a *Pseudo-Random Permutation*, *PRP*, rather than by a PRF. The PRP/PRF switching lemma (Lemma 2.2) shows that we could simply use a block-cipher instead of the PRF, since a block-cipher (PRP) is indistinguishable from a PRF; however, recall this is not advisable, since the use of block cipher instead of PRF involves loss in security. Instead, use one of the efficient, simple constructions of PRF from a block cipher, which avoid the loss of security, e.g., [33, 154].

The section contains three subsections, In the first, we observe that given a PRF, we can actually use it directly as a MAC, i.e., *every PRF is also a MAC*. There is a caveat: the input domain of the MAC is the same as that of the PRF, which, in turn, is the same as of the underlying block cipher (if the PRF is implemented from a block cipher as explained above). Namely, if we use n -bit blocks, i.e. the domain of the block-cipher (and PRF) is $\{0, 1\}^n$, then the

MAC function also applies (only) to n -bit messages. This is not satisfactory, since typical messages are longer.

The second subsection presents the *CBC-MAC* construction, which constructs a $l \cdot n$ -bit PRF from an n -bit PRF, for a given constant number of blocks l . This allows efficient and secure use of n -bit-input PRF (or block cipher), to encrypt longer, $l \cdot n$ -bits messages.

Finally, in the third subsection we discuss extensions that allow a MAC for messages of *arbitrary length*.

In the following section, we discuss other construction of MAC schemes, which are not based on the use of a secure block cipher, most notably, the *HMAC* construction of a MAC scheme from a cryptographic hash function.

4.5.1 Every PRF is a MAC

In this subsection, we take the first step toward the CBC-MAC construction. This step is the observation that *every PRF whose range is $\{0, 1\}^l$, is also an l -bit MAC*, with the same input and output domains. This is formalized in the following lemma, which we call the *PRF-is-MAC lemma*.

Lemma 4.1 (A PRF is a MAC). *Let F be a PRF from input domain D to the range $\{0, 1\}^l$. Then F is also an l -bit MAC, with input domain D and output domain $\{0, 1\}^l$.*

Proof: Assume that F is not a MAC (for same domain D and range $\{0, 1\}^l$). Namely, assume that there exists some adversary \mathcal{A}_{MAC} s.t. $\varepsilon_{\mathcal{A}_{MAC}, F}^{MAC}(n)$ is non-negligible in n (as defined in Equation 4.1). We use \mathcal{A}_{MAC} to construct another adversary, \mathcal{A}_{PRF} , s.t. $\varepsilon_{\mathcal{A}_{PRF}, F}^{PRF}(n)$ is non-negligible in n (as defined in Equation 2.29); this shows that F is (also) not a PRF, which proves the claim.

Let us now define \mathcal{A}_{PRF} . First, recall that in Equation 2.29, adversary A_{PRF} is given an oracle either to a random function $f \xleftarrow{\$} \{D \rightarrow \{0, 1\}^l\}$, or to the pseudorandom function $F_k : D \rightarrow \{0, 1\}^l$ for some random n -bit key $k \xleftarrow{\$} \{0, 1\}^n$. Adversary \mathcal{A}_{PRF} runs \mathcal{A}_{MAC} , letting it use the same oracle. Namely, whenever \mathcal{A}_{MAC} asks its oracle with input $x \in D$, adversary \mathcal{A}_{PRF} calls its oracle with the same input x ; and when it receives a result ξ , it returns that result to \mathcal{A}_{MAC} .

When \mathcal{A}_{MAC} terminates, it should return some pair, which we denote by (m, σ) . Upon receiving (m, σ) , adversary \mathcal{A}_{PRF} provides m as input to its oracle; denote the output by σ' . If $\sigma \neq \sigma'$, then \mathcal{A}_{PRF} returns ‘Rand’; otherwise, i.e., if $\sigma = \sigma'$, then \mathcal{A}_{PRF} returns ‘Pseudo’. Essentially, \mathcal{A}_{PRF} outputs ‘Rand’ (i.e., guess it was given a random function), when A_{MAC} fails was able to predict correctly the output of the oracle for the input m .

Let us consider what happens if \mathcal{A}_{PRF} is given an oracle to a random function $f \xleftarrow{\$} \{D \rightarrow R\}$. In this case, when running \mathcal{A}_{MAC} , the values returned from the oracle were for that random function f ; clearly, \mathcal{A}_{MAC} cannot be expected to perform as well as when given an oracle to the function $F_k(\cdot)$. In fact, \mathcal{A}_{MAC} has to return a pair (m, σ) , without giving input m to the oracle.

But if the oracle is to a random function f , then $f(m)$, is chosen independently of $f(x)$ for any other input $x \neq m$; learning other outputs cannot help you guess the output when the input is m ! Hence, the probability of a match is (only) 2^{-l} - the probability of a random match between two random l bit strings. Namely, $\Pr_{f \leftarrow \{D \rightarrow R\}} [\mathcal{A}^f(1^n) = \text{'Rand'}] = 2^{-l}$.

Now consider what happens if \mathcal{A}_{PRF} is given an oracle to a pseudorandom function $F_k(\cdot)$. The claim is that F is also a MAC, but we assumed, to the contrary, that it is *not*; so A_{MAC} is able to return a pair $(m, F_k(m))$ - with probability *significantly larger* than 2^{-l} . In these cases, \mathcal{A}_{PRF} will return ‘PR’. Namely, $\Pr_{k \leftarrow \{0,1\}^n} [\mathcal{A}^{F_k}(1^n) = \text{'Rand'}] = 2^{-l} + p(n)$, where $p(n)$ is a *significant* (not negligible) function.

It follows that $\varepsilon_{\mathcal{A}, F}^{PRF}(n)$ is not negligible, and hence, F is *not* a PRF. \square

4.5.2 CBC-MAC: ln -bit MAC (and PRF) from n -bit PRF

Lemma 4.1 shows that every n -bit PRF is also an n -bit MAC. But how can we deal with longer messages? In this subsection, we present the *CBC-MAC* construction, which produces an $l \cdot n$ -bit PRF, using a given n -bit PRF. Since every PRF is a MAC, this gives also an $l \cdot n$ -bit MAC. The CBC-MAC construction is a standard from 1989 [178], i.e., prior to the PRF-is-MAC lemma (from [31]), which is why it refers to construction of MAC (from block cipher) and not to construction of $l \cdot n$ -bit PRF from n -bit PRF.

Before we present the CBC-MAC construction, let us discuss some *insecure* constructions. First, consider performing MAC to each block independently, similar to the ECB-mode (Section 2.8). One drawback is that this would result in a long MAC. An even worse drawback is that this is insecure; an attacker may obtain a MAC for a different message, which contains re-ordered and/or duplicated blocks.

Next, consider adding a counter to the input, to which we refer as *CTR-MAC*. This prevents the trivial attack - but not simple variants, as shown in the following exercise. For simplicity, the exercise is given for $l = 2$. Of course, this design also has the disadvantage of a longer output tag.

Exercise 4.4 (CTR-MAC is insecure). *Let E be a secure $(n+1)$ -bit block cipher, and define the following $2n$ -bit domain function: $F_k(m_0 + m_1) = E_k(0 \# m_0) \# E_k(1 \# m_1)$ (CTR-MAC). Present a counterexample showing that F is not a secure $2n$ -bit MAC.*

Finally, we present the *CBC-MAC construction*, also known as the *CBC-MAC mode*. This is a widely used, standard construction of an $(l \cdot n)$ -bit MAC from an n -bit block cipher. The CBC-MAC mode, illustrated in Fig. 4.2, is a variant of the CBC mode used for encryption, see Section 2.8. Given a block-cipher E , we define $CBC-MAC^E$ as in Eq. 4.2, for an l -block input message (i.e., of length $l \cdot n$ bits), $m = m_1 \# m_2 \# \dots \# m_l$:

$$CBC-MAC_k^E(m) = \{c_0 \leftarrow 0^n; (i = 1 \dots l) c_i = E_k(m_i \oplus c_{i-1}); \text{output } c_l\} \quad (4.2)$$

See Fig. 4.2.

When E is obvious we may simply write $CBC - MAC_k(\cdot)$.

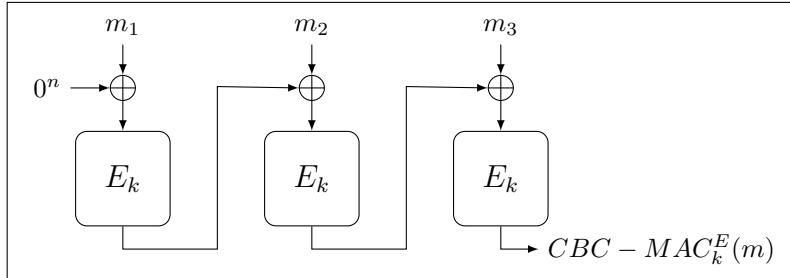


Figure 4.2: CBC-MAC: construction of $l \cdot n$ -bit PRF (and MAC), from n -bit PRF.

CBC-MAC is the most widely used MAC construction from block ciphers. Other constructions of secure MAC from PRFs and block ciphers, including more efficient constructions, e.g., avoiding the need to know the input length in advance (CMAC [114]) or allowing parallel computation and verification (e.g., XOR-MAC [30]). However, we focus on CBC-MAC, which is not only the most widely used, but also one of the most simple constructions of MAC from a block cipher.

We next present Lemma 4.2 which shows that CBC-MAC constructs a secure PRF (and hence also MAC), provided that the underlying function E is a PRF.

Lemma 4.2. *If E is an n -bit PRF, then $CBC - MAC_k^E(\cdot)$ is a secure $n \cdot l$ -bit PRF and MAC, for any constant integer $l > 0$.*

Proof: see in [31].

CBC-MAC does not support input of arbitrary length. The CBC-MAC construction is defined for input which is an integral number of blocks, i.e., $n \cdot l$ bits. How can we extend it so it does support input of arbitrary length, i.e., a *variable input length (VIL)* PRF (and MAC) - defined for input domain $\{0, 1\}^*$?

One obvious problem is that an arbitrary binary string, may not even consist of an integral number of blocks, while CBC-MAC is defined only for inputs which are of length $n \cdot l$, i.e., integral number of blocks. However, let us ignore that problem for now, and focus on the *complete-blocks input length (CBIL)* domain, i.e., inputs whose length is an *integer number of blocks*. Let us first precisely define the CBIL domain.

$$CBIL \equiv \{m \in \{0, 1\}^{n \cdot l} \mid l \in \mathbb{Z}^+\} \quad (4.3)$$

In the next exercise we show that CBC-MAC is not a PRF, or a MAC, for the CBIL domain, and, hence, surely not a VIL MAC/PRF.

Exercise 4.5 (CBC-MAC is not a VIL MAC). *Show that CBC-MAC is not a MAC or PRF for the domain CBIL (Equation 4.3), and hence is definitely not a VIL MAC/PRF (for the domain $\{0, 1\}^*$).*

Solution: Let $f_k(\cdot) = \text{CBC} - \text{MAC}_k^E(\cdot)$ be the CBC-MAC using an underlying n -bit block cipher E_k . Namely, for a single-block message $a \in \{0, 1\}^n$, we have $f_k(a) = E_k(a)$; and for a two block message $a \# b$, where $a, b \in \{0, 1\}^n$, we have $f_k(a \# b) = E_k(b \oplus E_k(a))$.

We present a simple adversary A^{f_k} , with oracle access to f_k , i.e., A is able to make arbitrary query $x \in \{0, 1\}^*$ to f_k and receive the result $f_k(x)$. Let X denote all the queries made by A during its run. We show that A^{f_k} generates a pair $x, f_k(x)$, where $x \notin X$, which shows that f_k (i.e., CBC-MAC) is not a MAC for domain CBIL (and hence also not a $\{0, 1\}^*$ -MAC, i.e., VIL MAC).

Specifically, the adversary A first makes an arbitrary single-block query, for arbitrary $a \in \{0, 1\}^n$. Let c denote the result, i.e., $c = f_k(a) = E_k(a)$. Then, A computes $b = a \oplus c$ and outputs the pair of message $a \# b$ and tag c .

Note that $c = f_k(a \# b)$, since $f_k(a \# b) = E_k(b \oplus E_k(a)) = E_k((a \oplus c) \oplus c) = E_k(a) = c$. Namely, c is indeed the correct tag for $a \# b$. Obviously, A did not make a query to receive $f_k(a \# b)$. Hence, A succeeds in MAC game against CBC-MAC. \square

However, as we next explain, by merely *prepend the length to the input*, we can create a VIL MAC from the CBC-MAC.

4.5.3 Constructing Secure VIL MAC from PRF

Lemma 4.2 shows that CBC-MAC is a secure *ln*-bit FIL PRF (and MAC); however, Exercise 4.5 shows that it is *not* a VIL MAC (and hence surely not VIL PRF). The crux of the example was that we used the CBC-MAC of a one-block string, and presented it as the MAC of a 2-block string. This motivates a minor change to the construction, where we *prepend the block-encoded length $L(m)$* of the input m to the input before applying CBC-MAC. We define $L(m)$ as an n -bit binary string (i.e., a block), whose binary value is the length $|m|$ of the input m . Lemma 4.3 shows that this construction is indeed a secure VIL MAC. We refer to this variant as *length-prepending CBC-MAC*.

Lemma 4.3 (Length-prepending CBC-MAC is a VIL PRF.). *Let $f_k(m) = \text{CBC} - \text{MAC}_k^E(L(m) \# m)$, where $L(m)$ is the block-encoded length of m (as defined above). Then $f_k(\cdot)$ is a PRF over the set of all binary strings (and MAC).*

Proof: See [31]. \square

Note that the block-encoded length $L(m)$, can only support message up to the maximal length encoded by n bits - i.e., $|m| < 2^n$. In practice, this isn't an issue - and it is not difficult to extend the construction to avoid this limitation, if you really want to. It is hard to imagine a practical scenario in which you will *have* to do this, however.

4.6 Other MAC Constructions

In the previous section, we presented constructions of MAC from PRFs and block ciphers. In the following subsections, we discuss other approaches for constructing a MAC function, including: (1) design a MAC ‘from scratch’, i.e., without provable reduction to the security of some other cryptographic scheme (subsection 4.6.1), (2) combine multiple candidate MAC functions (*robust combiner*, subsection 4.6.2), and, finally, (3) construct a MAC/PRF from a cryptographic hash function (subsection 4.6.3). This last approach, constructing a MAC/PRF from a hash function, is the most widely-use method to implement a MAC function, usually using the *HMAC* construction [25, 26].

4.6.1 MAC design ‘from scratch’

This approach attempts to design a candidate MAC function without requiring a reduction to the security of some cryptographic scheme; typically, the design simply does not involve any other, known cryptographic function. Instead, we may use some problems which are considered computationally-hard. The security of such design is based on the failure of significant cryptanalysis efforts against the MAC function.

This used to be the main method of design of new cryptographic mechanisms. However, following the cryptographic building block principle (principle 8), MAC functions are rarely designed ‘from scratch’. Let us give an example of one example: a (failed) attempt to construct MAC from EDC, and the resulting vulnerabilities.

Two (failed) attempts to construct MAC from EDC Let us consider a specific design, which, intuitively, may look promising: constructing a MAC from a (good) *Error Detection Code* (*EDC*). Error Detection Codes are designed to ensure integrity, i.e., to detect corruptions in data; however, they are designed to detect *random errors*, and may fail to detect *intentional modifications*. We have seen already, in Section 2.10, that the *WEP* protocol failed to ensure integrity against attack, in spite of its use of the CRC-32 *Cyclic Redundancy Check* (*CRC*) error-detecting code before encryption. Let us consider two other simple constructions of a MAC from an EDC, which do not involve encryption: $MAC_k(m) = EDC(k + m)$ and $MAC'_k(m) = EDC(m + k)$. We next show that these construction are insecure, when using CRC as the error-detection code (EDC).

Exercise 4.6 (Insecure CRC-based MACs). *Show that both (a) $CRC-MAC_k(m) = CRC(k + m)$ and (2) $CRC-MAC'_k(m) = CRC(m + k)$ are insecure.*

Solution: We only solve (a) and leave (b) as an (easy) exercise to the reader. In fact, we show how the attacker that receives only the MAC $CRC-MAC_k(m) = CRC(k + m)$ of any known message m , can compute the MAC for any other message $m' \neq m$ of the same length, i.e., compute $CRC-MAC_k(m') = CRC(k + m')$.

Recall that the CRC function is linear, namely, for any two strings of the same length, $|x| = |x'|$, holds: $\text{CRC}(x \oplus x') = \text{CRC}(x) \oplus \text{CRC}(x')$ (Equation 2.63). Hence:

$$\begin{aligned}
 \text{CRC-MAC}_k(m') &= \text{CRC}(k \# m') \\
 &= \text{CRC}\left((0^{|k|} \# m' \oplus m) \oplus (k \# m)\right) \\
 &= \text{CRC}(0^{|k|} \# m' \oplus m) \oplus \text{CRC}(k \# m) \\
 &= \text{CRC-MAC}_{0^{|k|}}(m' \oplus m) \oplus \text{CRC-MAC}_k(m)
 \end{aligned} \tag{4.4}$$

The adversary computes $|k| = |\text{CRC-MAC}_k(m')| - |\text{CRC}(m')|$, and can therefore compute $\text{CRC-MAC}_{0^{|k|}}(m' \oplus m)$. By plugging this into Equation 4.4, the adversary finds $\text{CRC-MAC}_k(m')$. \square

We conclude that *CRC-MAC*s are indeed insecure. The same seems to hold for other ECD-based MACs.

4.6.2 Robust combiners for MAC

A *robust combiner* for MAC combines two (or more) candidate MAC functions to create a new composite function, which is proven secure provided that one (or a sufficient number) of the underlying functions is secure. There is actually a very simple robust combiner for MAC schemes: *concatenation* (denoted $\#$). In the following exercise we show that concatenation is a robust combiner for MAC functions.

Exercise 4.7. Show that concatenation is a robust combiner for MAC functions.

Solution (from [160]): Let F', F'' be two candidate MAC schemes, and define $F_{k',k''}(m) = F'_{k'}(m) \# F''_{k''}(m)$. We should show that it suffices that either F' or F'' is a secure MAC, for F to be a secure MAC scheme as well. Without loss of generality, assume F' is secure; and assume, to the contrary, that F is not a secure MAC. Namely, assume an attacker $\mathcal{A}^{F_{k',k''}(\mu)|\mu \neq m}$ that can output a pair $m, F_{k',k''}(m)$, given access to an oracle that computes $F_{k',k''}$ on any value except m . We use \mathcal{A} to construct an adversary \mathcal{A}' which succeeds against F' .

Adversary \mathcal{A}' operates by running \mathcal{A} , as well as selecting a key k'' and running $F''_{k''}(\cdot)$; this is needed to allow \mathcal{A}' to provide the oracle service to $\mathcal{A}^{F_{k',k''}(\mu)|\mu \neq m}$, computing $F_{k',k''}(\mu)$ for any given input μ . Whenever \mathcal{A} makes a query q , then \mathcal{A}' makes the same query to the $F'_{k'}(\cdot)$ oracle, to receive $F'_{k'}(q)$. Then, \mathcal{A}' computes by itself $F''_{k''}(q)$, and combines it with $F'_{k'}(q)$ to produce the required response $(F'_{k'}(q), F''_{k''}(q))$.

When \mathcal{A} finally returns the pair $(m, F_{k',k''}(m)) = (m, F'_{k'}(m) \# F''_{k''}(m))$, then \mathcal{A}' simply returns the pair $(m, F'_{k'}(m))$, i.e., omitting the second part of the MAC that \mathcal{A} returned. \square

However, concatenation is a rather inefficient construction for robust combiner of MAC schemes, since it results in duplication of the length of the output. The following exercise shows that exclusive-or is also a robust combiner for

MAC - and since the output length is the same as of the component MAC schemes, it is efficient.

Exercise 4.8. Show that exclusive-or is a robust combiner for MAC functions. Namely, that $MAC_{(k', k'')}(x) = MAC'_{k'}(x) \oplus MAC''_{k''}(x)$ is a secure MAC, if one or both of $\{MAC', MAC''\}$ is a secure MAC.

Guidance: Similar to the solution of Ex. 4.7. □

4.6.3 HMAC and other constructions of a MAC from a Hash function

Finally, we consider constructions of MAC functions from cryptographic hash functions. Cryptographic *hash functions*, like block ciphers, are defined in multiple standards, therefore their use to construct MAC (and other schemes) follows the *cryptographic building blocks principle* (subsection 2.7.4). Furthermore, since both MAC and hash functions are defined for arbitrary (variable) input length (*VIL*), the constructions of MAC from hash functions are simpler than the constructions from block ciphers (subsection 4.5.2). Furthermore, some cryptographic hash functions are extremely efficient, and this efficiency can be mostly inherited by HMAC. For example, the Blake2b [13] cryptographic hash function achieves speeds of over 10^9 bytes/second, with a relatively-weak CPU (Intel I5-6600 with 3310MHz clock).

In fact, the use of hash functions to construct a MAC is so common, that many people use the term ‘keyed hash’ to refer to the resulting MAC function. The meaning is that the hash function uses a *secret* key k . This *differs from our use of the term ‘keyed hash function’*, as in subsection 3.2.3, which is also the usage in most works in cryptography, where the key k is *not secret* (i.e., the key k is known to the adversary).

An additional problem with the term ‘keyed hash’ for the use of hash with a secret key to construct MAC, is that it may be interpreted to imply that it is safe to use a *keyed CRHF* as a MAC, simply by keeping its key secret instead of publishing it. It *may* be possible, for the same keyed function h to be a keyed CRHF (given a public key) and a MAC (given a secret key); however, it is also possible for h to be a keyed CRHF yet not to be a MAC (given a secret key), as we show in the next exercise. See also Exercise 4.22 and Exercise 4.23.

Exercise 4.9. Let $h_k(m)$ be a keyed CRHF. Show a keyed hash function $h'_k(m)$ which (1) is a CRHF but (2) is not a secure MAC.

Solution: Let $h'_k(m) = k||h_k(m)$. Clearly h' exposes its key, so it cannot be a secure MAC. However, h' is still a CRHF, since any collision of h' is also a collision for h . □

We see that a function may be a keyed CRHF but not a secure MAC; can we, instead, *construct* a MAC from a cryptographic hash function? Ideally, we would want to construct the MAC from a *keyless* cryptographic hash function, since existing standard cryptographic hash functions are keyless (subsection 3.1.4).

In the remainder of this subsection, we discuss four such constructions, whose goal is to create a MAC from keyless hash functions. We begin with three designs studied by Tsudik [304], and then describe HMAC [24], a more recent construction which is now widely deployed and defined as an IETF standard [26].

Tsudik's constructions of MAC from hash: prepend key, append key and message-in-the-middle. Several heuristic proposals for the construction of a MAC from a cryptographic hash function were made, mostly constructing the MAC from a keyless hash function. Three of the most well known heuristics were presented and compared by Tsudik [304]. Given keyless hash function h , key k and message m , these are:

$$\text{Prepend Key: } KM_k^h(m) = h(k \# m)$$

$$\text{Append Key: } MK_k^h(m) = h(m \# k)$$

$$\text{Message-in-the-Middle: } KMK_k^h(m) = h(k \# m \# k)$$

An obvious question is whether these schemes are secure - assuming that the cryptographic hash function h satisfies some assumption. Let us first observe that *all* three constructions are *secure under the ROM* (Section 3.6).

Exercise 4.10. *Prove that (a) KM^h , (b) MK^h and (c) KMK^h are secure under the Random Oracle Methodology (ROM).*

Proof sketch: assume an adversary outputs m, σ for a message m which it did not give as input to the ‘oracle’ for h . Then the output of the corresponding h function, was never computed yet, i.e., it is still random. For example, for $KM_k^h(m) = h(k \# m)$, the value of $h(k \# m)$, for this m , was not computed yet. In fact, we need to pick it only to check the adversary’s guess σ ; at that point, we choose it randomly from the set $\{0, 1\}^n$. The probability that our choice will be the same as σ is only 2^{-n} , i.e., negligible. Hence, $KM_k^h(m) = h(k \# m)$ is secure under the ROM. This shows (a); essentially the same argument holds for (b) MK^h and (c) KMK^h . \square

To avoid the possible impression that every construction is secure under the ROM, let us give an example of construction which is *insecure even under the ROM*. Specifically, consider $KMKM_k^h(m) = h(k \# m) + h(k \# m \oplus 1^{|m|})$. Namely, $KMKM^h$ was made ‘more complex’ - maybe with the futile hope that this will make it more secure - by concatenating *two* hash values, one of $k \# m$ and the other of $k \# m \oplus 1^{|m|}$. Note that $m \oplus 1^{|m|}$ is just a weird way for writing the negation of m .

Example 4.4. *Show that $KMKM^h$ is insecure, (even) under the ROM.*

Solution: Adversary asks to receive $KMKM_k^h$ for the message $m = 0^l$ (for any length l); let the value returned by denoted $\sigma_L \# \sigma_R$, where $|\sigma_L| = |\sigma_R| = n$. Then the adversary returns the ‘guess’ $(1^l, \sigma_R \# \sigma_L)$. Verify that this is the correct pair. \square

We recommend to readers to follow carefully the arguments in Exercise 4.10 and find out why they do not hold for $KMKM_k^h(m)$. It is not trivial - and may help understanding these important concepts.

We next observe that these three constructions, which *are* secure under the ROM, can be *insecure* using a hash function which satisfies standard requirements such as collision-resistance and preimage-resistance (one-way function), as in the following exercise. This illustrates the fact that security under the ROM does not imply security under standard assumptions.

Exercise 4.11. *Present a keyless hash function h such that:*

1. *h is a CRHF, yet (a) KM^h , (b) MK^h , (c) KMK^h is not a secure MAC.*
2. *h is a SPR, yet (a) KM^h , (b) MK^h , (c) KMK^h is not a secure MAC.*
3. *h is a OWF-hash, yet (a) KM^h , (b) MK^h , (c) KMK^h is not a secure MAC.*
4. *h is a BRE, yet (a) KM^h , (b) MK^h , (c) KMK^h is not a secure MAC.*
5. *h is CRHF, OWF and BRE, yet (a) KM^h , (b) MK^h , (c) KMK^h is not a secure MAC.*

The examples may assume a hash function h' which has the corresponding property (CRHF, SPR, OWF, BRE or their combination).

Partial solution: Let h' be a hash function h' which is CRHF, SPR and OWF. We define $h(x)$ to return the n most significant bits of x if $|x| = 2n$ and the n least significant bits are all zero, and to return $h'(x)$ otherwise. We leave it to the reader to prove that h' is a secure CRHF, SPR and OWF, yet KM^h is not a secure MAC. Changing this construction to also cover BRE is not very difficult, as is modifying the constructions to show corresponding results for MK^h and KMK^h . \square

While the examples in the solutions to Exercise 4.11 would be ‘artificial’ and irrelevant to any ‘real’ candidate hash function, some weaknesses of these constructions can apply to realistic hash functions. In particular, many hash functions have the following *extend property*: given $h(x)$, one can compute $h(x + y)$, even *without* knowing anything about x . This property hold for any hash function using the (widely-used) Merkle-Damgård construction, which is used by many hash functions, including the MD5 and SHA-1 standards; see discussion in Section 3.8. In the following exercise we observe, after Tsudik, that if h has the *extend property* then KM^h is not a secure MAC.

Exercise 4.12. *Show that the KM^h is insecure, for any hash function h that has the extend property.*

Hint: Tsudik has shown this in [304]. \square

HMAC. HMAC [24, 26] is the most widely-used construction of a MAC from a keyless hash function. HMAC is defined as:

$$HMAC_k(m) = h(k \oplus OPAD \# h(k \oplus IPAD \# m)) \quad (4.5)$$

Where OPAD, IPAD are fixed constant strings.

It is not difficult to see that HMAC is secure under the ROM (Exercise 3.29). However, while the ROM is useful, and security under this model indicates that some attacks are infeasible, it would surely be much better, if we could show that HMAC is secure under some ‘reasonable’ cryptographic assumption. In fact, this was done, in [24]. It would have been great if the assumption was one of the standard hash-function assumptions, e.g., collision resistance; however, the assumption in [24], while arguably reasonable, is somewhat more complex than these standard hash function assumptions, and we will not discuss these details here.

Note that HMAC is *insecure* when using some collision-resistant hash function, i.e., collision-resistance is not a sufficient requirement from the hash function. You will show this in Exercise 3.30, where you should construct a CRHF $h(\cdot)$, for which *HMAC is not a secure MAC*. To ensure that h is a CRHF, the construction uses a given CRHF, $h'(\cdot)$.

Due to the importance and wide use of HMAC, confidence in its security grew over the years, with several additional results establishing its security under ‘even more reasonable’ assumptions (compared to [24]). The confidence in the security of HMAC also grew due to the fact that such important standard has not been ‘broken’ by cryptanalysis during this time. In fact, over time, HMAC is also often used for additional goals, such as a *pseudorandom function (PRF)* and as a *Key Derivation Function (KDF)*, which is essentially a keyed variant of a *randomness extraction* hash function; see discussion in Section 3.5.

4.7 Combining Authentication, Encryption and Other Functions

Message authentication combines authentication (sender identification) and integrity (detection of modification). However, when transmitting messages, we often have additional goals. These include security goals such as confidentiality, as well as fault-tolerance goals such as error-detection/correction, and even efficiency goals such as compression.

In the first four subsections, we focus on the combination of the two basic security goals: *encryption* and *authentication*. Finally, in subsection 4.7.5, we discuss the complete *secure session transmission* protocol, which addresses additional goals involving security, reliability and efficiency, for a session (connection) between two parties. We return to these issues in Section 7.2, where we discuss the SSL/TLS protocols, including their record protocol. There are several vulnerabilities of the SSL/TLS protocols which are due to its use of the (less-preferred, often vulnerable) attacks on the SSL/TLS exploited vulnerable, insecure combinations of authentication, confidentiality and other functions,

and the record protocol of TLS 1.3 was modified to a better design, to foils such attacks.

There are two main options for ensuring the confidentiality and authentication/integrity requirements together: (1) by correctly combining an encryption scheme with a MAC scheme, or (2) by using a combined *authenticated encryption* scheme. In the first subsection below, we discuss *authenticated encryption* schemes and authenticated encryption with associated data (*AEAD*) schemes, which combine encryption (for confidentiality) and authentication. In the following subsections, we discuss specific generic constructions, combining MAC and encryption schemes.

4.7.1 Authenticated Encryption (AE) and AEAD schemes

Authenticated Encryption (AE) schemes. The combination of confidentiality and authenticity is often required, but we have seen that incorrect combinations may lead to vulnerabilities. This motivates the design of schemes which combine the authentication and the confidentiality functions. We use the term *authenticated encryption (AE)* for such schemes [281], which consist of two main functions: *encrypt-and-authenticate* EnA and *decrypt-and-verify* DnV , plus, optionally, an explicit key-generation function. The decrypt-and-verify returns ERROR if the ciphertext is found not-authentic; similar verification property can be implemented by a MAC scheme, by comparing the authenticator received with a message to the result of computing the MAC on the message. AE schemes may also have a *key-generation function*; in particular, this is necessary when the keys are not uniformly random.

In addition to the support for both encryption and authentication, there is an additional innovative aspect to the definition of AE schemes. Namely, the AE encrypt-and-authenticate operation has *three* inputs. This is in contrast to the standard definition of encryption schemes, which defines only *two* inputs: the key and the plaintext.

The third input of AE schemes is called a *nonce*. To ensure security, a different nonce value should be used whenever performing the encryption operation. Essentially, an Authenticated-Encryption resembles, therefore, a *mode-of-operation* of an encryption scheme, with the nonce taking the role of the IV or counter (state) input. The same nonce should be given to the decrypt-and-verify operation, and the scheme should ensure *correctness*, namely that for every plaintext message m , key k and nonce n holds:

$$m = DnV_k^n(EnA_k^n(m)) \quad (4.6)$$

The use of a combined AE scheme allows simpler, less error-prone implementations compared to the use of two separate schemes, one for encryption and one for authentication. In particular, we need only a call to only one function (encrypt-and-authenticate or decrypt-and-verify) instead of requiring the correct use of both encryption/decryption and MAC functions.

Many constructions of authenticated encryption are *generic*, i.e., built by combining arbitrary implementation of cryptographic schemes, following the

‘cryptographic building blocks’ principle. The combinations of encryption scheme and MAC scheme that we study later in this subsection are good examples for such generic constructions. Other constructions are ‘ad-hoc’, i.e., they are designed using specific functions. Such ad-hoc constructions may have better performance than generic constructions, however, that may come at the cost of requiring more complex or less well-tested security assumptions, contrary to the Cryptographic Building Blocks principle.

Authenticated Encryption with Associated Data (AEAD). In many applications, e.g., TLS (Chapter 7), *some* of the data to be authenticated should *not* be encrypted. Typically, this would be data that is used also by agents which do not have the secret (decryption) key; for example, the identity of the destination. Such data is often referred to as *associated data*, and authenticated encryption schemes supporting it are referred to as *AEAD* (*Authenticated Encryption with Associated Data*) schemes [279]. AEAD schemes have the same three functions (key-generation, encrypt-and-authenticate, decrypt-and-verify), and their input also includes a nonce. However, they also have an additional (fourth) input, the *associated-data* field.

Scheme	Type	Goals	Metaphor
MAC	Symmetric	Authenticity	Document with secret mark
AE	Symmetric	Authenticity and confidentiality	Document with secret mark, in sealed envelop
AEAD	Symmetric	Authenticity, and confidentiality for <i>part</i> of document	Document with secret mark, in sealed envelop with window
Signature	Asymmetric	Authenticity and non-repudiation	Signed document

Table 4.1: Authentication schemes: MAC, Authenticated Encryption (AE), Authenticated Encryption with Associate Data (AEAD) and Signatures.

Together with AE and AEAD schemes, we now have four different cryptographic authentication schemes. In Table 4.1, we sum up these four schemes; for each scheme, we list its intuitive goal, along with a metaphor to it. Like all metaphors, these should not be taken too seriously; hopefully, readers would find them helpful and not confusing.

Authenticated encryption: attack model and success/fail criteria
 We now briefly discuss the attack model (attacker capabilities) and the goals (success/fail criteria) for the combination of authentication and confidentiality (encryption), as is essential for any security evaluation (principle 1). Essentially, this combines the corresponding attack model and goals of encryption schemes (indistinguishability test) and of message authentication code (MAC) schemes (forgery test).

As in our definitions for encryption and MAC, we consider a computationally-limited (PPT) adversary. We also allow the attacker to have similar capabilities as in the definitions of secure encryption / MAC. In particular, we allow *chosen plaintext* queries, where the attacker provides input messages (plaintext) and receives their authenticated-encryption, as in the chosen-plaintext attack (CPA) we defined for encryption.

Exercise 4.13. *Present precise definitions for IND-CPA and security against forgery for AE and AEAD schemes.*

4.7.2 Authentication via EDC-then-Encryption?

Several practical secure communication systems first apply an Error-Detecting-Code (EDC) to the message, and then encrypt it, i.e.: $c = E_k(m \parallel EDC(m))$. We believe that the motivation for this design is the hope to ensure authentication as well as confidentiality, i.e., the designers were (intuitively) trying to develop an authenticated-encryption scheme. Unfortunately, such designs are often insecure; in fact, often, the application of EDC/ECC before encryption allows attacks on the confidentiality of the design. We saw one example, for WEP, in Section 2.10. Another example of such vulnerability is in the design of GSM, which employs not just an Error Detecting Code but even an Error Correcting Code, with very high redundancy. In both WEP and GSM, the encryption was performed by XORing the plaintext (after EDC/ECC) with the keystream (output of PRG).

However, EDC-then-Encrypt schemes are often vulnerable, also when using other encryption schemes. For example, the following exercise shows such vulnerability, albeit against the authentication property, when using CBC-mode encryption.

Exercise 4.14 (EDC-then-CBC does not ensure authentication). *Let E be a secure block cipher and let $CBC_k^E(m; IV)$ be the CBC-mode encryption of plaintext message m , using underlying block cipher E , key k and initialization vector IV , as in Eq. (2.58). Furthermore, let $EDCtCBC_k^E(m; IV) = CBC_k^E(m \parallel h(m); IV)$ where h is a function outputting one block (error detecting code). Show that $EDCtCBC^E$ is not a secure authenticated encryption; specifically, that authentication fails.*

Hint: attacker asks for $EDCtCBC^E$ encryption of the message $m' = m \parallel h(m)$; the output gives also the encryption of m . \square

4.7.3 Generic Authenticated Encryption Constructions

We now discuss ‘generic’ constructions, combining arbitrary MAC and encryption schemes to ensure both confidentiality and authentication/integrity. As discussed above, these constructions can be used to construct a single, combined ‘authenticated encryption’ scheme, or to ensure both goals (confidentiality and authenticity) in a system.

Different generic constructions were proposed - but not all are secure. Let us consider three constructions, all applied in important, standard applications. For each of the designs, we present the process of authenticating and encrypting a message m , using two keys - k' used for encryption, and k'' used for authentication.

Authenticate and Encrypt (A&E) , e.g., used in early versions of the SSH protocol: $C = Enc_{k'}(m)$, $A = MAC_{k''}(m)$; send (C, A) .

Authenticate then Encrypt (AtE) , e.g., used in the SSL and TLS standards: $A = MAC_{k''}(m)$, $C = Enc_{k'}(m, A)$; send C .

Encrypt then Authenticate (EtA) , e.g., used by the IPsec standard: $C = Enc_{k'}(m)$, $A = MAC_{k''}(C)$; send (C, A) .

Exercise 4.15 (Generic AE and AEAD schemes). *Above we described only the ‘encrypt-and-authenticate’ function of the authenticated-encryption schemes for the three generic constructions, and even that, we described informally, without the explicit implementation. Complete the description by writing explicitly, for each of the three generic constructions above, the implementation for the encrypt-and-authenticate (EnA) and the decrypt-and-verify (DnV) functions. Present also the AEAD (Authenticated Encryption with Associated Data) version.*

Partial solution: we present only the solution for the A&E construction. The AE implementations are:

$$\begin{aligned} A\&E.\text{EnA}_{(k', k'')}(m) &\leftarrow (Enc_{k'}(m), MAC_{k''}(m)) \\ A\&E.\text{DnV}_{(k', k'')}(c, a) &\leftarrow \begin{cases} m \leftarrow Dec_{k'}(c); \\ \text{If } a \neq MAC_{k''}(m), \text{return } m; \\ \text{Otherwise, return ERROR;} \end{cases} \end{aligned}$$

The AEAD implementations are very similar, except also with Associated Data (wAD); we present only the EnA function:

$$A\&E.\text{EnAwAD}_{(k', k'')}(m, d; r) = (Enc_{k'}(m; r), d, MAC_{k''}(m + d))$$

□

Some of these three generic constructions are insecure, as we demonstrate below for particular pairs of encryption and MAC functions. Can you identify - or guess - which? The answers were given, almost concurrently, by two beautiful papers [34, 199]; the main points are in the following exercises.

Exercise 4.16 shows that A&E is insecure; this is quite straightforward, and hence readers should try to solve it alone before reading the solution.

Exercise 4.16 (Authenticate and Encrypt (A&E) is insecure). *Show that a pair of secure encryption scheme Enc and secure MAC scheme MAC may be both secure, yet their combination using the A&E construction would be insecure.*

Solution: given any secure MAC scheme MAC , let

$$MAC'_{k''}(m) = MAC_{k''}(m) \parallel m[1]$$

where $m[1]$ is the first bit of m .

If MAC is a secure MAC then MAC' is also a secure MAC. However, MAC' exposes a bit of its input; hence, its use in $A\&E$ would allow the adversary to distinguish between encryptions of two messages, i.e., the resulting, combined scheme is not IND-CPA secure - even when the underlying encryption scheme E is secure. \square

Exercise 4.17 shows that AtE is also insecure. The argument is more elaborate than the A&E argument from Exercise 4.16, and it may not be completely necessary to understand it for a first reading; however, it is a nice example of a cryptographic counterexample, so it may be worth investing the effort. Readers may also consult [199] for more details.

Exercise 4.17 (Authenticate then Encrypt (AtE) is insecure). *Show that a pair of secure encryption scheme Enc and secure MAC scheme MAC may be both secure, yet their combination using the AtE construction would be insecure.*

Solution: Consider the following simplified version of the Per-Block Random (PBR) mode presented in subsection 2.8.2, defined for single block messages: $Enc_k(m; r) = m \oplus E_k(r) \parallel r$, where E is a block cipher; notice that this is also essentially OFB and CFB mode encryption, applied to single block messages. When the random bits are not relevant, i.e., simply selected uniformly, then we do not explicitly write them and use the simplified notation $Enc_k(m)$.

As shown in Theorem 2.1, if E is a secure block cipher (or even merely a PRF or PRP), then Enc is an IND-CPA secure encryption scheme. Denote the block length by $4n$, i.e., assume it is a multiple of 4. Hence, the output of Enc is $8n$ -bits long.

We next define a randomized transform $Split : \{0, 1\} \rightarrow \{0, 1\}^2$, i.e., from one bit to a pair of bits. The transform always maps 0 to 00, and randomly transforms 1 to $\{01, 10, 11\}$ with the corresponding probabilities $\{49.9\%, 50\%, 0.1\%\}$. We extend the definition of $Split$ to $2b$ -bit long strings, by applying $Split$ to each input block, i.e., given $2n$ -bit input message $m = m_1 \parallel \dots \parallel m_{2n}$, where each m_i is a bit, let $Split(m) = Split(m_1) \parallel \dots \parallel Split(m_{2n})$.

We use $Split$ to define a ‘weird’ variant of Enc , which we denote Enc' , defined as: $Enc'_k(m) = Enc_k(Split(m))$. The reader should confirm that, assuming E is a secure block cipher, then Enc' is IND-CPA secure encryption scheme (for $2n$ -bit-long plaintexts).

Consider now $AtE_{k,k'}(m) = Enc'_k(m \parallel MAC_{k'}(m)) = Enc_k(Split(m \parallel MAC_{k'}(m)))$, where m is an n -bits long string, and where MAC has input and outputs of n -bits long strings. Hence, the input to Enc' is $2n$ -bits long, and hence, the input to Enc is $4n$ -bits long - as we defined above.

However, AtE is *not* a secure authenticated-encryption scheme. In fact, given $c = AtE_{k,k'}(m)$, we can decipher m , using merely feedback-only CCA queries.

Let us demonstrate how we find the first bit m_1 of m . Denote the $8n$ bits of c as $c = c_1 + \dots + c_{8n}$. Perform the query $c' = \bar{c}_1 + \bar{c}_2 + c_3 + c_4 + c_5 + \dots + c_{8n}$, i.e., inverting the first two bits of c . Recall that $c = AtE_{k,k'}(m) = Enc_k(Split(m \oplus MAC_{k'}(m)))$ and that $Enc_k(m; r) = m \oplus E_k(r) \parallel r$. Hence, by inverting c_1, c_2 , we invert the two bits of $Split(m_1)$ upon decryption.

The impact depends on the value of m_1 . If $m_1 = 0$, then $Split(m_1) = 00$; by inverting them, we get 11, whose ‘unsplit’ transform returns 1 instead of 0, causing the MAC validation to fail, providing the attacker with an ‘ERROR’ feedback. However, if $m_1 = 1$, then $Split(m_1)$ is either 01 or 10 (with probability 99.9%), and inverting both bits does not impact the ‘unsplit’ result, so that the MAC validation does not fail. This allows the attacker to determine the first bit m_1 , with very small (0.1%) probability of error (in the rare case where $Split(m_1)$ returned 11). \square

Note that the AtE construction *is* secure - for specific encryption and MAC schemes. However, it is *not* secure for arbitrary secure encryption and MAC schemes, i.e., as a generic construction. Namely, Encrypt-then-Authenticate (EtA) is the only remaining candidate generic construction. Fortunately, *EtA is secure*, for any secure encryption and MAC scheme, as the following lemma states.

Lemma 4.4 (EtA is secure [199]). *Given a CPA-IND encryption scheme Enc and a secure MAC scheme MAC , their EtA construction ensures both CPA-IND and secure MAC.*

Proof sketch: We first show that the IND-CPA property holds. Suppose, to the contrary, that there is an efficient (PPT) adversary \mathcal{A} that ‘wins’ against EtA in the IND-CPA game, with significant probability. We construct adversary \mathcal{A}' that ‘wins’ in the IND-CPA game against the encryption scheme Enc , employed as part of the EtA scheme. Specifically, \mathcal{A}' generates a key k'' for the MAC function, and runs \mathcal{A} . Whenever \mathcal{A} chooses the two challenge messages m_0, m_1 , and should be provided with the authenticated-encryption of m_b , then \mathcal{A}' chooses the same two messages and receives $c* = Enc_{k'}(m_b)$. Then \mathcal{A}' uses the key k'' it generated to compute $a* = MAC_{k''}(c*)$ and return the pair $(c*, a*)$ which would be the authenticated-encryption of m_b , as required.

Similarly, whenever \mathcal{A} asks for encryption of a message m , then \mathcal{A}' uses its oracle to compute $c = Enc_{k'}(m)$, and k'' to compute $a = MAC_{k''}(c)$. \mathcal{A}' then returns the pair (c, a) to \mathcal{A} , which is exactly the required $EtA.Enc_{k',k''}(m)$.

Finally, when \mathcal{A} guesses a bit b , then \mathcal{A}' guesses the same bit. If \mathcal{A} ‘wins’, i.e., correctly guesses, then \mathcal{A}' also ‘wins’. It follows that there is no efficient (PPT) adversary \mathcal{A} that ‘wins’ against EtA in the IND-CPA game.

We next show that EtA also ensures security against forgery, as in Def. 4.1, adjusted for AE / AEAD schemes, as in Ex. 4.13. Suppose there is an efficient (PPT) adversary \mathcal{A} that succeeds in forgery of the EtA scheme, with significant probability. Namely, \mathcal{A} produces a message c and tag a s.t. $m = EtA.DnV_{k',k''}(c, a)$, for some message m , without making a query to $EtA.Enc_{k',k''}(m)$. By construction, this implies that $a = MAC_{k''}(c)$.

However, from the definition of encryption (Def. 2.1), specifically the correctness property, there is no other message $m' \neq m$ whose encryption would result in same ciphertext c . Hence, \mathcal{A} did not make a query to $Eta. EnA_{k', k''}$ that returned $MAC_{k''}(c)$ as the tag - yet \mathcal{A} obtained $MAC_{k''}(c)$ somehow - in contradiction to the assumed security of MAC. \square

Additional properties of EtA: efficiency and ability to foil DoS, CCA
Not only is EtA secure given any secure Encryption and MAC scheme - it also has three additional desirable properties:

Efficiency: Any corruption of the ciphertext, intentional or benign, is detected immediately by the verification process (comparing the received tag to the MAC of the ciphertext). This is much more efficient than encryption.

Foil DoS: This improved efficiency implies that it is much harder, and rarely feasible, to exhaust the resources of the recipient by sending corrupted messages (ciphertext).

Foil CCA: By validating the ciphertext before decrypting it, EtA schemes prevent CCA attacks against the underlying encryption scheme, where the attacker provides specially-crafted ciphertext messages, receives the corresponding plaintext (or failure indication if the ciphertext was not valid encryption), and uses the resulting plaintext and/or failure indication to attack the underlying encryption scheme. If the attacker is creating such crafted ciphertext and sending the EtA scheme, then it *should* fail the MAC validation, and would not even be input to the decryption process. Therefore, as long as the attacker cannot forge legitimate MAC, they can only attack the MAC component of the EtA, and the encryption scheme is protected from this threat.

4.7.4 Single-Key Generic Authenticated-Encryption

All three constructions above used two separate keys: k' for encryption and k'' for authentication. Sharing two separate keys may be harder than sharing a single key. Can we use a single key k for both the encryption and the MAC functions used in the generic authenticated encryption constructions (or, specifically, in the EtA construction, since it is always secure)? Note that this excludes the obvious naive ‘solution’ of using a ‘double-length’ key, split into an encryption key and a MAC key. The following exercise shows that such ‘key re-use’ is insecure.

Exercise 4.18 (Key re-use is insecure). *Let E', MAC' be secure encryption and MAC schemes. Show (contrived) examples of secure encryption and MAC schemes, built using E', MAC' , demonstrating vulnerabilities for each of the three generic constructions, when using the same key for authentication and for encryption.*

Partial solution:

A&E: Let $E_{k',k''}(m) = E'_{k'}(m) + k''$ and $MAC_{k',k''}(m) = k' + MAC_{k''}(m)$. Obviously, when combined using the A&E construction, the result is completely insecure - both authentication and confidentiality are completely lost.

AtE: To demonstrate loss of authenticity, let $E_{k',k''}(m) = E'_{k'}(m) + k''$ as above.

EtA: To demonstrate loss of confidentiality, let $MAC_{k',k''}(m) = k' + MAC_{k''}(m)$ as above. To demonstrate loss of authentication, with a hint to one elegant solution: combine $E_{k',k''}(m) = E'_{k'}(m) + k''$ as above, with a (simple) extension of Example 4.3.

The reader is encouraged to complete missing details, and in particular, to show that all the encryption and MAC schemes used in the solution are secure (albeit contrived) - only their combined use, in the three generic constructions, is insecure. \square

Since we know that we cannot re-use the same key for both encryption and MAC, the next question is - can we use two separate keys, k', k'' from a single key k , and if so, how? We leave this as a (not too difficult) exercise.

Exercise 4.19 (Generating two keys from one key). *Given a secure n -bit-key shared-key encryption scheme (E, D) and a secure n -bit-key MAC scheme MAC , and a single random, secret n -bit key k , show how we can derive two keys (k', k'') from k , s.t. the EtA construction is secure, when using k' for encryption and k'' for MAC, given:*

1. A secure n -bit-key PRF f .
2. A secure n -bit-key block cipher (\hat{E}, \hat{D}) .
3. A secure PRG from n bits to $2n$ bits.

4.7.5 Authentication, encryption, compression and error detection/correction codes

Data is often encoded for additional, non-security goals:

Reliability via error detection and/or correction, typically using Error Detection Code (EDC) such as Checksum, or Error Correction Code (ECC), such as Reed–Solomon codes. These mechanisms are designed against random errors, and are not secure against intentional, ‘malicious’ modifications. Note that secure message authentication, such as using MAC, also ensures error detection; however, as we explain below, it is often desirable to *also* use the (insecure) error detection codes.

Compression for efficiency. Compression is applied to improve efficiency by reducing message length. As we explain below, this requirement may conflict with the confidentiality requirement.

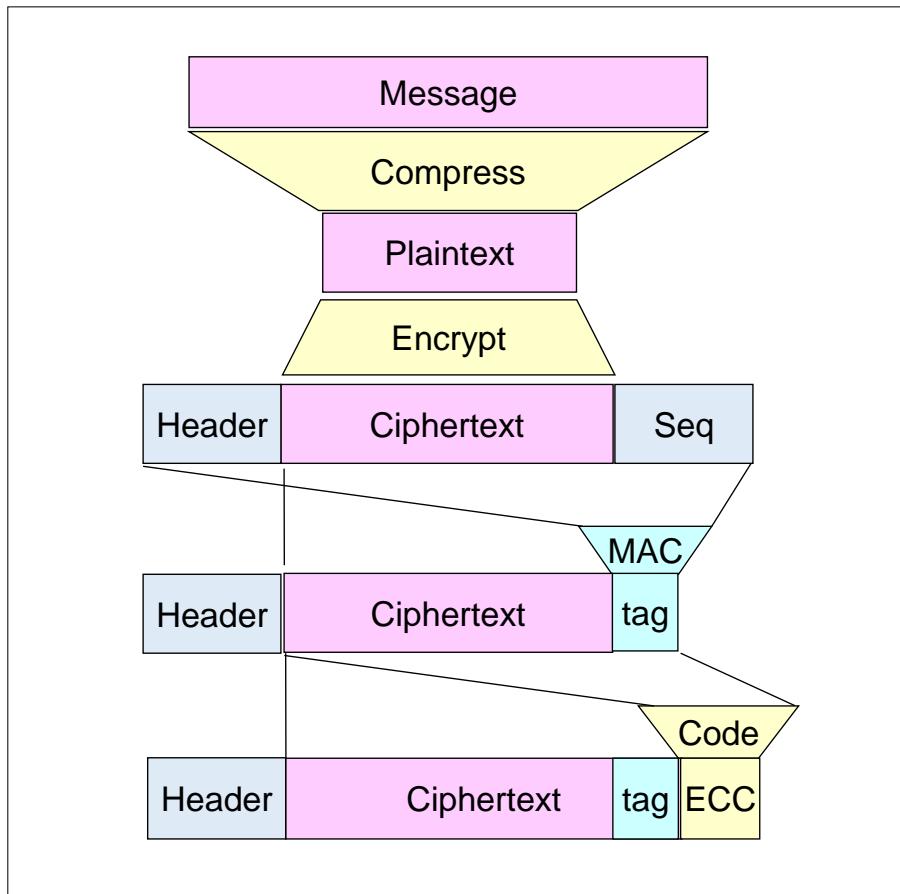


Figure 4.3: Combining Security (encryption, authentication) with Reliability (EDC/ECC) and Compression. The application of EDC/ECC *after* the MAC, allows recipients to discard messages corrupted by noise, without computing the MAC function; this saves overhead, and allows the recipient to detect attacks on the authentication.

In this subsection, we discuss how to correctly and securely combine the security goals of encryption and authentication, with these additional goals of reliability and compression (for efficiency). Fig. 4.3 presents the recommended process for combining all of these functions. Let us explain this recommendation:

- Compression is only effective when applied to data with significant redundancy; plaintext is often redundant, in which case, applying compression to it could be effective. In contrast, ciphertext would normally not have redundancy. Hence, if compression is used, it must be applied before encryption. Note, however, that this may conflict with the confidentiality requirement, as we explain below; for better confidentiality, avoid com-

pression completely, or take appropriate measures to limit the possible exposure.

- Encryption is applied next, before authentication (MAC), following the ‘Encrypt-then-Authenticate’ construction. Alternatively, we may use an authenticated-encryption with associated data (AEAD) scheme, to combine the encryption and authentication functions. Notice that by applying authentication after encryption or using an AEAD scheme, we facilitate also authentication of a sequence-number or similar field used to prevent re-play/re-order/omission, which is often known to recipient, and hence may not be sent explicitly. We can also authenticate ‘header’ fields such as destination address, which are also not encrypted, since they are used to process and route the encrypted message. The Encrypt-then-Authenticate mode also allows prevention of chosen-ciphertext attacks and more efficient handling of corrupted messages.
- Finally, we apply error correction / detection code. This allows efficient handling of messages corrupted due to noise or other benign reasons. An important side-benefit is that authentication failures of messages to which errors were not detected imply an intentional forgery attack - an attacker made sure that the error-detecting code will be correct.

Compress-then-Encrypt Vulnerability Note that there is a subtle vulnerability in applying compression before encryption, since encryption does not hide the length of the plaintext, while the length of compressed messages depends on the contents. In particular, a message containing randomly-generated strings typically does not compress well (length after compression is roughly as long as before compression), while messages containing lots of redundancy, e.g., strings composed of only one character, compress well (length after compression is much shorter). This allows an attacker to distinguish between the encryptions of two compressed messages, based on the redundancy of the plaintexts.

This vulnerability was first presented in [188], and later exploited in several attacks, including attacks on the SSL/TLS record protocol; see subsection 7.2.6 and Exercise 7.6.

4.8 Additional exercises

Exercise 4.20. *Mal intercepts a message sent from Alice to her bank, and instructing the bank to transfer 10\$ to Bob. Assume that the communication is protected by One-Time-Pad OTP encryption, using a random key shared between Alice and her bank, and by including Alice's password as part of the plaintext, validated by the bank. Assume Mal knows that the message is an ASCII encoding of the exact string Transfer 10\$ to Bob. From: Alice, PW:, concatenated with Alice's password (unknown to Mal). Show how Mal can change the message so that upon receiving it, the bank will, instead, transfer 99\$ to Mal. (The modified message must have the same password!)*

Exercise 4.21. *Let \mathcal{S} be a correct signature scheme over domain $\{0, 1\}^n$, and let $h : \{0, 1\}^* \rightarrow \{0, 1\}^n$ be a hash function whose output is n bits long. Prove that the HtS construction \mathcal{S}_{HtS}^h , defined as in Equation 3.6, is correct.*

Exercise 4.22 (TCR hash is not necessarily a MAC). *Let $h_k(m)$ be a target collision resistant (TCR) hash function (subsection 3.2.3). Show a keyed hash function $h'_k(m)$ which (1) is also TCR hash function but (2) is not a secure MAC.*

Exercise 4.23 (A MAC is not necessarily a CRHF). *Let $MAC_k(m)$ be a secure MAC function. Show a keyed hash function $h_k(m)$ which (1) is a secure MAC yet (2) is not a (keyed) CRHF or a TCR hash function.*

Exercise 4.24. *Let \mathcal{S} be a existentially unforgeable signature scheme over domain $\{0, 1\}^n$, and let $h(x + y) = x \oplus y$ be a hash function whose input is $2n$ bits long, and whose output is the n -bit string resulting from the bit-wise exclusive-OR of the most-significant n input bits, with the least significant n input bits. Show an attacker \mathcal{A} that shows that the HtS construction \mathcal{S}_{HtS}^h , defined as in Equation 3.6, is not an existentially unforgeable signature scheme.*

Exercise 4.25. *Hackme Inc. proposes the following highly-efficient MAC, using two 64-bit keys k_1, k_2 , for 64-bit blocks: $MAC_{k_1, k_2}(m) = (m \oplus k_1) + k_2 \pmod{2^{64}}$. Show that this is not a secure MAC.*

Hint: Compare to Exercise 2.49.

Exercise 4.26. *Let $F : \{0, 1\}^n \rightarrow \{0, 1\}^l$ be a secure PRF, from n bit strings to $l < n$ bit strings. Define $F' : \{0, 1\}^n \rightarrow \{0, 1\}^l$ as: $F'_k(m) = F_k(m) \parallel F_k(\bar{m})$, i.e., concatenate the results of F_k applied to m and to the inverse of m . Present an efficient algorithm $ADV^{F'_k}$ which demonstrates that F' is not a secure MAC, i.e., outputs tuple (x, t) s.t. $x \in \{0, 1\}^n$ and $t = F'_k(x)$. Algorithm $ADV^{F'_k}$ may provide input $m \in \{0, 1\}^n$ and receive $F'_k(m)$, as long as $x \neq m$. You can present $ADV^{F'_k}$ by ‘filling in the blanks’ in the ‘template’ below, modifying and/or extending the template if desired, or simply write your own code if you like.*

$$ADV^{F'_k} : \{t' = F'_k(\underline{\hspace{1cm}});$$

254 *Return* (_____); }

Exercise 4.27. Consider $CFB - MAC_k^E$, defined below, similarly to the definition of $CBC - MAC$ (Eq. (4.2)):

$$CFB - MAC_k^E(m_1 + m_2 + \dots + m_\eta) = \left\{ \begin{array}{l} c_0 \leftarrow 0^l; \\ (i = 1 \dots \eta) c_i = m_i \oplus E_k(c_{i-1}); \\ \text{output } c_\eta \end{array} \right\}$$

1. Show an attack demonstrating that $CFB - MAC_k^E$ is not a secure $l \cdot \eta$ -bit MAC, even when E is a secure l -bit block cipher (PRP). Your attack should consist of:

- a) Up to three ‘queries’, i.e., messages m , m' and m'' , each of one or more blocks, to which the attacker receives $CFB - MAC_k^E(m)$, $CFB - MAC_k^E(m')$ and $CFB - MAC_k^E(m'')$. Note: one query suffices, although you may use up to three.

- $m = \underline{\hspace{2cm}}$
- $m' = \underline{\hspace{2cm}}$
- $m'' = \underline{\hspace{2cm}}$

- b) A forgery, i.e., a pair of a message $m^F \underline{\hspace{2cm}}$ and its authenticator $a = \underline{\hspace{2cm}}$ such that $m^F \notin \{m, m', m''\}$ and $a = CFB - MAC_k^E(m^F)$.

2. Would your attack also work against the ‘improved’ variant $ICFB - MAC_k^E(m) = E_K(CFB - MAC_k^E(m))$? If not, present an attack against $ICFB - MAC_k^E(m)$:

- $m = \underline{\hspace{2cm}}$
- $m' = \underline{\hspace{2cm}}$
- $m'' = \underline{\hspace{2cm}}$
- $m^F \underline{\hspace{2cm}}$
- $a = \underline{\hspace{2cm}}$

Exercise 4.28. 1. Alice sends to Bob the 16-byte message ‘I love you Bobby’, where each character is encoded using one-byte (8 bits) ASCII encoding. Assume that the message is encrypted using the (64-bit) DES block cipher, using OFB mode. Show how an attacker can modify the ciphertext message to result with the encryption of ‘I hate you Bobby’.

2. Can you repeat for CFB mode? Show or explain why not.
 3. Can you repeat for CBC mode? Show or explain why not.
 4. Repeat previous items, if we append to the message its CRC, and verify it upon decryption.

Exercise 4.29. 1. Our definition of FIL CBC-MAC assumed that the input is a complete number of blocks. Extend the construction to allow input of arbitrary length, and prove its security.

2. Repeat, for VIL CBC-MAC.

Exercise 4.30. Consider a variant of CBC-MAC, where the value of the IV is not a constant, but instead the value of the last plaintext block, i.e.:

$$CBC-MAC_k^E(m_1 \# m_2 \# \dots \# m_\eta) = \left\{ \begin{array}{l} c_0 \leftarrow m_\eta; \\ (i = 1 \dots \eta) c_i = E_k(m_i \oplus c_{i-1}); \\ \text{output } c_\eta \end{array} \right\}$$

Is this a secure MAC? Prove or present convincing argument.

Exercise 4.31. Let E be a secure PRF. Show that the following are not secure MAC schemes.

1. ECB-encryption of the message.
2. The XOR of the output blocks of ECB-encryption of the message.

Exercise 4.32 (MAC from a PRF). In Exercise 2.38 you were supposed to construct a PRF, with input, output and keyspace all of 64 bits. Show how to use such (candidate) PRF to construct a VIL MAC scheme.

Exercise 4.33. This question discuss a (slightly simplified) vulnerability in a recently proposed standard. The goal of the standard is to allow a server S to verify that a given input message was ‘approved’ by a series of filters, F_1, F_2, \dots, F_f (each filter validates certain aspects of the message). The server S shares a secret k_i with each filter F_i . To facilitate this verification, each message m is attached with a tag; the initial value of the tag is denoted T_0 and each filter F_i receives the pair (m, T_{i-1}) and, if it approves of the message, outputs the next tag T_i . The server s will receive the final pair (m, T_f) and use T_f to validate that the message was approved by all filters (in the given order).

A proposed implementation is as follows. The length of the tag would be the same as of the message and of all secrets k_i , and that the initial tag T_0 would be set to the message m . Each filter F_i signals approval by setting $T_i = T_{i-1} \oplus k_i$. To validate, the server receives (m, T_f) and computes $m' = T_f \oplus k_1 \oplus k_2 \oplus \dots \oplus k_f$. The message is considered valid if $m' = m$.

1. Show that in the proposed implementation if the tag T_f is computed as planned (i.e. as described above), then the message is considered valid if and only if all filters approved of it.
2. Show that the proposed implementation is insecure.
3. Present a simple, efficient and secure alternative design for the validation process.

4. Present an improvement to your method, with much improved, good performance even when messages are very long (and having tag as long as the message is impractical).

Note: you may combine the solutions to the two last items; but separating the two is recommended, to avoid errors and minimize the impact of errors.

Exercise 4.34 (Single-block authenticated encryption?). Let E be a block cipher (or PRP or PRF), for input domain $\{0, 1\}^l$, and let $l' < l$. For input domain $m \in \{0, 1\}^{l-l'}$, let $f_k(m) = E_k(m \# 0^{l'})$.

1. Prove or present counterexample: f is a secure MAC scheme.
2. Prove or present counterexample: f is an IND-CPA symmetric encryption scheme.

Exercise 4.35. Let $F : \{0, 1\}^\kappa \times \{0, 1\}^{l+1} \rightarrow \{0, 1\}^{l+1}$ be a secure PRF, where κ is the key length, and both inputs and outputs are $l + 1$ bits long. Let $F' : \{0, 1\}^\kappa \times \{0, 1\}^{2l} \rightarrow \{0, 1\}^{2l+2}$ be defined as: $F'_k(m_0 \# m_1) = F_k(0 \# m_0) \# F_k(1 \# m_1)$, where $|m_0| = |m_1| = l$.

1. Explain why it is possible that F' would not be a secure $2l$ -bit MAC.
2. Present an adversary and/or counter-example, showing F' is not a secure $2l$ -bit MAC.
3. Assume that, indeed, it is possible for F' not to be a secure MAC. Could F' then be a secure PRF? Present a clear argument.

Exercise 4.36. Given a keyed function $f_k(x)$, show that if there is an efficient operation ADD such that $f_k(x+y) = \text{ADD}(f_k(x), f_k(y))$, then f is not a secure MAC scheme. Note: a special case is when $\text{ADD}(a, b) = a + b$.

Exercise 4.37 (MAC from other block cipher modes). In subsection 4.5.2 we have seen given an n -bit block cipher (E, D) , the CBC-MAC, as defined in Eq. (4.2), is a secure $n \cdot \eta$ -bit PRF and MAC, for any integer $\eta > 0$; and in Ex. 4.4 we have seen this does not hold for CTR-mode MAC. Does this property hold for...

ECB-MAC, defined as: $\text{ECB-MAC}_k^E(m_1 \# \dots \# m_\eta) = E_k(m_1) \# \dots \# E_k(m_\eta)$

PBC-MAC, defined as: $\text{PBC-MAC}_k^E(m_1 \# \dots \# m_\eta) = m_1 \oplus E_k(1) \# \dots \# m_\eta \oplus E_k(\eta)$

OFB-MAC, defined as: $\text{OFB-MAC}_k^E(m_1 \# \dots \# m_\eta) = \text{pad}_0, m_1 \oplus E_k(\text{pad}_0) \# \dots \# m_\eta \oplus E_k(\text{pad}_{\eta-1})$ where pad_0 is random.

CFB-MAC, defined as: $\text{CFB-MAC}_k^E(m_1 \# \dots \# m_\eta) = c_0, c_1, \dots, c_\eta$ where c_0 is random and $c_i = m_i \oplus E_k(c_{i-1})$ for $i \geq 1$.

XOR-MAC, defined as: $\text{XOR-MAC}_k^E(m_1 \# \dots \# m_n) = \bigoplus E_k(i \oplus E_k(m_i))$

Justify your answers, by presenting counterexample (for incorrect claims) or by showing how an adversary against the MAC function, you construct an adversary against the block cipher.

Exercise 4.38. Let (Enc, Dec) be an IND-CPA secure encryption scheme, and let $\text{Enc}'_k(m) = \text{Enc}_k(\text{Compress}(m))$, where Compress is a compression function. Show that Enc' is not IND-CPA secure.

Exercise 4.39. Figure 4.3 shows the typical sequence of operations when sending a message, with confidentiality (encryption), authentication (MAC), error detection/correction code and (optional) compression. Show the corresponding sequence of operations when receiving such message, in a figure and/or in pseudo-code. In both cases, clarify the reaction if the MAC or EDC/ECC validation fails.

Chapter 5

Shared-Key Protocols

In the previous chapters, we discussed cryptographic schemes, which consist of one or more functions. For example, MAC, PRF and PRG schemes consist of a single function, with different security criteria, e.g., for MAC, security against forgery (Definition 4.1). Similarly, encryption schemes consist of multiple functions (encryption, decryption and possible key generation), with criteria such as CPA-indistinguishability (CPA-IND, Definition 2.10).

Most often, cryptographic schemes are used as a part of a *protocol* involving two or more parties (entities). In this chapter, we focus on basic shared-key protocols used for securing the communication between two parties:

Session/record protocols secure the communication of a session between two parties, which typically includes exchange of multiple messages, using a key shared between the two parties. See Section 5.1.

Entity-authentication protocols ensure the identity of a peer involved in the communication. We discuss the vulnerable SNA protocol, and its replacement, the 2PP protocol, in Section 5.2.

Request-response protocols ensure the authentication and/or confidentiality of the communication between two parties. We discuss them in Section 5.3.

Key Exchange protocols are run between the two parties, to establish shared keys to encrypt and authenticate communication. In this chapter, we focus on *shared-key Key Exchange protocols*, which use an already shared-key between the parties, to establish keys for encryption and authentication; these protocols ensure improved security compared to direct use of the shared key for these functions. See Section 5.4.

Key distribution protocols establish shared keys between two parties, with the help of a *trusted third party (TTP)*, often referred to as the *Key Distribution Center (KDC)*. We discuss key distribution protocols in Section 5.5; much of our discussion is dedicated to studying the vulnerabilities of the GSM protocol.

Resilient Key Exchange protocols are Key Exchange protocols with mechanisms to reduce exposure due to key exposure. These include forward secrecy, perfect forward secrecy (PFS) and recover security Key Exchange protocols. See Section 5.7.

This chapter is mostly informal. In particular, we do not present rigorous definitions of security as we did in the previous chapters, and as we do in subsection 5.1.3. The main for not presenting rigorous definitions for *any* of the protocols we discuss, is that precise definitions for protocols, especially practical cryptographic protocols, require excessive complexity, much beyond what we consider appropriate for this text. We hope that the informal discussion will clarify the main issues and empower readers to properly use cryptographic protocols and avoid pitfalls. Hopefully, the discussion will also prepare readers interested in design and analysis of cryptographic protocols to more in-depth study, e.g., following [141].

5.1 Modeling cryptographic protocols

We begin our discussion by dedicating this section to informally explain what we mean by a *cryptographic protocol*, and to give intuition to how one may define security requirements.

We use the term ‘cryptographic protocols’ to refer to cryptographic algorithms that *involve interactions between two or more distinct entities*¹, including *benign entities* and an *adversary*.

In this textbook, we mostly focus on protocols involving only two benign parties, *Alice (A)* and *Bob (B)*, and on a *Man-in-the-Middle (MitM)* adversary, denoted *M*. Both benign entities run the same protocol \mathcal{P} , which is an efficient (PPT) algorithm; to model parties with different roles, e.g., client vs. server, the ‘role’ can be provided as part of their initial state. The adversary *M* runs an arbitrary efficient (PPT) algorithm, which we denote \mathcal{M} . Many extensions are possible, e.g., a simple extension allows more than two benign parties.

5.1.1 The execution process

An *execution* of a cryptographic protocol \mathcal{P} , with an adversary \mathcal{M} , is the random outcome of a process that we refer to as the *execution process*. Unfortunately, there is no widely-deployed definition of the execution process; furthermore, the more widely-known processes/models are often quite complex.

In this section, we study a relatively-simple execution model, presented in Figure 5.1, which is a simplification of the execution process of [164]. The execution process is specified by the pseudo-code in the grey rectangular in the figure, which also illustrates the fact that the all interactions between benign parties, as well as interactions with the adversary, are done via the execution

¹In some works, the term ‘cryptographic protocol’ is used in a different way, mostly to mean what we refer to as a ‘cryptographic scheme’, i.e., not necessarily involving interactions between entities; e.g., you may see mention of ‘encryption protocol’.

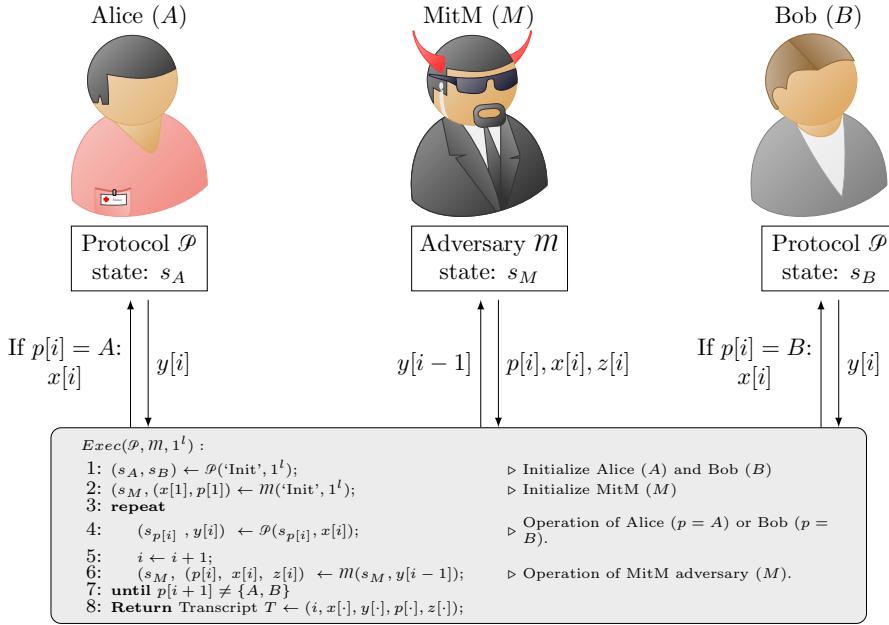


Figure 5.1: The Two-Party MitM Execution Model. The execution process $\text{Exec}(\mathcal{P}, m, 1^l)$ returns the *transcript* (result) of a run of two parties running algorithm \mathcal{P} interacting via a MitM adversary M , which runs algorithm m .

process. Furthermore, the execution process passes outputs from the benign party to the adversary, and lets the adversary control the inputs to the benign parties, i.e., the adversary has *Man-in-the-Middle (MitM)* capabilities.

Note that this execution process was simplified in several ways, including:

No corruptions: the execution process allows for a strong, MitM adversary; however, it does not allow the adversary to corrupt entities. Namely, the adversary can ‘only’ observe and control the communication, but not the parties.

Only two parties: the execution process supports only two benign parties, to whom we refer to as Alice and Bob.

Secure initialization: the execution process allows the protocol to initialize all parties, using a special initialization call of the protocol (line 1).

The execution process can be extended in different ways, e.g., for protocols with more than two benign parties, to analyze protocols that do not assume shared keys/initialization, for protocols that handle corruptions (e.g., see Section 5.7), or to define confidentiality requirements such as indistinguishability.

Let us now describe the lines (steps) of the execution process $\text{Exec}(\mathcal{P}, m, 1^l)$. In the first two lines, we initialize the parties. In line 1, we apply \mathcal{P} to initialize

the state s_A, s_B of the two benign entities, Alice (A) and Bob (B). This initialization consists simply of applying the protocol \mathcal{P} on the string ‘Init’ and the security parameter 1^l . In line 2, we initialize the MitM adversary M , by applying its algorithm \mathcal{M} on the security parameter 1^l ; the output consists of the initial state of M , denoted s_M , the number of rounds in the executions t (in unary, i.e., as 1^t , to ensure it is polynomial in 1^l), the $p[1]$ party to be invoked in the first event in the loop and the input value $x[1]$ to be provided.

Lines 6 to 8 are a loop, executed for $i = 1$ to t . Each round i of the loop includes one call to the benign entity $p[i] \in \{A, B\}$ and one call to adversary M . In every call to the algorithms (\mathcal{M} and \mathcal{P}), the execution process provides the corresponding input state (s_M or $s_{p[i]} \in \{A, B\}$), and receives the corresponding output state.

The loop begins in line 4, where the execution process invokes the protocol \mathcal{P} in benign party $p[i] \in \{A, B\}$, providing it with input interaction $x[i]$ and with its input state $s_{p[i]}$. The output consists of an output interaction $y[i]$ and a new state $s_{p[i]}$. We discuss the interactions in the following subsection.

In line 6, the execution process invokes the MitM adversary algorithm \mathcal{M} . As illustrated in Figure 5.1, the execution process provides to \mathcal{M} the output $y[i]$ of the event of the benign party, and the input state s_M . The adversary outputs a new state s_M and three other values: a value $z[i]$ to be part of the output of the execution, an identification of the benign party to be next invoked $p[i+1] \in \{A, B\}$, and the input $x[i+1]$ to be given to party $p[i+1]$.

Termination and transcript. Recall that in line 2, the adversary M outputs the number of rounds in the executions, by outputting 1^t . Hence, the execution $Exec(\mathcal{P}, \mathcal{M}, 1^l)$ terminates after t rounds, in line 8, and returns the *transcript* $T = (t, x[i], y[i], p[i], z[i])$ where $(x[i], y[i], p[i], z[i])$ are the values of the corresponding variables collected from each round $i = 1, \dots, t$ of the execution.

As we explain in subsection 5.1.3, we use the transcript to limit the capabilities used by the adversary, and to check if security requirements were satisfied, in a particular execution or (with certain probability) at a random execution.

In the following subsection, we discuss the *interactions* of the protocol and their classification into *interfaces*.

5.1.2 Interactions and interfaces

In every round of the execution (Figure 5.1), the protocol is invoked once, in line 4. The input to the party $p[i]$ consists of the current state $s_{p[i]}$ and an *input interaction* $x[i]$. The output consists of the new value for the state $s_{p[i]}$, and an *output interaction* $y[i]$.

Both input and output interactions consist of three fields, $(interface, label, parm)$:

interface: the *interface* upon which the interaction occurs. Many protocols, including those we discuss, have the following three interfaces: *APP*, *NET* and (not always) *SYS*, as illustrated in Figure 5.2.

label: the *operation* or *label* identifying the type of interaction. Different protocols usually have different labels (operations), at least in their APP interface; operations in the SYS and NET interfaces are often common among multiple protocols.

parm: the *parameter*, i.e., a value specific to a particular interaction.

The *interface* corresponds to the module with whom the protocol interacts, and the *label* identifies the *semantics* of the interaction. Most interactions also include a non-empty *value* field, e.g., a message m or time t .

In this subsection, we discuss the interfaces, focusing on the three ‘typical’ interfaces (APP, NET and SYS). Each protocol is associated with a specific set of labels (operations) on each interface. Since different protocols have different operations, we describe the interfaces and their operations for a specific protocol: the *session/record* protocol.

A session/record protocol uses a key shared between the two parties, to authenticate and/or encrypt the messages or *records* sent in a *session* or connection between them. In Chapter 7, we present a practical session/record protocol; this is the *TLS record protocol*, which is part of TLS.

Session/record protocols are among the simplest practical protocols, and are quite different from the other protocols we discuss in this chapter, so we use them as an example. In the following subsections, we continue to use the session/record protocol as an example: for *security requirements* (in the next subsection) and for the design of a specific protocol (in subsection 5.1.4).

Let us now describe the interfaces and their operations for the session/record protocol. We categorize the interactions by their *direction* (input or output) as well as their *interface* (APP, SYS or NET) and *label*, as illustrated in Figure 5.2.

Application (APP): an interface for input/output interactions between a benign party and the *application* which uses it, to whom we sometimes refer as the *user* of the protocol. These interactions provide the inputs from the application or user to the protocol running on the (benign) party, and allow the protocol to provide outputs to the application/user. For the session/record protocol, the only input interaction in the APP interface, is transmission of a message m from the application to be sent to the peer in a *send*(m) interaction. There are two output interactions, the receipt of a message m from the peer in a *received*(m) event, and an indication that the protocol cannot send information due to a communication failure, in a *failure* interaction. Here, *send*, *received* and *failure* are labels and m is a value.

Network (NET): an interface for the communication between benign parties, allowing a benign party to exchange messages with another benign party, subject to manipulations by the adversary. To send a message μ to the peer, the protocol uses the *send*(μ) output event on the NET interface; *send* is the label and μ is the value. The protocol receives a message (purportedly) from the peer in a *received*(μ) input event on the NET

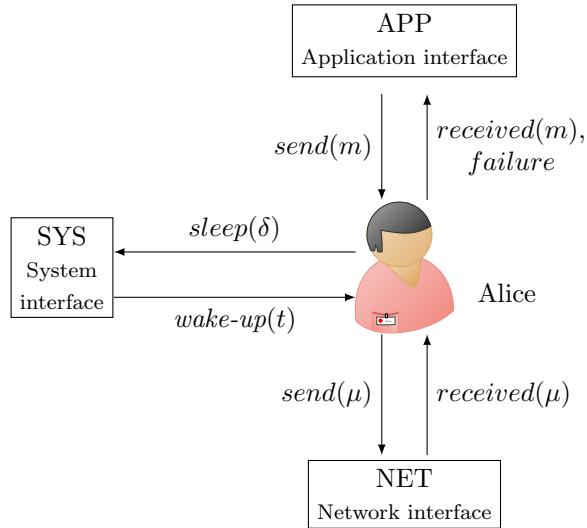


Figure 5.2: Interactions for the record/session protocols, illustrated for Alice; Bob has the same interfaces. Note that while we use the labels *send* and *received* for both the APP and NET layers, the semantics are very different; see text for details. Other protocols we study use the same SYS and NET events, but different APP events.

interface. We use the symbol μ for the messages in the NET interface, to separate them from the symbol m which we use for messages in the APP interface. Typically, μ contains an APP-interface message m , together with some *header* information; the protocol uses the header to process the message.

System (SYS): an interface to other interactions of the protocol, typically to ‘local’ services such as clock, sensors and relays/actuators. In this textbook, we only use clock service, with two interactions. Specifically, the protocol may invoke $sleep(\delta)$ to request a ‘wake-up call’ after δ time units (e.g., seconds); upon that time, the protocol should receive an incoming $wake-up(t)$ event, with t indicating the current time. To reduce notation, this interface allows both ‘wake-up’ service as well as a ‘clock lookup’ service (using $sleep(0)$).

The APP interface events are specific to each type of protocols, e.g., differ between session/record protocols and between entity-authentication and other protocols. However, all the protocols we study use the same set of SYS and NET events, as described above.

5.1.3 Adversary capabilities and security requirements

Adversary capabilities. Our description of the interactions focused on their *intended* actions; however, obviously, the adversary can significantly interfere with the interactions in all three interfaces in different ways. Following the Attack Model principle (Principle 1), we only limit the adversary’s *capabilities*, rather than assuming a specific attacker *strategy*. The adversary’s capabilities include both *computational capabilities* and *access capabilities*.

In most works in cryptography, the *computational capabilities* of the adversary are limited by requiring the adversary to be a *PPT* algorithm (see Section A.1), i.e., its run time must be polynomial in the length of its inputs. As usual, this is done by providing the security parameter 1^l as an input to the adversary (in line 2 of the execution process, Figure 5.1). Notice that the adversary outputs, in line 2, the number of rounds of the execution in unary (1^t); this ensures that the number of rounds is also polynomial in the security parameter (1^l).

The *access capabilities* define the ability of the adversary to observe the outputs and control the inputs of the benign parties. The execution process (Figure 5.1) passes *every* interaction via the adversary. We refer to this type of adversary, which observes all outputs and determines all inputs of the benign parties, as a *Man-in-the-Middle (MitM)* adversary.

Most of the work in cryptographic protocols uses the MitM adversary model. However, specific security requirements are defined for weaker adversary models, i.e., restrict the abilities of the adversary to observe outputs and/or to determine inputs of benign parties. The restrictions can be with respect to the interactions on the three interfaces, as follows:

Adversarial application. Following the *conservative design principle* (Principle 3), the execution process allows the adversary complete control and observation capabilities over the application interface. In the case of the session/record protocol, this means that the adversary determines when a $\text{send}(m)$ request from the application occurs, and what is the message m . The ability to control the input message m corresponds to the *chosen plaintext attack (CPA)* model, introduced in Chapter 2. Two examples of weaker adversary models are the *known plaintext attack (KPA)* and the *ciphertext-only (CTO)* adversary models, also presented in Chapter 2; in both, the message m is chosen randomly.

Synchronization. Session/record protocols use the SYS-interface only for the clock services, if at all; in fact, the simplified session/record protocol (Algorithm 3) does not use the clock at all. In any case, the execution process allows the adversary to completely control the clock. In reality, we expect bounded delays, and (usually) also some level of clock synchronization; realistic security requirements may restrict accordingly the adversary’s capabilities, to fixed or bounded delays, and to provide clock values and wake-up service with some level of synchronization.

Network restrictions. While most work on cryptographic protocols assume a MitM adversary, with complete control and observation capabilities for network traffic, some security requirements restrict the adversary's capabilities. In particular, an *eavesdropping adversary* can observe messages but cannot modify, inject or drop messages; note, however, that eavesdropping adversaries may be able to control the *delay* of messages.

Defining security. When the execution process terminates (line 8 of Figure 5.1), it returns the *transcript* $T = (i, x[\cdot], y[\cdot], p[\cdot], z[\cdot])$ of the execution. We use the transcript to define security requirements. As an example, let us present the *authentication requirement* of session/record protocols.

Intuitively, the execution resulting in T satisfies the existential unforgeability requirement, if the sequence of message received by Bob, denoted $M^{rcv}(T) = (m_1^B, \dots, m_{n_B}^B)$, is a prefix of the sequence of message sent by Alice, denoted $M^{sent}(T) = (m_1^A, \dots, m_{n_A}^A)$. A session/record protocol \mathcal{P} ensures existential unforgeability, if executions with every PPT adversary \mathcal{M} satisfy existential unforgeability, except with negligible probability. We next define this precisely.

Definition 5.1. Let $T = (i, x[\cdot], y[\cdot], p[\cdot], z[\cdot])$ be the transcript of an execution $\text{Exec}(\mathcal{P}, \mathcal{M}, 1^l)$ of a session/record protocol \mathcal{P} with adversary \mathcal{M} . Use $M^{sent}(T)$ to denote the sequence of messages m received by Alice in a $\text{send}(m)$ input interaction (on the APP interface). Similarly, use M^{rcv} to denote the sequence of messages m outputted by Bob in $\text{received}(m)$ interactions (on the APP interface).

The existential-unforgeability advantage $\varepsilon^{\text{EUF-Session}}(\mathcal{P}, \mathcal{M}, 1^l)$ of adversary \mathcal{M} against session/record protocol \mathcal{P} is defined as:

$$\begin{aligned} \varepsilon^{\text{EUF-Session}}(\mathcal{P}, \mathcal{M}, 1^l) &\equiv \\ &\equiv \Pr \left(\begin{array}{l} T \stackrel{\$}{\leftarrow} \text{Exec}(\mathcal{P}, \mathcal{M}, 1^l); \\ M^{rcv}(T) \text{ is not a prefix of } M^{sent}(T) \end{array} \right) \end{aligned} \quad (5.1)$$

Where the probability is taken over the random coin tosses of \mathcal{M} and \mathcal{P} during the execution resulting in transcript T , and where $M^{sent}(T)$, $M^{rcv}(T)$ are defined as above.

A session/record protocol \mathcal{P} is existentially unforgeable if for all PPT algorithms \mathcal{M} , the advantage of \mathcal{M} against \mathcal{P} is negligible, i.e.: $\varepsilon^{\text{EUF-Session}}(\mathcal{P}, \mathcal{M}, 1^l) \in \text{NEGL}(l)$.

5.1.4 \mathcal{P}_{EtA} : a simple EtA session/record protocol

In the previous subsections, we discussed the interfaces and security definitions of session/record protocols, as examples of interfaces and security definitions of cryptographic protocols. Let us now present the *simple EtA session/record protocol*, which we denote \mathcal{P}_{EtA} . This is a very simple record protocol, following the *Encrypt-then-Authenticate* (EtA) design (Section 4.7), and assuming an underlying reliable-communication service, such as provided by the widely-used

TCP standard; read about TCP in [263] and any introductory textbook on Internet protocols, e.g., [202]. The use of EtA makes \mathcal{P}_{EtA} a highly-simplified variant of the *IPsec* protocol [129], another important Internet standard, while its assumption of an underlying reliable-communication service is similar to that of the SSL/TLS protocols, which we discuss in Chapter 7.

We present \mathcal{P}_{EtA} , using pseudocode, in Algorithm 3. This provides a simple example for the definition of a protocol using pseudocode. A pseudocode definition helps to avoid ambiguities which often exist as to the operation of the protocol, and therefore, is helpful for understanding and implementing the protocol, and for analysing and proving the security of the protocol. Of course, another motivation is that EtA is an important, secure design for session/record protocols, as discussed in Section 4.7. Note that the SSL/TLS protocol which we discuss in Chapter 7, uses either the (less-preferred, often vulnerable) Authenticate-then-Encrypt (AtE) design, or the (preferred, secure) AEAD design (subsection 4.7.1).

About pseudocode description of protocols. Properly-written pseudocode provides a precise description of protocols, allowing clear understanding, implementation and analysis. Pseudocode is especially important for analysis and proofs of the *security* of protocols. However, pseudocode is not that easy to describe or understand. Therefore, for most of the protocols we discuss, we do not present pseudocode. Algorithm 3, where we present the pseudocode of the simple session/record protocol \mathcal{P}_{EtA} , serves as an example for pseudocode description of protocols. We hope that Algorithm 3 will help interested readers to understand how protocols can be precisely specified using pseudocode, as well as to better understand the interactions during an execution of a protocol. Other readers may prefer to only read the explanation of the protocol below, and ignore the references to the pseudocode (Algorithm 3).

Simplifications made by \mathcal{P}_{EtA} . The \mathcal{P}_{EtA} protocol makes the following simplifications:

Reliable network (connection): the protocol delivers messages, only if every packet sent by one party to its peer, over the network (NET interface), is delivered *reliably*. Reliable delivery means that the sequence of packets is delivered exactly as sent: no forgery, reordering, modification, duplication, or loss of packets. However, we do allow that some of the last messages sent, are not delivered. The protocol ignores ('drops') any packet received which was not sent, or received out of sequence, with a 'fail' indication to the application. In practice, this simplification implies that the protocol assumes an underlying reliable communication protocol, such as *TCP*.

No freshness: the protocol does not validate that a message was sent reasonably recently, i.e., *freshness*. Ensuring freshness requires (simple) extensions to the protocol; alternatively, freshness may be implemented by the application using the protocol. Note that non-security network

Algorithm 3 \mathcal{P}_{EtA} : a simple secure session protocol.

```

1: Protocol  $\mathcal{P}_{EtA}(s, x)$ 
2:   if  $s = \text{'Init'}$  then
3:      $k_E, k_{MAC}^S \xleftarrow{\$} \{0, 1\}^{|x|}$                                  $\triangleright$  Choose random shared keys
4:      $sent, rcved \leftarrow 0$                                           $\triangleright$  Initialize sent, received counters
5:      $s_A \leftarrow (A, B, k_E, k_{MAC}^S, sent, rcved)$ 
6:      $s_B \leftarrow (B, A, k_E, k_{MAC}^S, sent, rcved)$ 
7:     Return  $(s_A, s_B)$                                          state contains: identity, peer, both
                                                                $\triangleright$  shared keys, and sent, received counters
8:   else
9:      $(p, \hat{p}, k_E, k_{MAC}^S, sent, rcved) \leftarrow s$            $\triangleright$  Parse state into state variables, e.g.,
                                                                $p \in \{A, B\}$  is identity.
10:     $(interface, label, parm) \leftarrow x$                             $\triangleright$  Parse interaction  $x$ 
11:    for  $interface = APP$ ,  $label = send$  do
12:       $c \xleftarrow{\$} E_{k_E}(parm)$                                 $\triangleright$  Value ( $parm$ ) is message to send
                                                                $\triangleright$  Encrypt message;  $c$  is ciphertext. We
                                                               use  $getr$  since encryption is (usually)
                                                               randomized.
13:       $a \leftarrow MAC_{k_{MAC}^S}(c + (sent + 1) + p)$             $\triangleright$  Compute authenticator  $a$  of cipher-
                                                               text
14:      Return  $(p, \hat{p}, k_E, k_{MAC}^S, sent + 1, rcved), (NET, send, (c, a))$   $\triangleright$  Value ( $parm$ ) is pair  $(c, a)$  sent by
                                                               peer
15:    for  $interface = NET$ ,  $label = received$  do                     $\triangleright$  Value ( $parm$ ) is pair  $(c, a)$  sent by
                                                               peer
16:       $(c, a) \leftarrow parm$ 
17:      if  $a = MAC_{k_{MAC}^S}(c + (rcved + 1) + \hat{p})$  then
18:        Return  $(p, \hat{p}, k_E, k_{MAC}^S, sent, rcved + 1), (APP, send, D_E(c))$ 
19:      else
20:        Return  $(p, \hat{p}, k_E, k_{MAC}^S, sent, rcved), (APP, fail, parm)$ 

```

protocols that allow validation of liveness, e.g., TCP (using *keepalive*), cannot ensure freshness against a MitM attacker.

No secure delivery: the protocol does not validate that messages sent are actually received by the peer (within bounded delay). This simplifies the protocol; there is no need to send acknowledgements and to use the *wake-up* service.

No compression, padding or fragmentation: the protocol sends messages of unbounded size ‘as-is’, without applying compression and without fragmenting long messages (into multiple bounded length fragments). Furthermore, the protocol assumes that the MAC and encryption functions receive arbitrary-length messages, i.e., the protocol does not perform any padding before applying MAC and encryption.

Initialization of two shared keys: the protocol assumes initialization of a shared encryption key k_E , as well as a shared authentication key k_{MAC}^S .

For a more practical session/record protocol, see Chapter 7, where we present the *TLS record protocol*, possibly the most important and widely used session/record protocol. Note that the TLS record protocol also makes the first two simplifications, *reliable network (connection)* and *no freshness*. Practical session/record protocols that avoid all simplifications include *IPsec* [105, 129] and *DTLS* [273].

Explanation of the simple secure session/record protocol \mathcal{P}_{EtA} (Algorithm 3). The \mathcal{P}_{EtA} protocol uses two cryptographic schemes:

Encryption: a shared key cryptosystem (E, D) , using a shared key k_E .

Authentication (MAC): a Message Authentication Code MAC , using a shared key k_{MAC}^S .

Initialization. The two keys should be generated securely and then shared securely with the two parties. The execution process (Figure 5.1) supports such secure initialization. Specifically, at the very beginning of the execution (line 1), the execution process securely executes the special ‘Init’ operation of the protocol, which defines the required initialization of the parties, returning initial state for each party (s_A and s_B).

In Algorithm 3, the ‘Init’ operation is defined in line 2 to line 7; and, specifically, the keys are initialized in line 3.

The rest of the initialization process is quite ‘technical’. In line 4 we initialize the counter of sent messages ($sent$) and the counter of received messages ($rcved$). In line 5 we collect the entire initial state of each party (s_A and s_B). Throughout the execution, the state s_p of party $p \in \{A, B\}$ is the tuple $s_p = (p, \hat{p}, k_E, k_{MAC}^S, sent, rcved)$, where \hat{p} is the peer party, and $k_E, k_{MAC}^S, sent, rcved$ are as defined above.

Finally, in line 7, the initialization operation of the protocol returns the initial state for each of the two parties (Alice, A , and Bob, B). This is saved by the execution process in the corresponding state variables s_A and s_B ; see line 1 of Figure 5.1.

Post-initialization operation. Let us now discuss the operation of the protocol during the execution, i.e., after initialization. Each round, the protocol in one entity $p \in \{A, B\}$ receives as input the pair (s_p, x) , where s_p is the current state of entity p , and $x = (\text{interface}, \text{label}, \text{parm})$ is the interaction to be handled (line 4 of Figure 5.1).

This protocol only handles incoming interactions from the NET and APP interfaces, and for each of them, there is only one relevant operation (i.e., label). Let us discuss both of them.

An incoming $send(m)$ request from the application (APP interface). The protocol should send the incoming message m , properly encrypted and authenticated. This protocol follows the recommended *Encrypt-then-Authenticate* (*EtA*) approach, presented in subsection 4.7.5. Namely, the protocol first encrypts the message m (line 12). The ciphertext is $c \xleftarrow{\$} E_{k_E}(m)$; we use the $\xleftarrow{\$}$ notation to emphasize that the ciphertext c is random. Then, in line 13, the adversary computes the authenticator: $a \leftarrow MAC_{k_{MAC}^S}(c + (sent + 1) \# p)$. The input to the MAC includes the sequence number $sent + 1$ of this message-send event, and the sender identification p ; this prevents manipulation by a MitM adversary. Finally, in line 14, the protocol returns the output: an updated state (incrementing the $sent$ counter) and a $send$ request from the network

(NET interface). The network-message sent is the pair (c, a) , i.e., ciphertext and authenticator. Notice that the protocol does not send the counter $sent + 1$ or sender identification p , since the underlying networking service is assumed to ensure reliable delivery, namely, the messages should be received in the order sent, therefore, the recipient can count them and should have exactly the same inputs to the MAC; see below.

An incoming $received(\mu)$ from the network (NET interface). Since we assumed a reliable network service, the incoming network-message μ should be the same as the next (not yet received) network-message sent by the peer. If this holds, then $\mu = (c, a)$, where $a = MAC_{k_{MAC}^S}(c + (rcved + 1) + \hat{p})$. Notice that we change from p to \hat{p} and from $sent + 1$ to $rcved + 1$; this is since we now handle an interaction in the peer party (the recipient), and the relevant counter is $rcved$, the counter of received messages. The protocol therefore validates that indeed $a = MAC_{k_{MAC}^S}(c + (rcved + 1) + \hat{p})$ (line 17). If the authenticator a is valid, then the protocol passes to the application the decryption of the c , i.e., $D_{k_E}(c)$ (line 18). The decrypted message should be the $rcved + 1$ sent message; it would now also be the $rcved + 1$ received message. The protocol returns also an updated state, namely, incrementing $rcved$. If validation fails, the protocol returns *fail* indication to the application.

5.2 Entity Authentication Protocols

In this section we discuss shared-key protocols for authenticating an interaction between a pair of parties (entities). Entity authentication protocols are among the simplest and oldest cryptographic protocols, and were in extensive use for years, although later they were mostly replaced by *authenticated Key Exchange protocols*, which combine entity authentication with exchange of a *session key*. We discuss Key Exchange protocols in the following sections. We focus on the common case of *mutual entity authentication*, i.e., both parties authenticate each other; however, our discussion mostly applies also to the simpler case where only one party authenticates its peer.

For convenience, in our examples we usually have Alice initiate the handshake; however, the protocols also allow Bob to initiate handshakes. To refer to the party that initiates the handshake, we use the term *initiator*, and similarly the term *responder* for the party that responds.

We focus on *shared-key* entity authentication protocols, which use a key κ shared between the parties during initialization (line 1 of Figure 5.1).

5.2.1 Interactions and requirements of entity authentication protocols

Basically, the goal of an entity authentication protocol is to validate interaction with the (correct) peer. We focus on Mutual Entity Authentication protocols which require, at least, one exchange of messages between the two parties, e.g.,

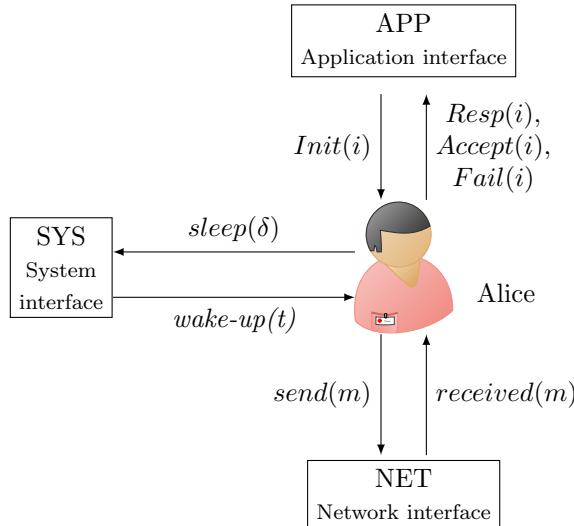


Figure 5.3: Interactions for entity authentication protocols, illustrated for Alice; Bob has the same interfaces. The *handshake identifier* i allows association of the different APP interactions related to the same handshake in each party, as explained in the text. We do not discuss protocols or requirements related to *Fail* events; you may ignore it.

a message from Alice to Bob and a response from Bob to Alice. The protocol should support *mutual* authentication; for example, after Bob receives the first message from Alice and sends her a response, Bob also expects at least one response back from Alice, so that Bob can also validate interaction with Alice.

We refer to the entire set of messages involved with a single run of the authentication protocol as the *handshake*; a handshake, therefore, requires at least two message flows (e.g., Alice to Bob and response from Bob to Alice). We will see also protocols that require three flows, i.e., also a response from Alice to Bob.

Concurrent handshakes. Most entity authentication protocols allow *multiple concurrent handshakes*. For example, a handshake begins with the protocol in Alice receiving an *Init* request from the application (APP interface), requesting to initialize a new handshake with Bob. However, before this handshake terminates, Alice is requested to initiate another handshake with Bob; or, Bob receives an *Init* request from its own application, requesting Bob to initiate a handshake with Alice, while Alice is still in the middle of an ongoing handshake. Note that the same party may also be a responder for one or more concurrent handshakes, purportedly initiated by its peer.

We elaborate on the motivation for supporting concurrent handshakes at the end of this subsection.

Interactions of entity authentication protocols. An entity authentication protocol has the interactions illustrated in Figure 5.3. The interactions in the SYS and NET interfaces are the same as presented in Figure 5.2; let us explain the APP interface interactions, i.e., the interactions between the entity authentication protocol and the application using it.

A handshake begins at an initiator, say Alice, upon $Init(i)$ input interaction (request) from the user or application (APP interface). In the responder, say Bob, the handshake begins with a $Resp(i)$ output interaction, i.e., a notification by the protocol of a new handshake initiated by the peer (e.g., Alice). The identifier i is called the *session identifier*; identifiers are required to be distinct for different handshakes in each party.

Note that each handshake is assigned a unique identifier at each party, e.g., i_I at the initiator and i_R at the responder. The two identifiers may be assigned independently at each party; they only need to be unique for that party. One simple implementation would be as a counter maintained at each party, incremented by the application when it issues a new $Init(i_I)$ request, and by the protocol when it notifies of a new handshake with a new $Resp(i_R)$ interaction.

A handshake *terminates* at a party, e.g., Alice, when she *Accepts*, i.e., validates successfully the interaction with the peer (Bob); this is done by the protocol returning $Accept$ to the application (i.e., on the APP interface). If the validation fails - e.g., Bob never responds - the handshake may never terminate, or terminate with a failure indication, by returning $Fail^2$.

To summarize, the interactions of the APP interface are as follows:

$Init(i)$ is an *input* interaction, i.e., a request from the application to initiate a new connection (with a distinct identifier i). $Init$ is the only input interaction.

$Resp(i)$ is an *output* interaction, where the protocol informs that it is responding to a new session, and assigns this new session an identifier value i . This session identifier must differ from other session identifiers used by this party (in $Init$ or $Resp$ interactions).

$Accept(i)$ is an *output* interaction, where the protocol informs the applications that session i has successfully completed.

$Fail(i)$ is an *output* interaction, used by the protocol to indicate a failure to authenticate the peer, following an $Auth(id)$ request. For simplicity, we ignore this indicator.

Security of entity-authentication protocols. Intuitively, the requirements from entity-authenticating protocols are simple: every handshake successfully terminated in one party, say Alice, should have a *matching* handshake in

²The $Fail$ indicator is used to ensure that handshakes terminate within bounded time; for simplicity, we do not discuss this requirement or utilize it in the protocols and requirements that we cover.

Bob, and vice versa. Furthermore, the mapping should be one-to-one, i.e., a handshake in Alice (Bob) must be matched to at most one handshake in the peer Bob (or Alice, respectively). Note that a handshake which does *not* terminate successfully, is not required to have a matching handshake in the peer. A secure entity-authentication protocol should have negligible probability of a handshake which violates these matching requirements.

For interested readers, let us define this security (matching) requirement more precisely, following the execution process of Figure 5.1.

Definition 5.2 (Secure matching requirement). *Let $T = (t, x[i], y[i], p[i], z[i])$ be a transcript of an execution of an entity authentication protocol (with the interactions in Figure 5.3). For $p \in \{A, B\}$ (for Alice and Bob), let I_p (R_p , A_p) denote the sequence of identifiers in $\text{Init}(i)$ (respectively, $\text{Resp}(i)$, $\text{Accept}(i)$) interactions in party $p \in \{A, B\}$.*

Let us also introduce, for $p \in \{A, B\}$, the notation \hat{p} for the ‘peer party’:

$$\hat{p} = \{A \text{ if } p = B \text{ and } B \text{ if } p = A\}$$

The matching advantage $\varepsilon^{\text{Match}}(\mathcal{P}, M, 1^l)$ of adversary M against entity authentication protocol \mathcal{P} is defined as:

$$\varepsilon^{\text{Match}}(\mathcal{P}, M, 1^l) \equiv \Pr \left(\begin{array}{l} \exists p \in \{A, B\} \text{ s.t.} \\ \text{One-to-one functions } f, g : \\ f : A_p \rightarrow (I_{\hat{p}} \cup R_{\hat{p}}), g : R_p \rightarrow I_{\hat{p}} \\ \text{s.t. } (\forall i \in A_p \text{ s.t. } f(i) \in R_p) i = g(f(i)) \end{array} \right) \quad (5.2)$$

Where the probability is taken over the random coin tosses of M and \mathcal{P} during the execution resulting in the transcript T .

An entity authentication protocol \mathcal{P} is securely matching if for all PPT algorithms M , the advantage of M over \mathcal{P} is negligible, i.e.: $\varepsilon^{\text{Match}}(\mathcal{P}, M, 1^l) \in \text{NEGL}(l)$.

Sequential vs. Concurrent Authentication Note that ensuring mutual-authentication is easier, if handshakes must be done *sequentially*, i.e., no *concurrent handshakes*. However, support for multiple concurrent sessions is a critical aspect of Mutual Entity Authentication protocols. Concurrent sessions are required in many scenarios; e.g., web communication often uses concurrent connections (sessions) between browser and server, to improve performance.

Concurrent handshakes are also necessary to prevent ‘lock-out’ due to synchronization-errors (e.g., lost state by Initiator), or an intentional ‘lock-out’ by a malicious attacker, as part of a *denial-of-service attack*. In any case, there is no strong motivation to allow only consecutive handshakes; protocols that support concurrent handshakes are as efficient, and not significantly more complex, than protocols that only allow sequential authentications. Therefore, following the conservative design principle (Principle 3), we focus on the case where concurrent handshakes are allowed.

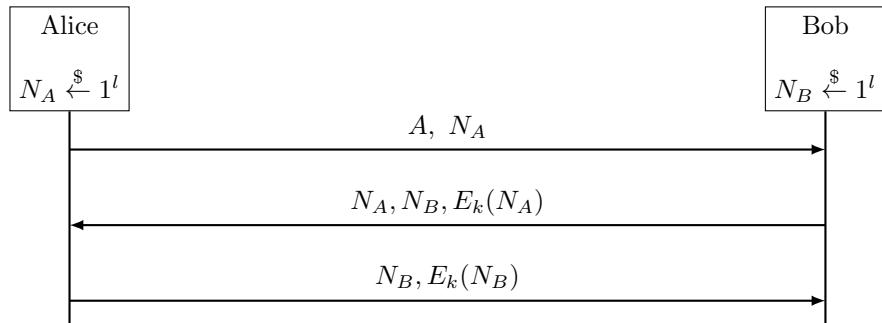


Figure 5.4: The (vulnerable) SNA mutual authentication protocol.

When designers do not consider the threats due to concurrent sessions, yet the implementation allows concurrent sessions, the result is often a vulnerable protocol. This is a typical example of the results of failures to articulate the requirements from the protocol and the adversary model, and of *not* following Principle 3. We next study such vulnerability: the SNA mutual-authentication protocol.

5.2.2 Vulnerability case study: SNA mutual-authentication protocol

As a simple, yet realistic, example of an (insecure) two-party, shared key Mutual Entity Authentication protocol, consider the *SNA mutual-authentication protocol*. IBM's SNA (Systems Network Architecture) was the primary networking technology from 1974 till the late 1980s, and is still in use by some 'legacy' applications.

We describe the original, insecure version of the SNA Mutual Entity Authentication protocol, and later its replacement - the 2PP Mutual Entity Authentication protocol. Both protocols use a shared secret key k , to authenticate two parties to each other, without deriving a session key; we later describe extensions which also provide Key Exchange. We first explain the SNA protocol, illustrated in Figure 5.4, and then discuss its security.

The SNA Mutual Entity Authentication protocol operates in three simple flows, as illustrated in Figure 5.4. The protocol uses a block cipher E . The initiator, say Alice, sends to its peer, say Bob, her identifier, which we denote A , and N_A , a random l -bit binary string which serves as a challenge; such a random challenge is often called a *nonce*. Here, l is the size of the inputs and outputs to a block cipher E used by the protocol.

The responder, say Bob, replies with a *proof of participation* $E_k(N_A)$, using the shared key k . Bob also sends his own random l -bit challenge (*nonce*) N_B . To help Alice match this response with the correct handshake, as necessary to support concurrent handshakes, Bob also attaches Alice's nonce N_A (or another handshake identifier, not related to security).

Upon receiving Bob's response, Alice validates that the response contains the correct function $E_k(N_A)$ of the nonce that she previously selected and sent. If so, Alice concludes that she communicates indeed with Bob. Alice then completes the Mutual Entity Authentication by sending her own 'proof of participation' $E_k(N_B)$. Alice also includes N_B , to help Bob match the response with the correct handshake, similarly to the inclusion of N_A by Bob.

Finally, Bob similarly validates that he received the expected function $E_k(N_B)$ of its randomly selected nonce N_B , and concludes that this Mutual Entity Authentication was initiated by Alice.

Both parties, Alice and Bob, signal successful completion of the Mutual Entity Authentication, using *Accept* interactions, (only) upon receiving the expected response ($E_k(N_A)$ for Alice and $E_k(N_B)$ for Bob).

Exercise 5.1. Present precise, concise pseudo-code for the SNA protocol as presented above and in Figure 5.4. Use the interactions in Figure 5.3.

SNA Mutual Entity Authentication ensures sequential, but not concurrent, mutual authentication. The simple SNA Mutual Entity Authentication of Figure 5.4 *ensures mutual authentication if restricted to sequential Mutual Entity Authentication* - but is *vulnerable when allowing concurrent Mutual Entity Authentication*. The attack is illustrated in Figure 5.5.

Let us first explain why the protocol ensures mutual authentication, when restricted to a *sequential handshakes*. Suppose, first, that Alice completes the protocol successfully. Namely, Alice received the expected second flow, $E_k(N_A)$. Assume that this happened *without* Bob previously receiving N_A as the first flow from Alice (and sending $E_k(N_A)$ back). Due to the sequential restriction, Alice surely did not *receive* N_A as a challenge in the time since Alice sent N_A , and hence did not compute and send $E_k(N_A)$. Since Alice selected N_A randomly, from a sufficiently large set (i.e., N_A is sufficiently long), it is unlikely that either Alice or Bob has received N_A before Alice completed the protocol. Hence, the adversary must have computed $E_k(N_A)$ rather than intercepted it. However, if the adversary can compute $E_k(N_A)$ then the adversary can distinguish between E_k and a random permutation, contradicting the PRP assumption for E . Note that an eavesdropping attacker may collect such pairs $(N_A, E_k(N_A))$ or $(N_B, E_k(N_B))$, however, since N_A, N_B are quite long strings (e.g., 64 bits), the probability of such re-use of the same N_A, N_B is negligible.

However, the SNA Mutual Entity Authentication fails to ensure *concurrent* mutual authentication; namely, if the parties are willing to run two concurrent Mutual Entity Authentication handshakes, then an attacker can cause a party to complete the protocol 'successfully', i.e., thinking it has communicated with its peer, while in reality the peer did not receive any message. For example, Figure 5.5 illustrates an attack where Alice initiates one session with Bob, which is actually intercepted by an attacker. The attacker, impersonating as Bob, initiates another session with Alice. Both sessions terminate correctly, i.e., in both, Alice is tricked into believing that it has successfully interacted with Bob; however, in reality, Bob was never involved.

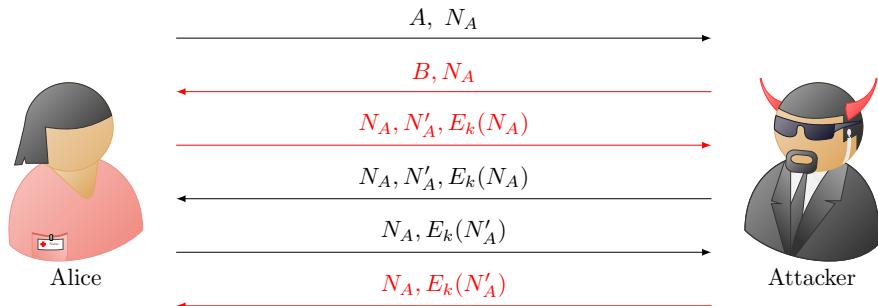


Figure 5.5: Attack on SNA Mutual Entity Authentication with Alice initiating one session with Bob, which is actually intercepted by an Attacker; flows related to this session are marked in black and ‘regular’ arrows. In the attack, the Attacker, impersonating as Bob, initiates another session with Alice; flows related to this (second) session are marked in red and with dashed arrows. Both sessions terminate correctly, i.e., Alice would believe that it has successfully interacted with Bob in both sessions, while in reality, Bob was never involved. Notice that this attack requires that Alice would agree to act both as an initiator and as a responder of sessions.

Notice that the attack of Figure 5.5 requires that Alice would agree to act both as an initiator of a session and as a responder of a session. One may hope that SNA may be secure for concurrent Mutual Entity Authentication, if each party is only willing to act in one role (initiator or responder). However, this is not the case; see Exercise 5.9.

In practice, SNA allows concurrent Mutual Entity Authentication handshakes - and for good reasons, as motivated earlier (at the end of subsection 5.2.1). In particular, this is essential for *resiliency* and *availability*. To ensure security, the Mutual Entity Authentication protocol of SNA was changed into the *2PP* protocol, which we present in the following subsection.

Authentication-Protocol Design Principles. Before we present secure protocols, it is useful to identify weaknesses of the SNA protocol, exploited in the attacks on it, and derive some *design principles*, applicable to the authentication performed by different protocols.

Authenticate sender/recipient. The SNA protocol allowed *redirection*: giving to *Bob* a value from a message which was originally sent - in this case, by *Bob* himself - to a different entity (*Alice*). This motivates the following design principle: *identify the sending and/or receiving party in each message*, e.g., by authenticating appropriate identifier(s). An even better solution may be to use independently-pseudorandom keys for sending messages by the two parties.

Authenticate the handshake identifier. The SNA attack sent to *Bob* (part) of a message sent (by *Bob*) to *Alice*, during a different handshake. This

motivates the design principle of authenticating the handshake-identifier (i in Definition 5.2).

Authenticate flow number and initiator/responder bit. The attack sent in the third protocol flow (second message from the initiator), the authenticator received from the second protocol flow (first message from the responder). This motivates us to *authenticate the flow number (e.g., second vs. third) and a bit indicating if authenticator is sent by initiator or responder.*

Authenticate using MAC or signatures. The SNA Mutual Entity Authentication protocol was designed to ensure authentication; however, it uses a block cipher, instead of using an authentication function such as MAC or signatures. Admittedly, from the *Switching lemma* (Lemma 2.2), a block cipher, i.e., a PRP, is also a PRF, and from the PRF-is-a-MAC lemma (Lemma 4.1), every PRF is also a MAC. However, SNA uses the block cipher as encryption, not as a MAC; a VIL MAC would allow longer inputs.

Never provide an oracle. The protocol applied the block cipher to input - the nonces - received entirely from the network, i.e., fully controlled by a MitM adversary, and returns the result of applying the block cipher to this input. Namely, this provides the adversary with an ‘oracle’ to the cryptographic function - in this case, the block cipher (PRP). This could have made it easier for the adversary to attack the block cipher, compared with a protocol that does not provide such ‘oracle’. A simple way to avoid providing ‘oracle’, which works in many scenarios (including Mutual Entity Authentication protocols), is to include some random input(s) before applying the cryptographic function, in addition to the adversary-provided inputs.

Note that adopting even a subset of these design principles would have sufficed to prevent the attack of Figure 5.5.

5.2.3 Secure Mutual Entity Authentication with the 2PP protocol

In this subsection we present *2PP*, a secure two party shared-key Mutual Entity Authentication protocol; the name *2PP* simply stands for *two party protocol*. The 2PP protocol, presented in [53], was a replacement to the insecure SNA Mutual Entity Authentication protocol.

The flows of the 2PP Mutual Entity Authentication protocol are presented in Figure 5.6. Like in the SNA Mutual Entity Authentication protocol, the values N_A and N_B are n -bit nonces, where n is the security parameter - typically, the length of the shared key k . The nonces N_A, N_B are selected randomly, by Alice (initiator) and Bob (responder), respectively.

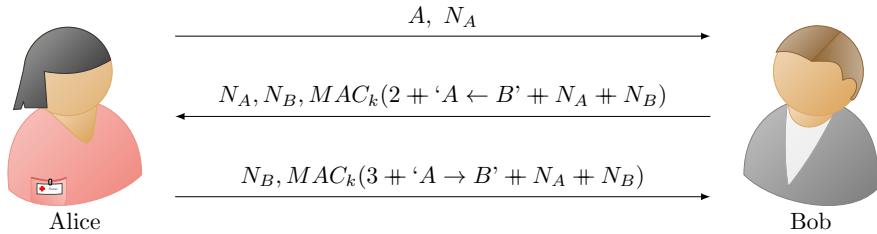


Figure 5.6: The 2PP Mutual Entity Authentication protocol.

The protocol, at both parties, outputs $\text{Accept}(i)$ once it authenticates correctly the last flow from the peer for handshake i (second flow for the initiator, third flow for the responder). By validating this last flow, 2PP ensures mutual authentication (Definition 5.2).

We will not *prove* the security of 2PP, but let us give an intuitive argument why it ensures mutual authentication, i.e., authentication of both responder and initiator. Note that the security of 2PP requires the MAC scheme to be secure, and requires that the key k and the nonces are ‘sufficiently long’, i.e., at least as long as the security parameter 1^l .

Security of 2PP: responder authentication. Consider an execution in which the initiator (Alice in Figure 5.6) ‘accepts’, although Bob is not responding. In 2PP, Alice ‘accepts’ (only) upon receiving $MAC_k(2 + A \leftarrow B + N_A + N_B)$. In 2PP, a party authenticates a message beginning with 2 only when sending a response, and according to the identifiers, Alice would never send this. If Bob sent it after Alice began this handshake, then responder authentication holds. If Bob computed it *before* Alice began this handshake, then Alice randomly selected same N_A as previously received by Bob, which would occur with negligible probability, since N_A and N_B are of the same length as the security parameter 1^l .

There remains the possibility that Bob also did not compute $MAC_k(2 + A \leftarrow B + N_A + N_B)$. In this case, the adversary somehow found this value ‘on its own’, i.e., neither Alice nor Bob computed it. Such an adversary can produce a valid MAC on a message that was never MAC-ed by an ‘oracle’ knowing the MAC key; this contradicts the assumption that the MAC scheme used is secure. Hence, 2PP seems to ensure responder authentication.

Security of 2PP: sender authentication. The argument for sender authentication is very similar; we leave it to the reader to work it out!

5.3 Authenticated Request-Response Protocols

Authenticated request-response protocols extend mutual authentication protocols such as 2PP: not only do they authenticate the *entities*, they further

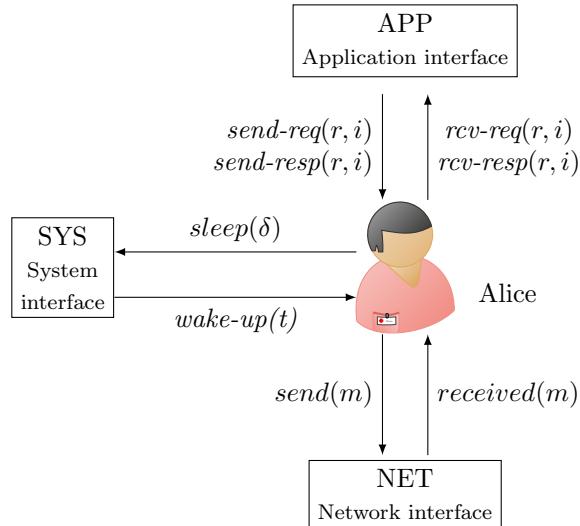


Figure 5.7: Interactions for Authenticated Request-Response Protocols, illustrated for Alice; Bob has the same interfaces. Each party may send either a request or a response (to a previously received request).

authenticate the exchange of a *request* and corresponding *response* messages between the two parties. More precisely, they ensure the following properties:

Request authentication: every request received by a party, was sent by its peer.

Response authentication: every response received by a party, was sent by its peer, in response to a request sent by the party.

No Replay: every request/response is received by a party, at most the number of times it was sent by the peer.

These notions are hopefully quite clear; however, the careful reader may note that they are not yet a precise definition of security for authenticated request-response protocols. Such a precise definition is a bit subtle; see Note 5.1. We expect many readers will prefer to proceed with a more intuitive discussion of these protocols.

Let us first describe the interactions of authenticated request-response protocols, over the different interfaces (APP, NET and SYS). These are illustrated in Figure 5.7. Notice that the interactions on the NET and SYS interfaces are exactly the same as in Figure 5.3 and Figure 5.2. It only remains to explain the interactions on the APP interface, which are:

send-req(r, i): the application instructs the protocol to send request r to the peer, using the identifier i to identify the response (if and when it arrives). The identifier should be unique, i.e., different from previous identifiers

used by the application in $send\text{-}req(r, i)$ interactions, e.g., a sequential number of this $send\text{-}req$ interaction (in this entity).

$rcv\text{-}req(r, i)$: the protocol delivers an incoming request r from the peer, specifying identifier i to be used by the application if and when it provides the corresponding response. The identifier i should be unique, i.e., different from previous identifiers used by the protocol in $rcv\text{-}req(r, i)$ events. Note that this identifier is unrelated to these used to identify $send\text{-}req(r, i)$ interactions; it is Ok to use the same identifier for both purposes as this will not cause anomaly.

$send\text{-}resp(r, i)$: the application instructs the protocol to send to the peer a response, r . The response r is to a previously-received request, which was identified by i ; a $send\text{-}resp(r, i)$ may occur only if a $rcv\text{-}req(r', i)$ occurred earlier. The previous $rcv\text{-}req(r', i)$ must have the same identifier i , but the request r' is usually different from the response, i.e., usually, $r' \neq r$.

$rcv\text{-}resp(r, i)$: the protocol delivers a response r from the peer, specifying identifier i ; this must be the same identifier as in a previous $send\text{-}req(r', i)$ interaction in this entity. Note that the value r of the response is typically different from the value r' of the corresponding request.

5.3.1 Summary of request/response protocols

We discuss four authenticated request/response protocols:

2PP-RR: this protocol extends 2PP by also authenticating a request message from the responder (e.g., Bob) and the corresponding response message from the initiator (e.g., Alice). Note the ‘reverse’ role of the parties: the ‘initiator’ is the party sending the response, while the ‘responder’ is the party sending the request. This is a bit confusing, and worst, it is often inconvenient for applications.

2RT-RR: this protocol allows the initiator to send a query and the responder to respond, which is often the required usage pattern; however, this requires four flows.

Counter-based-RR: this protocol authenticates a session rather than just request-response, and requires only one flow per each message. However, it requires both entities to maintain persistent state (counter).

Time-based-RR: authenticates a request message and the corresponding response message, using only the minimal two flows: one for the request and one for the response. These protocols require both parties to have *synchronized clocks*. The protocol can handle (bounded) delay and clock drift, but this requires the responder to maintain persistent state for a limited time.

Note 5.1: Defining security of authenticated request-response protocols.

A precise security definition of authenticated request-response is more subtle than may be expected. We present it here, since some readers may find it helpful, but we expect many readers may prefer to skip it and to use the intuitive understanding of the security requirements. For this reason, our explanation is also a bit terse.

A pair of two APP-interface interactions in an execution are *matching* if:

1. The first is a $send\text{-}req(r, i)$ interaction in peer $p \in \{A, B\}$, and the second is a $recv\text{-}req(r, i')$ interaction, occurring later and in the other peer, where both have the same request r .
2. The first is a $recv\text{-}req(r, i')$ interaction, and the second is a $send\text{-}resp(r', i')$ interaction, both in the same peer and with the same identifier i' .
3. The first is a $send\text{-}resp(r', i')$ interaction, and the second is a $recv\text{-}resp(r', i)$ interaction, occurring later and in the other peer, where both have the same response r' .

A sequence of up to four APP-interface interactions is *matching* if and consecutive pairs in the sequence are matching; a sequence of four interactions is matching, if consecutive pairs in the sequence are matching and, in addition, the first and last (fourth) interactions, have the same identifier i .

An execution T has valid matching if there is a mapping of the APP-interface interactions to matching sequences of one to four interactions, such that;

1. Every $recv\text{-}resp$ interaction is the fourth (and last) interaction in a sequence of four interactions.
2. Every $send\text{-}resp$ interaction is the third interaction in a sequence.
3. Every $recv\text{-}req$ interaction is the second interaction in a sequence.

Definition 5.3 (Security of authenticated request-response protocols). *Let $T = (t, x[i], y[i], p[i], z[i])$ be a transcript of an execution $Exec(\mathcal{P}, m, 1^l)$ of an authenticated request-response protocol \mathcal{P} (with the interactions in Figure 5.7) and let $p \in \{A, B\}$ (for Alice and Bob).*

The Request-Response matching advantage $\varepsilon^{RR\text{-}matching}(\mathcal{P}, m, 1^l)$ of adversary m against entity authentication protocol \mathcal{P} is defined as:

$$\varepsilon^{RR\text{-}matching}(\mathcal{P}, m, 1^l) \equiv \Pr(\text{execution has no valid matching}) \quad (5.3)$$

Where the probability is taken over the random coin tosses of m and \mathcal{P} during the execution.

A request-response protocol \mathcal{P} is RR-matching if for all PPT algorithms m , the advantage of m over \mathcal{P} is negligible, i.e.: $\varepsilon^{RR\text{-}matching}(\mathcal{P}, m, 1^l) \in NEGL(l)$.

Protocol	Figure	Section	Flows	Requirements/drawbacks
2PP-RR	5.8	5.3.2	3	Responder sends request
2RT-2PP-RR	5.9	5.3.3	4	Two round-trips
Counter-based-RR	5.10	5.3.4	2	Persistent state
Time-based-RR	5.11	5.3.5	2	Synchronization

Table 5.1: Authenticated Request-Response (RR) protocols.

We compare these four types of authenticated request-response protocols in Table 5.1. We notice that one of the most important differences among these protocols is the number of flows, which implies the number of *round trip times* required by the protocol. This number of round trip times is an important consideration in many practical scenarios; let us explain why.

Round trip times are significant. Minimization of round trips is important in many practical scenarios. This fact may not be a bit obscured by the fact that, for simplicity, our sequence diagrams use horizontal arrows for transmissions, i.e., ignore the delays of messages. We adopt this simplification since in this textbook we mostly avoid discussing the details of networking and related delays, since their relation to applied cryptography is limited, and we prefer to keep the text usable also for students without background in networking. However, since round trip times are so important, we next state it as a fact and briefly explain it.

Fact 5.1 (Round trip times are significant). *Typical round-trip times over the Internet can be quite significant, often 0.1 second or more. This holds even when using connections with fast transmission rate. For example, sending a typical request-response, say, each containing 10,000 bytes, over a connection with a (not very high) transmission rate of 10 million bytes/second, would take the number of round-trips required (times, say, 0.1 seconds), plus 0.02 seconds for the transmission time. Clearly, the transmission time is negligible compared to the round-trip time. The delay is dominated by the number of round-trips and their round-trip delay.*

5.3.2 The 2PP-RR Authenticated Request-Response Protocol.

We first discuss 2PP-RR, a three flow nonce-based authenticated Request-Response protocol, which is a minor extension to 2PP. The 2PP-RR protocol is illustrated in Figure 5.8. In fact, the only change compared to the 2PP protocol (Figure 5.6), is the addition of the request (*req*) from responder to initiator, and of the response (*resp*) from initiator to responder, to the second and third flows, respectively.

The 2PP-RR protocol is simple and not too difficult to prove secure, by a reduction to the security of the underlying MAC function. Namely, suppose that we know an efficient algorithm (adversary) \mathcal{M} which shows that 2PP-RR

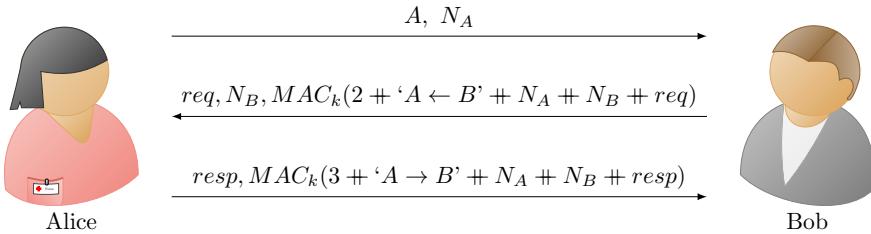


Figure 5.8: The 2PP-RR protocol: a three flow nonce-based authenticated Request-Response protocol, based on 2PP

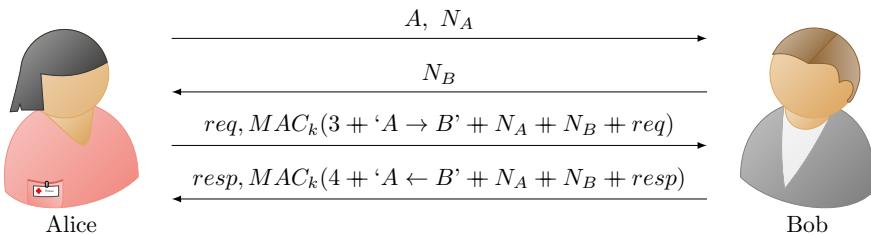


Figure 5.9: 2RT-2PP RR: a two-round-trips Authenticated Request-Response protocol

does not meet the definition of a secure authenticated request-response protocol (Definition 5.3). We can show that m produces some MAC without knowing the key, allowing an efficient adversary against the MAC function. Hence, if we use a secure MAC, then 2PP-RR is secure.

This protocol has, however, a significant drawback, which makes it ill-suited for many applications. Specifically, in this protocol, the request is sent by the responder, and the initiator sends the response. In most applications, it makes sense for a party to initiate the protocol when it needs to make some request, rather than to wait for the initiator to contact it and only then, as a responder, send the response. The next protocol is a different adaptation of 2PP which avoids this drawback - but requires four flows, i.e., two full round trips.

5.3.3 2RT-2PP Authenticated Request-Response protocol

In Figure 5.9 we present *2RT-2PP RR*, another authenticated request-response protocol based on 2PP. As the name implies, the 2RT-2PP RR protocol requires *four flows*, i.e., *two round-trips*; this is a significant drawback. However, 2RT-2PP improves upon 2PP-RR in that it authenticates a request from the initiator, and the corresponding response to it from the responder.

The 2RT-2PP Request-Response protocol involves two simple extensions of the basic 2PP protocol. The first extension is the transmission and authentication of the request and response, similarly to their addition in 2PP-RR. The second extension is an additional (fourth) flow, from the responder back to the initiator, which carries the response of the responder to the request from

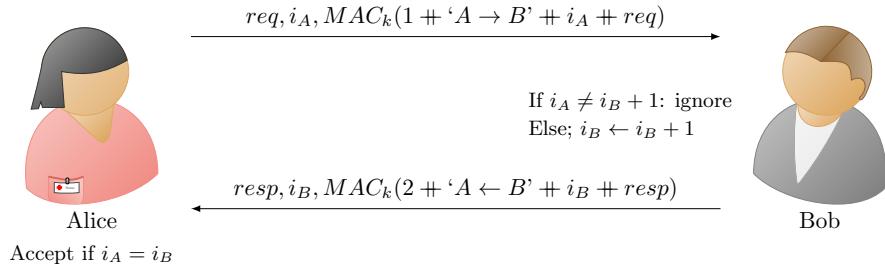


Figure 5.10: Counter-based Authenticated Request-Response protocol

the initiator. In a sense, 2RT-2PP ‘splits’ the contents of the second flow of the 2PP-RR. In 2RT-2PP, these contents are split between the second flow (providing the nonce N_B) and the fourth flow (providing the authenticated response).

5.3.4 Counter-based Authenticated Request-Response protocol

In Figure 5.10 we present the *Counter-based Authenticated Request-Response protocol*. In contrast to the 2PP protocols, this protocol requires only *one round trip* - sending the (authenticated) request and receiving the (authenticated) response. However, to prevent replay of previously-sent requests, in only one round-trip, this protocol requires *both parties to maintain a synchronized counter*.

The challenge for this protocol, as well as for the time-based protocol of the next subsection, is for the responder to verify the *freshness* of the request, i.e., that the request is not a *replay* of a request already received in the past. Freshness also implies no *reordering*; for example, a responder, say Bob, should reject request x from Alice, if Bob already received request x or a later-sent request x' from Alice. Freshness prevents an attacker from replaying information from previous exchanges. For example, consider the request-response authentication of Figure 5.9; if N_B is removed (or fixed), then an eavesdropper to the flows between Alice and Bob in one request-response session can copy these flows and cause Bob to process the same request again. For some requests, e.g., *Transfer \$100 from my account to Eve*, this can be a concern.

To ensure *freshness* without requiring the extra flows, one may use *state*, as in this subsection, or *synchronization*, as in the next subsection.

Specifically, the counter-based protocol of Figure 5.10 requires both parties to maintain a counter; we use i_A to denote the counter kept by Alice, and i_B to denote the counter kept by Bob. Alice’s counter i_A represents the number of queries that Alice sent, and Bob’s counter i_B represents the number of responses that Bob sent; hence, both are initialized to zero. The protocol

maintains these two counters synchronized, in the sense that at any time holds: $i_B \leq i_A \leq i_B + 1$.

Note that this design implies that this protocol does not allow concurrent transmission of requests. Furthermore, the protocol does not provide retransmissions or any other mechanisms to handle message-losses or corruptions; any such loss or corruption is likely to prevent any further query/response. However, it is not too difficult to extend the protocol to handle such issues, in particular, to allow concurrent requests and responses.

Exercise 5.2. Extend the protocol of Figure 5.10, to allow Alice to send concurrent requests to Bob; allow Bob to respond to requests, even when they are received out-of-order.

5.3.5 Time-based Authenticated Request-Response protocol

Figure 5.11 presents another alternative single-round Authenticated Request-Response protocol; this variant allows the Initiator (e.g., Alice) to be stateless, and also limits the time that the responder (e.g., Bob) must keep state. Instead of relying on a counter maintained, in synchronized way, by the two parties, the protocol of Figure 5.11 relies on the use of *time* and on two *synchronization assumptions*, specifically, *bounded delay* and *bounded clock skew*.

Bounded delay assumption. Let $\Delta_{\text{delay}} \geq 0$ denote a bound on the *maximal delay*. Namely, if one party sends a message at time t , then this message is received by $t + \Delta_{\text{delay}}$ or earlier.

Bounded clock skew. Let $\Delta_{\text{skew}} \geq 0$ denote a bound on the *maximal clock skew*, i.e., the maximal difference between the values of the clocks of two entities at any given time. Let $\text{clk}_A(t)$ ($\text{clk}_B(t)$) denote the value of the clock at Alice (respectively, Bob) at time t ; then we have:

$$\text{clk}_A(t) - \Delta_{\text{skew}} \leq \text{clk}_B(t) \leq \text{clk}_A(t) + \Delta_{\text{skew}} \quad (5.4)$$

The protocol is illustrated in Figure 5.11, with Alice sending the request and Bob responding. For simplicity, we use a combined bound: $\Delta \equiv \Delta_{\text{skew}} + \Delta_{\text{delay}}$, and the notation $\text{clk}_A(\cdot)$, $\text{clk}_B(\cdot)$ for the value of clk_A (respectively, clk_B) at the time Alice sends (Bob receives) the *req* message.

The protocol at Bob confirms the received request *req* is valid, as follows:

No modification: compare the received MAC value to the MAC computed with the correct inputs.

***req* is a request from Alice to Bob:** The fact that the input to the MAC begins with $1 \# 'A \rightarrow B'$ ensures this is a request (first flow) from Alice to Bob.

No replay: Bob validates that the received value of T_A is larger than the largest previously received value of T_A .

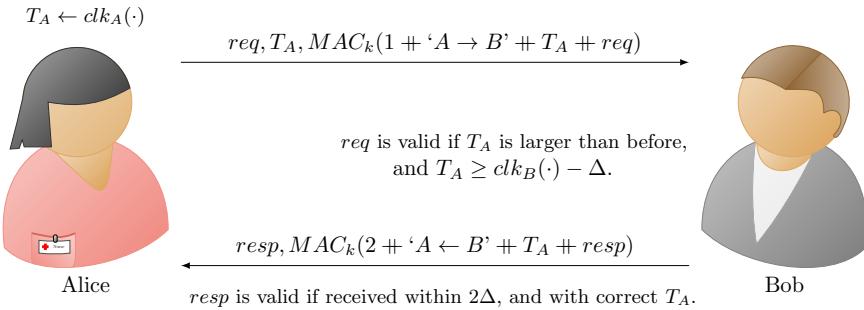


Figure 5.11: Time-based Authenticated Request-Response protocol, using a bound Δ on the maximal delay plus maximal clock bias. We use $clk_A(\cdot)$ to denote the time according to the local clock of Alice upon sending req , and $clk_B(\cdot)$ for Bob's clock upon receiving req . Alice sets $T_A \leftarrow clk_A(\cdot)$ when she sends the request, and authenticates it with the request. Bob uses T_A to validate that the request is *fresh*, using the bound Δ , and ensuring T_A is larger than previously received T_A values.

Freshness (acceptable delay): Bob validates that the received T_A is within Δ from its own clock $clk_B(\cdot)$ at the time the req is received.

When Alice receives the response $resp$, she similarly confirms it is valid, as follows:

No modification: compare the received MAC value to the MAC computed with the correct inputs.

$resp$ is a response from Bob to Alice: The fact that the input to the MAC begins with $2 + 'A \leftarrow B'$ ensures this is a response (second flow) from Bob to Alice.

$resp$ is a response to request req , and not a replay: the value of T_A increments whenever Alice sends a new request, and therefore Alice sent only one request with this value of T_A ; furthermore, this cannot be a replay of previous response from Bob, since no previous response would use this T_A .

Freshness (acceptable delay): Alice validates that the response is received within at most³ $2 \cdot \Delta_{delay}$.

Note that both Alice and Bob may discard their state (the $s_A.T_A$ and $s_B.T_B$ variables) after some time, which the reader may compute.

Exercise 5.3. Show that Alice and Bob do not need to keep forever the stored values of T_A ; at what time may each of them free up this storage?

³For simplicity, Figure 5.11 specifies validation of 2Δ , which is also fine but a bit unnecessarily lax.

Like the counter-based protocol, we also presented the time-based protocol allowing only one query-response at any given time, and assuming reliable communication. However, it is not too difficult to extend it to allow multiple concurrent requests and responses, and handle unreliable communication.

Exercise 5.4. Extend the protocol of Figure 5.11, to allow Alice to send up to three concurrent requests to Bob; allow for receiving and responding to requests out-of-order.

5.4 Shared-key Key Exchange Protocols

Not all communication follows the request-response pattern, and even when it does, many times we may prefer to secure the communication using a *session/record protocol*, which secures arbitrary interactions, using a shared key, such as the simplified protocol in subsection 5.1.4, or the TLS record protocol in Chapter 7.

In principle, the parties could simply use the same shared secret keys for all sessions between them. However, as already stated in Principle 4, it is desirable to limit the use of secret keys. The goal of shared-key Key Exchange protocols is to setup separate *session keys* $\{k_i^S\}$ for each session i , in such a way that exposure of session key k_i^S will not expose other session keys. In this section we focus on two-party shared-key Key Exchange protocols, which derive all session keys from one fixed key k , to which we refer as the *master key* and/or as the *long-term key*. These protocols should be contrasted from *public-key key exchange* protocols, which we study in subsection 6.1.3, and which are often used to *establish* the shared master key. In Chapter 7 we study the widely-used SSL/TLS protocols, which combine public-key key exchange to establish a shared master key, and shared-key Key Exchange to derive session keys from the master key.

The use of session keys k_i^S securely derived from the master key k has multiple security benefits:

1. By changing the key periodically, we reduce the number of ciphertext messages, encrypted using the same key, available to the cryptanalyst. This usually makes cryptanalysis harder, and possibly infeasible, compared to the use of a fixed key, which allows the adversary to collect plenty of ciphertext messages, which often makes the attack easier, as we have seen in Chapter 2. We also reduce the amount of plaintext exposed if the cryptanalyst succeeds in finding a key, thereby reducing the amortized ‘return on investment’ for cryptanalysis.
2. Keys may also be exposed by hacking attacks; an attacker can use an exposed key until it is changed. By changing session keys periodically, and making sure that each of the session keys remains secret (pseudo-random) even if other session keys are exposed, we limit or reduce the damages due to exposure of some of the keys.

3. The separation between session keys and master key allows some or all security to be preserved even after attacks which expose the entire storage of a device. One way to achieve this is when the master key is confined to a separate *Hardware Security Module (HSM)*, protecting it even when the computer storage - except the internal storage of the HSM - is exposed. We also discuss ways to ensure or restore security following exposure, even without an HSM.
4. Finally, most session/record protocols, including the one in subsection 5.1.4 and the TLS record protocol in Chapter 7, rely on persistent counters kept by the peers; however, counters are often reset, or may otherwise get out of sync.

Key Exchange protocols are an extension to mutually-authenticating protocols, with the same functions, inputs and outputs. There is only one more output of the protocol: the session key k_i^S . This key is provided to the session/record protocol running in each of the peers.

We say that a two-party shared-key Key Exchange protocol *ensures secure key-setup* if it ensures the following two requirements.

Key agreement: if both parties complete successfully, then they *both output the same key k_i^S* .

Key secrecy: each session key k_i^S is secret, or more precisely, pseudo-random, i.e., indistinguishable from a random string of same length, even if the adversary is given all the other session keys. This implies that the master key k must remain pseudorandom, even if all session keys $\{k_i^S\}$ are given to the adversary.

5.4.1 The Key Exchange extension of 2PP

In this subsection we discuss a simple extension to the 2PP protocol, which ensures secure key-setup. This is achieved by outputting the session key k_i^S as:

$$k_i^S = \text{PRF}_k(N_{A,i} \# N_{B,i}) \quad (5.5)$$

In Equation 5.5, $N_{A,i}$ and $N_{B,i}$ are the nonces exchanged in the i^{th} session of the protocol, and k_i^S is the derived i^{th} session key. We use k^M to denote the master (long-term) shared secret key, provided to both parties during initialization. The protocol is illustrated in Figure 5.12.

Since both parties compute the session key k_i^S in the same way from $N_{A,i} \# N_{B,i}$ and the master key k_i^M , it follows that they will receive the same key, i.e., the Key Exchange 2PP extension ensures key agreement. Since the session keys are computed using a pseudorandom function, $k_i^S = \text{PRF}_k(N_{A,i} \# N_{B,i})$, it follows that the key of each session is pseudo-random, even given all other session keys. Namely, *the Key Exchange 2PP extension ensures secure key setup*.

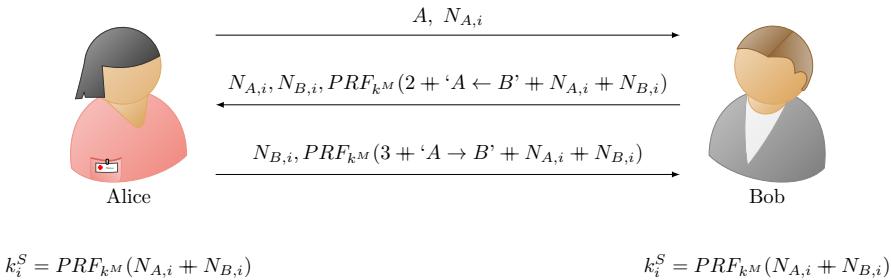


Figure 5.12: The 2PP Key Exchange protocol, shown generating i^{th} session key, k_i^S .

Notice that there is another, seemingly unrelated change between the Mutual Entity Authentication 2PP (Figure 5.6) and the Key Exchange 2PP (Figure 5.12) protocols, namely, the use of PRF instead of MAC to authenticate the messages in the protocol. This change is needed to avoid using the same key in two different cryptographic schemes (MAC and PRF), which could, at least in some ‘absurd’ scenarios, be insecure. The change is also allowed, since every PRF is also a MAC.

5.4.2 Deriving Per-Goal Keys

Following the *key separation principle* (Principle 7), session protocols often use two separate keyed cryptographic functions, one for encryption and one for authentication (MAC); the key used for each of the two goals should be pseudo-random, even given the key to the other goal. We refer to such keys as *per-goal keys*. The next exercise explains how we can use a single shared session key, from the 2PP or another key-setup protocol, to derive multiple per-goal session keys.

Exercise 5.5 (Per-goal keys).

1. Show why it is necessary to use separately pseudorandom keys for encryption and for authentication (MAC), i.e., per-goal keys.
2. Show how to securely derive one session key $k_{E,i}^S$ for encryption and another session key $k_{A,i}^S$ for authentication, both from the same session key k_i^S , yet each key (e.g., $k_{E,i}^S$) is pseudo-random even given the other key (resp., $k_{A,i}^S$). Your solution may use any cryptographic scheme or function that we learned - your choice!

Explain the security of your solutions.

Hints for solutions:

1. See Exercise 4.18.

2. One simple solution is $k_{E,i}^S = \text{PRF}_{k_i^S}('E')$, $k_{A,i}^S = \text{PRF}_{k_i^S}('A')$, where ‘E’, ‘A’ are just two separate inputs to the PRF, ensuring two independently-pseudorandom session keys.

□

To further improve the security of the session/record protocol, we may use *two separate pairs of per-goal keys*, depending on the direction: one pair $(k_{E,i}^{A \rightarrow B}, k_{A,i}^{A \rightarrow B})$ for (encryption, authentication) of messages from Alice to Bob, and another pair $(k_{E,i}^{B \rightarrow A}, k_{A,i}^{B \rightarrow A})$ for (encryption, authentication) of messages from Bob to Alice.

Exercise 5.6. 1. How may the use of separate, pseudo-random pairs of per-goal keys for the two ‘directions’ improve security?

2. Show how to securely derive all four keys (both pairs) from the same session key k_i^S .
3. Show a modification of the Key Exchange 2PP extension, which securely derives all four keys (both pairs) ‘directly’, a bit more efficiently than by deriving them from k_i^S .

Explain the security of your solutions.

5.5 Key Distribution Center Protocols

In this section, we expand a bit beyond our focus on two party protocols, to briefly discuss *three-party* shared-key, *Key Distribution Protocols*. In general, key distribution protocols establish a shared key between two or more entities. We focus on Key Distribution Protocols which use only symmetric cryptography (shared keys), and involve only three parties: Alice, Bob - and a *trusted third party (TTP)*, often referred to as the *Key Distribution Center (KDC)*; the use of the term KDC is so common, that these protocols are often referred to as *KDC protocols*. The goal of the protocol - and of the KDC - is to establish a shared key between the other parties.

There are many types of Key Distribution Protocols. We present two important and very different protocols, both simplified versions of practical protocols: the *Kerberos* protocol [247], which is the most widely-known and widely-used KDC protocol for computer networks, in subsection 5.5.1, and the *GSM* protocol [20], which is the first widely-deployed cellular communication protocol and still supported by essentially all existing mobile devices and networks, in Section 5.6. Due to the unique nature of GSM, we describe it separately, in the next section.

The Kerberos and GSM protocols are very different. They even differ in their assumptions: in Kerberos, the KDC shares a key with each of two parties, while in GSM, the TTP shares a key only with one party, the client (e.g., Alice), and is assumed to have secure connection to the other party (e.g., Bob).

Another important difference is that the Kerberos protocol is secure, while the GSM protocol is notoriously insecure - and will provide good opportunity

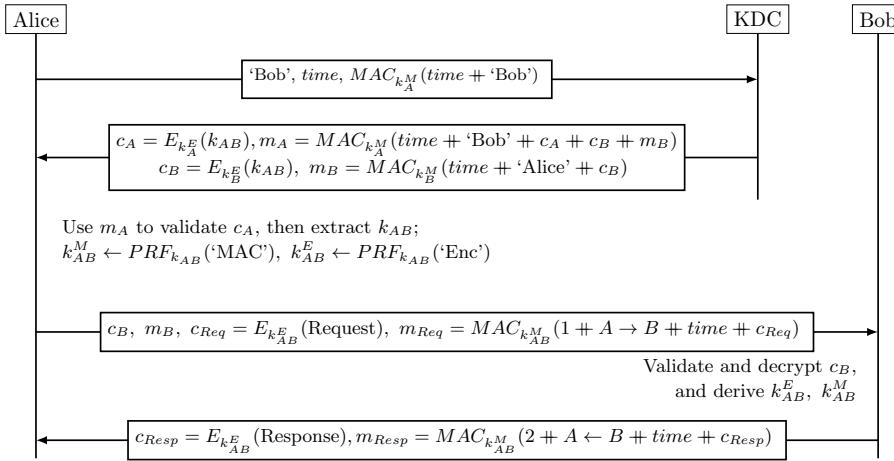


Figure 5.13: The Kerberos Key Distribution Center Protocol (simplified). The KDC shares with Alice k_A^E for encryption and k_A^M for MAC, and with Bob, k_B^E for encryption and k_B^M for MAC. The KDC selects a shared session key k_{AB} to be used by Alice and Bob for the specific session (request-response). Alice and Bob use k_{AB} and a pseudorandom function PRF to derive two shared keys, $k_{AB}^E = PRF_{k_{AB}}(\text{'Enc'})$ (for encryption) and $k_{AB}^M = PRF_{k_{AB}}(\text{'MAC'})$ (for authentication, i.e., MAC). All parties validate contents of MACs before decrypting authenticated ciphertexts.

to introduce important attack techniques. These attacks are practical and well known; there are even *products* that perform these and other attacks on GSM, and on some other cellular-communication protocols.

5.5.1 The Kerberos Key Distribution Protocol

In this subsection we present a simplified version of the Kerberos [247] key-distribution protocol. Kerberos [247] is the most widely known and deployed shared-key system for authorization and authentication in computed networks and distributed systems.

In Kerberos, as in many other KDC protocols, the KDC shares a key with each party: k_A with Alice and k_B with Bob; using these keys, the KDC helps Alice and Bob to share a symmetric key k_{AB} between them. In fact, the term *KDC protocol* is usually used for protocols following this model. The (simplified) protocol is shown in Figure 5.13.

The process essentially consists of two exchanges, both resembling the Time-based Authenticated Request-Response (Figure 5.11).

The first exchange is between Alice and the KDC. In this exchange, the KDC sends to Alice the key k_{AB} that will be shared between Alice and Bob, encrypted (c_A) and authenticated (m_A). In addition, Alice receives the pair c_B, m_B , which also consist of encryption and authentication of k_{AB} , but this

time, using the keys shared between the KDC and Bob. Alice would next relay these to Bob. Notice that these values are also authenticated when sent to Alice (within m_A).

In the second exchange, Alice sends her request to Bob, encrypted and authenticated using k_{AB} . Alice also sends the pair c_B, m_B , which Bob uses to securely retrieve k_{AB} . Alice and Bob both derive, from k_{AB} , the shared encryption and authentication (MAC) keys k_{AB}^E and k_{AB}^M , respectively.

Note that in the above protocol, the KDC never initiates communication, but only *responds* to an incoming request; this communication pattern, where a server machine (in this case, the KDC) only responds to incoming requests, is referred to as *client-server*. Server machines usually use client-server communication, since it relieves the server (e.g., KDC) from the need to maintain state for different clients, except for the long-term keys (e.g., k_A and k_B). This makes it easier to implement an efficient service, especially when clients may access different servers.

In Kerberos, the TTP has an additional role: *access control*. Namely, the TTP controls the ability of the client (Alice) to contact the service (Bob). In this case, the m_B authenticator will also be a *ticket* or *permit* for the use of the server. Access control is an important aspect of computer and network security.

5.6 The GSM Key Exchange Protocol

We next discuss the *GSM Key Distribution and Key Exchange protocol*, an important-yet-vulnerable shared-key Key Exchange protocol. This protocol is performed at the beginning of each connection between a *Mobile* device belonging to a user, e.g., mobile phone, a *Visited Network (VN)*, and the user's *Home Network*. The mobile is only connected via the Visited Network, i.e., any communication between the mobile and the Home Network must be via the Visited Network.

Due to its wide use and importance, there are many publications on GSM; unfortunately, there is no complete agreement on the terms. We try to use terms which are widely used and intuitive, but readers should be ready to see different terms in different publications, e.g., the Visited Network (VN) is often referred to as the *Base station (BS)*, or *Visited Location*; the Home Network is sometimes referred to either as the *Home location* or as the *Authentication Center (AuC)*.

The GSM protocol is based on a shared key k_i associated with mobile user identifiers; this identifier is called the *International Mobile Subscriber Identifier (IMSI)*, but we refer to it simply as i . This key, k_i , is known to the Home Network and to the user's mobile device. More specifically, the mobile device of user with identifier (IMSI) i , has a copy of k_i ; and the Home Network has a mapping from each identifier i of any of its users, to the corresponding k_i .

The Visited Network is not fully trusted by the user and by the Home Network; therefore, is not provided with the shared key k_i , which should remain secret from it. The GSM design assumes secure communication between

the Visited Network and the Home Network; in particular, information sent by the Home Network to the Visited Network is not exposed to any other party. This may be ensured by running TLS (Chapter 7) or another secure-communication protocol between these two parties; however, GSM simply assumes such secure communication, without specifying how it should be secured. Apparently, at least originally, visited and home networks simply often used private communication lines between them, and assumed this is secure enough, although we believe by now they probably all use TLS or a similar protocol.

The basic idea of the GSM Key Exchange protocol is for the Home Network to provide the Visited Network with a *triplet* (r, K_c, s) for every session of the mobile, where:

r (or Rand) is a *random* 128-bits string selected by the Home Network, and used, together with the client's key k_i , to compute (K_c, s) , as: $(K_c, s) \leftarrow A38(k_i, r)$, where $A38$ is an algorithm⁴, which the Home Network operator is free to select, and the GSM specifications requires to be a *One-Way Function*. Since (K_c, s) are derived from k_i and r , it suffices for the Visited Network to send to the Mobile device only r , and the mobile can compute the same values (K_c, s) as computed by the Home Network (which were sent to the Visited Network).

K_c is the *session key*. This key is used by the Visited Network and the client to encrypt the connection between them.

s (or SRES) is a *secret authenticator/result*; the mobile device authenticates itself to the Visited Network by sending to it the same value of s , as the Visited Network received in the triplet (r, K_c, s) from the Home Network.

The GSM Key Exchange protocol, and two messages illustrating how GSM protects data transfer with the session key K_c , are illustrated in Figure 5.14. The design uses two ‘cryptographic functions’, $A38$, introduced above, and $A5$. In the specifications, $A38$ is described as a *One Way Function (OWF)*, and $A5$ is referred to as *encryption*; however, from their use in the protocol, both functions should really be *pseudorandom functions (PRFs)*. The standard defines three variants of $A5$, denoted $A5/v$ for $v \in \{1, 2, 3\}$, and uses $A5/0$ to denote no encryption.

The GSM specification allows some flexibility as to the specific functions. In fact, the operator of the Home Network is free select the $A38$ function. A common choice is an algorithm called COMP128 which was defined by the GSM consortium, and shared under non-disclosure agreement.

The $A5/i$ ‘encryption’ functions, however, must be supported by both mobile and the Visited Network. The GSM specifications included two implementations

⁴Actually the GSM specifications defines two separate functions, $A3$ and $A8$, to compute each parameter: $s \leftarrow A3(k_i, r)$, $K_c \leftarrow A8(k_i, r)$. But since they are always computed together and are expected to have similar cryptographic properties, they are usually considered and implemented by a single algorithm, which we denote $A38$; you may also see it denoted $A3A8$.

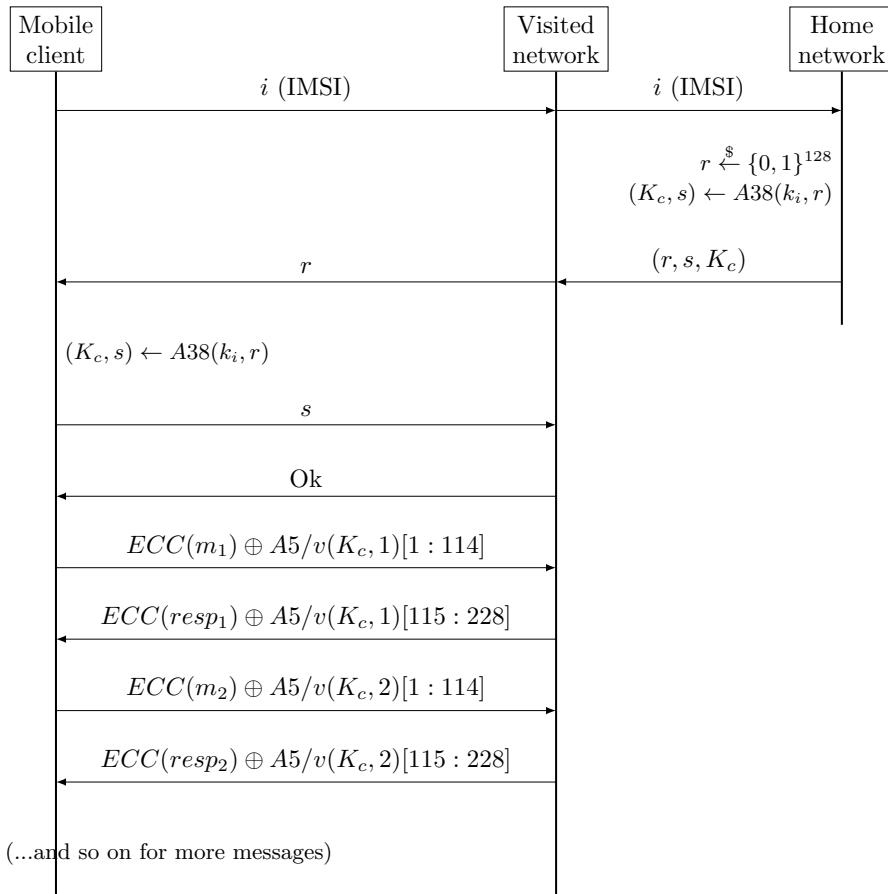


Figure 5.14: The GSM Key Exchange Protocol; the standard defines ‘cryptographic functions’ $A38$ (defined in the specifications as a OWF, but actually used as a PRF) and $A5$ (referred in the specifications as encryption, but actually also used as a PRF). The standard defines three variants of $A5$ denoted $A5/v$ for $v \in \{0, 1, 2, 3\}$, where $A5/0$ denotes no encryption. The GSM standard also specifies the *Error Correction Code* function $ECC(\cdot)$.

of $A5$, denoted $A5/1$ and $A5/2$; the $A5/2$ algorithm is an intentionally-weaker variant of $A5/1$, included to allow export of GSM system for network operators in countries to whom it was, at the time, not allowed to export ‘strong security’ encryption products. Similarly to COMP128, both $A5/1$ and $A5/2$ were shared under non-disclosure agreement, i.e., kept ‘secret’. Later, another option was added - the $A5/3$ algorithm, based on the *KASUMI* block cipher. Another option is $A5/0$, which simply means that encryption is not performed at all.

As can be seen in the bottom (last) messages of Figure 5.14, the $A5/i$ functions outputs 228 bits. Half of them, bits 1 to 114, are used to encrypt

messages from the Mobile client to the Visited Network; the other half, say from bit 115 to bit 228, is used to encrypt ‘responses’, i.e., messages from the Visited Network to the Mobile client.

Note that the input to the $A5/i$ functions is always the key k , and a non-secret number - for simplicity, we show it as a sequential counter⁵. These values are synchronized between Mobile client and Visited Network; this synchronization is due to the underlying communication protocol (*TDMA*).

Overview of the GSM Key Exchange. The Key Exchange begins with the mobile sending its identifier *IMSI* (*International Mobile Subscriber Identity*) to the Visited Network; we denote this as simply i . The Visited Network forwards i to the Home Network.

The Home Network uses i to retrieves the key k_i of the mobile client. Then, the Home Network selects a random 128-bit binary string r , and uses k_i and r to compute: $(K_c, s) \leftarrow A38(k_i, r)$, using the *A38* algorithm (see discussion above). The Home Network sends the resulting *GSM authentication triplet* (r, s, K_c) to the Visited Network.

Figure 5.14 also shows an example of two messages m_1, m_2 sent from Mobile client to the Visited Network, and two corresponding ‘responses’, $resp_1, resp_2$, sent from the Visited Network to the Mobile client. Note that these do not have to be really responses to the messages; the Visited Network would send in the same way any other message to the Mobile client, e.g., from some remote communicating client - we just used ‘resp’ (for ‘response’) since it seems a bit clearer, avoiding confusion with messages from the Mobile client. Of course, in typical real use, the mobile and the Visited Network exchange more than two messages and responses.

All the messages, including responses, that are exchanged between the Mobile client and the Visited host, are encrypted using the connection’s key K_c . The encryption uses the *A5* algorithm chosen by the client, e.g., *A5/1*. More correctly, the messages (and responses) are encrypted by *bitwise-XOR* to the output of the *A5* function, since the *A5* algorithms define a pseudorandom function (PRF), as we explained above.

Error correction then encryption? Before XORing the messages (and responses) with the output of *A5*, the protocol first applies an *Error Correcting Code (ECC)* to the message/response, to allow recovery from bit errors - common in wireless communication. In fact, GSM uses quite extreme error correction codes, e.g., with input of 184 (non-encoded) bits and output of 456 (encoded) bits⁶. Note that since every flow XORs the message/response with only 114 bits from the output of *A5*, this means that every ‘real’ GSM message, would actually be transmitted using multiple of these 114-bit flows; what we show for the messages and responses in Figure 5.14 is only a simplification.

⁵The actual numbers used in GSM are a bit more complex, but still non-secret.

⁶This is a simplification; in reality, different lengths are used for different types of messages.

The use of error-correction *before* encryption may have been designed as a heuristic attempt to provide *authentication as a by-product of encryption*. However, as discussed in subsection 4.7.2, this approach is *not advisable*: it may fail to prevent message modification; it foils the use of authentication and then applying error-detection/correction code, as a way to detect attacks; and, significantly, it may make it *easier to attack the encryption scheme*, as the plaintext will have a lot of redundancy due to the ECC. *This is the case for GSM.*

In fact, the use of ECC-then-Encrypt in GSM may be the best (or worst...) example of the risk in performing ECC and then encrypting. Specifically, see [20] for an effective ‘ciphertext-only’ attack on the GSM encryption using *A5/1* or *A5/2* encryption. While the attacks are presented as ‘ciphertext-only’, they fully exploit the fact that the plaintext has huge redundancy due to the use of ECC before encryption.

While this attack is highly recommended reading, we will not describe it here. Instead, we proceed to discuss *protocol-based* attacks on the GSM Key Exchange.

5.6.1 Vulnerability case study: VN-impersonation Replay attack on GSM

Figure 5.15 shows the simple *VN-impersonation replay attack* against the GSM Key Exchange protocol. The attack involves a *fake Visited network (VN)*, i.e., the attacker is impersonating as a legitimate VN.

The VN-impersonation attack has three phases:

Eavesdrop: in the first phase, the attacker eavesdrops on a legitimate connection between the mobile client and a legitimate Visited Network (VN). The Key Exchange between the client and the Home Network is exactly like in Figure 5.14, except that, for simplicity, Figure 5.15 does not show the Home Network and the messages exchanged between the Visited Network and the Home Network.

Cryptanalysis: in the second phase, the attacker cryptanalyzes the ciphertexts collected during the eavesdrop phase. Assume that the attacker is successful in finding the session key K_c shared between client and Visited Network; this is reasonable, since multiple effective attacks are known on the GSM ciphers *A5/1* and *A5/2*.

Impersonate: finally, once cryptanalysis has exposed the session key K_c , the attacker impersonates as a legitimate Visited Network, and *replays the same random challenge r* sent by the legitimate Visited Network in the eavesdrop phase. Since the connection key K_c is derived in a deterministic way from r , by $(K_c, s) \leftarrow A38(r)$, then the client would reuse the same connection key K_c as in the eavesdropped-to connection - the one exposed by the attacker during the cryptanalysis phase. The attacker now uses K_c to communicate correctly with the client; in particular this allows the

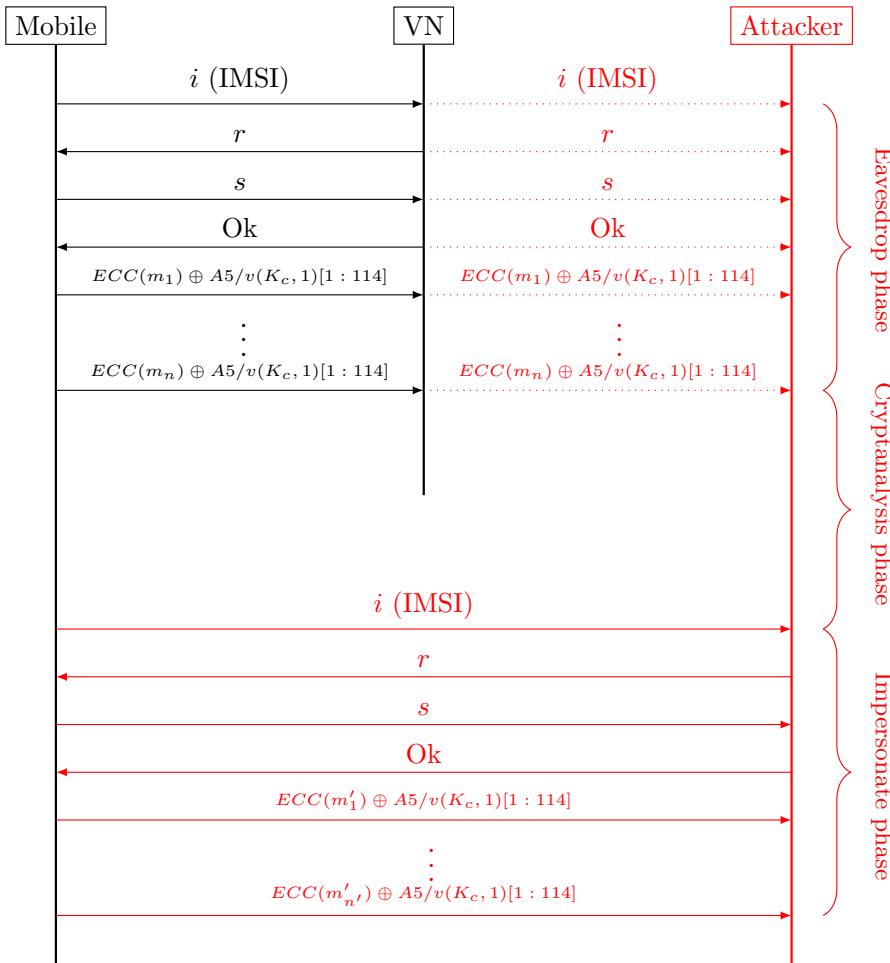


Figure 5.15: The VN-impersonation attack by a MitM attacker on the GSM Key Exchange. The Key Exchange between the client and the Home Network is exactly like in Figure 5.14, but here we omit the Home Network and the messages exchanged between the Visited Network and the Home Network. This figure is simplified, in particular, it does not include the cipher-negotiation details; see these in Figure 5.16. $A5/v$ denotes the GSM encryption scheme; standard values are for $v \in \{0, 1, 2, 3\}$.

adversary to decrypt any messages $m'_1, \dots, m'_{n'}$ encrypted and sent by the client in this new connection.

Are MitM attacks feasible against GSM? After the VN-impersonation attack and the MitM *downgrade attack* (subsection 5.6.3) were published [20], some responses argued that building a MitM adversary is ‘too complex’ and

therefore such attacks are not a real concern. Nevertheless, devices allowing GSM-MitM attacks have been constructed - by academic researchers, students, independent developers - and also companies; such products are available for purchase from multiple vendors.

Is the VN-impersonation attack effective and a real threat? The VN-impersonation allows the attacker to impersonate as a Visited Network and cause the mobile client to send (new) messages encrypted with the (old) key, which the attacker can now decrypt. The attacker may also respond with fake messages, of course, to continue the dialog. However, this attack has one significant drawback: the attacker cannot *impersonate as the client to a legitimate Visited Network*. In particular, the attack does not allow the attacker to decrypt *new responses* sent from a remote peer to the client. In practice, this may also make it hard for the attacker to deploy this attack in some scenarios, e.g., to become a MitM on a complete call between the client and a remote party. The attacker may try to connect to the remote party using a separate call from the attacker's own mobile device, relaying traffic between the client and the remote party via the client's device - but the remote party may notice the use of a different client device. This serious limitation is avoided by the downgrade attack we discuss next.

5.6.2 Crypto-agility and cipher suite negotiation in GSM

In this subsection, we first introduce the important principle of *Crypto-agility* (also known as *cryptographic agility*). *Crypto-agility* means that the cryptographic protocol allows the use of different cryptographic functions and schemes, as long as they satisfy some requirements, e.g., an *IND-CCA* encryption scheme or a secure *PRG*. We refer to a specific choice of functions/schemes used by a protocol as a *cipher suite*. Then, we explain that the GSM protocol supports crypto-agility, including a *cipher suite negotiation* mechanism, allowing entities to negotiate the specific cryptographic functions and schemes to be used. Finally, we show that the GSM ciphersuite negotiation mechanism is vulnerable to a devastating cipher suite *downgrade attack*.

Principle 11 (Crypto-agility). *Cryptographic protocols should be designed using abstract ‘building block’ cryptographic functions and schemes; the set of functions and schemes is called a cipher suite. Each function/scheme should have well-defined requirements. It is desirable for protocols to allow cipher suite negotiation to determine the specific cipher suite to be used in a particular run of the protocol, as long as it is secure. Cipher suite negotiation is secure if the negotiated cipher suite is never inferior to another cipher suite that is supported by both/all parties involved in the protocol.*

Basically, crypto-agility requires a modular design, where the design of the protocol does not depend on the specific components (cipher suite) used, only on their required security properties. This has several important benefits:

1. crypto-agility allows replacing a cryptographic scheme/function which is found or suspected to be vulnerable, while continuing to use the same protocol.
2. crypto-agility allows different users to use the same protocol, but using different schemes, due to different trust assumptions, different efficiency/security tradeoffs and considerations, or other reasons, such as licensing, availability and legal (often export) restrictions. In particular, some countries restrict the export of some cryptographic mechanisms; specifically, until about 2000, the USA restricted export of cryptographic systems using symmetric encryption with keys longer than 40 bits. A protocol may further support a secure *cipher suite negotiation* to allow the parties to choose the ‘best’ cipher suite supported by both/all of them.
3. The security of a crypto-agile protocol can be established based on the well-defined requirements from the cipher suite, which makes it easier to design, evaluate, understand and prove security of the protocol.

Note, however, that all too often, protocol designers focus on crypto-agility and cipher suite negotiation - but fail to properly ensure the security of the negotiation mechanism, creating serious vulnerabilities and allowing different *downgrade attacks*, which allow an attacker to trick the parties into using a particular, vulnerable cipher suite chosen by the attacker. These attacks usually involve a Man-in-the-Middle (MitM) attacker. In this section, we will see downgrade attacks against GSM; later, in Chapter 7, we will see several downgrade attacks on SSL/TLS.

GSM: crypto-agility with cipher suite negotiation vulnerable to downgrade attack. GSM supports crypto-agility, in the sense that the protocol is defined for any stream cipher, with three specific options (*A5/1*, *A5/2* and *A5/3*) as well as the ability to use other stream ciphers. Furthermore, GSM supports cipher suite negotiation, since the visited Network and the client (mobile) negotiate which stream-cipher to use. Namely, the client sends the list of supported stream ciphers, and the visited network indicates which of them it prefers; this stream cipher is then used by GSM. In particular, GSM’s cipher suite negotiation was essential to allow interoperability between a client / visited network that support only an exportable (cryptographically weak) stream cipher, and a visited network / client that support both the exportable stream cipher and a more secure stream cipher.

However, the *GSM cipher suite negotiation is not well protected*, allowing a *downgrade attacks* by a MitM adversary, as we show next. In fact, we show two attacks. First, in this subsection, we outline the simple *downgrade to A5/1 attack*, that allows downgrading GSM connections to use *A5/1*. Then, in the subsection subsection 5.6.3, we present the (more advanced) *downgrade to A5/2 attack*. Both attacks work even when both mobile client and visited network support and prefer a stronger cipher, e.g., *A5/3*.

First, let us explain the GSM ciphersuite negotiation mechanism.

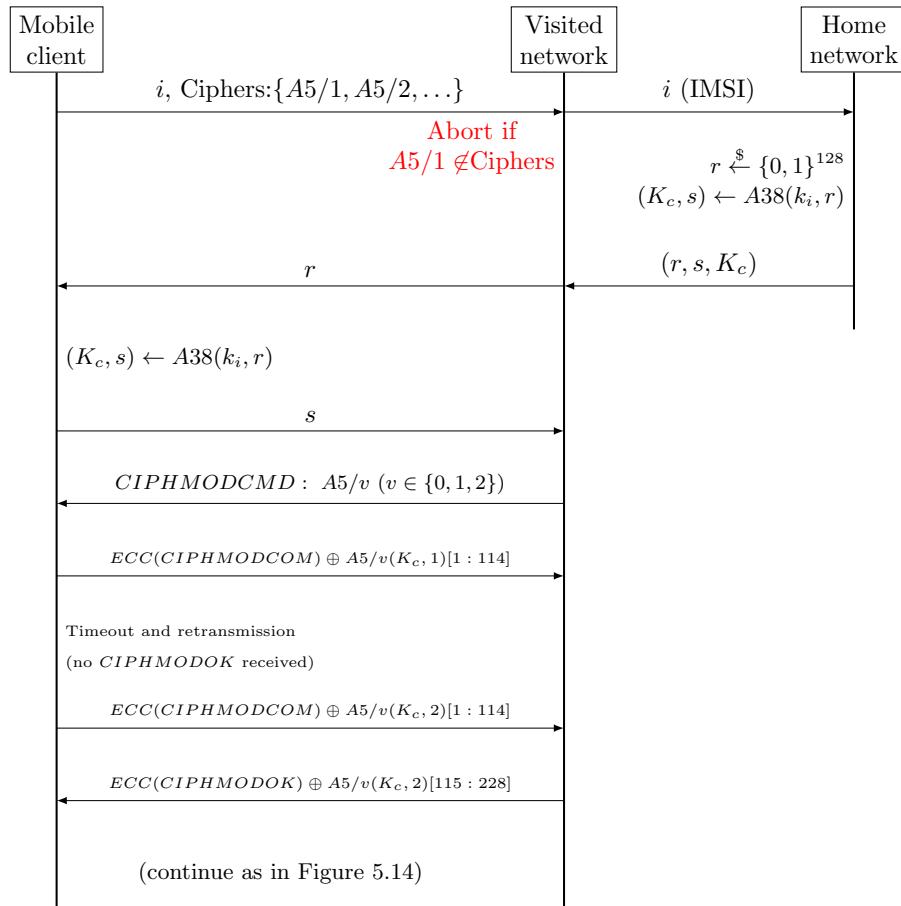


Figure 5.16: The GSM Key Exchange Protocol, including details of cipher suite negotiation (omitted in Figure 5.14). Note that the visited network aborts, if the ciphers offered by the client do not include $A5/1$.

The GSM ciphersuite negotiation. The GSM ciphersuite negotiation process is shown in Figure 5.16. In the first message of the Key Exchange, containing the mobile client's identity (IMSI). The Mobile client also lists the supported ciphers, i.e., the $A5/v$ functions. For example, in the figure, the Mobile supports $A5/1$ and $A5/2$. The Visited Network selects the stream cipher to be used from this list. Usually, the Visited Network would select the stream cipher considered most secure that this Visited Network supports, among those offered by the mobile client.

A critical property of the GSM negotiation mechanism is that *all clients support the stronger $A5/1$ protocol*. Furthermore, GSM specifies that *Visited networks that support $A5/1$, as most do, should refuse to use $A5/2$ - even if $A5/2$ is the only option on the list*. This is an important fact which has significant

impact on GSM downgrade attacks:

Fact 5.2 (GSM Visited Networks refuse to downgrade to A5/2). *All GSM Visited networks support A5/1, and refuse to open (abort) a connection if the cipher suites offered by the mobile client do not include A5/1.*

The reason for Fact 5.2 is that the A5/2 cipher is known to have been designed intentionally to provide vulnerable encryption. This vulnerable cipher was necessary to gain government permission to export GSM equipment; GSM network operator equipment was allowed for export to certain countries, only if restricted to use only A5/2.

Without the defense of Fact 5.2, a MitM attacker could have simply removed A5/1 (and any other ‘strong’ cipher) from the list of ciphers sent by the Mobile, and as a result, communication would have used the vulnerable A5/2. Unfortunately, Fact 5.2 does not prevent the simple *downgrade to A5/1 attack*, which we present below. In the next subsection, we present the (more advanced) *downgrade to A5/2 attack*.

Downgrade to A5/1 attack. Fact 5.2 is the only defense of GSM against downgrade attacks; yet, it does not refer at all to *other* ciphers, except A5/1. As a result, GSM is vulnerable to a simple *downgrade to A5/1 attack*.

For example, if a mobile supports $\{A5/1, A5/2, A5/3\}$, then a MitM attacker can simply remove the A5/3 option from the list sent by the Mobile client, to cause the Visited Network and Mobile client to use the (weaker) A5/1 protocol. This attack is much simpler than the ones we present next, in subsection 5.6.3; therefore, we leave it as an exercise for the reader (Exercise 5.13). This exercise - finding the simple downgrade attack - should not be difficult, especially after learning the more advanced *downgrade to A5/2 attack* which we describe next.

5.6.3 The downgrade to A5/2 attack on GSM

In this subsection, we present the *downgrade to A5/2 attack* on GSM. This is a devastating attack, since A5/2 is an absurdly vulnerable algorithm; indeed, it was intentionally designed that way, since such weakened protocol was necessary to obtain permission to export GSM devices. This is a non-trivial attack, and therefore, we first present a simplified variant that is unlikely to work in practice. Then, we present the ‘real’ downgrade to A5/2 attack.

Both attacks are based on the GSM key-reuse vulnerability, which we describe next.

GSM key-reuse vulnerability. GSM has an *unusual additional vulnerability*, making downgrade attacks much worse than with most systems/protocols. This vulnerability is due to the following fact:

Fact 5.3 (GSM Key-Reuse vulnerability). *The GSM Key Exchange establishes the same key K_c , regardless of the cipher used (e.g., A5/1, A5/2 or A5/3). Namely, GSM protocol uses the same key K_c for all ciphers.*

Note that this vulnerability is due to the fact that the GSM designers completely ignored the *Key separation* principle (Principle 10).

Fact 5.3 allows an attacker to find a key used with one (weak) scheme, and use it to decipher communication protected with a different (stronger) cipher. This is indeed deployed by the GSM downgrade attacks we present. Another fact used by the attack is that A5/2 is absurdly vulnerable, i.e., very effective and efficient attacks against A5/2 are known.

Fact 5.4 (CTO attack on A5/2 requires 900 ciphertext bits and 1 seconds). *The ciphertext-only (CTO) attack of [20] finds the connection encryption key K_c , given 900 bits or more of ciphertext, encrypting ECC-encoded messages; the attack takes less than one second (using standard computing capabilities).*

The following few steps of the Key Exchange as described in Figure 5.16, are exactly as in Figure 5.14. In fact, since the interactions with the Home Network are not impacted or changed by the ciphersuite mechanism or by the attacks, we do not even include them in the discussion and figures of the downgrade attack and its variants.

After receiving the (correct) authenticator, s , from the mobile, the visited network identifies its choice of A5 cipher to the mobile. This is done in the message *CIPHMODCMD : A5/v*, where v indicates the cipher to be used, i.e., in this case, $v \in \{0, 1, 2\}$; this is instead of merely sending ‘Ok’ as in Figure 5.14.

The following message from the client, *and the first encrypted*, is the special message *CIPHMODCOM*, i.e., ‘cipher mode complete’, which acknowledges that the Mobile is using the cipher mode indicated in the *CIPHMODCMD* (‘cipher mode command’) sent by the Visited Network.

Recall that GSM is designed for wireless communication, with significant probability for noise and corruption of the transmitted information - which is the motivation for its extensive use of Error Correcting Code, with extensive redundancy. In spite of that, messages may get lost. Hence, important control messages should be acknowledged; in particular, the Mobile client waits for an acknowledgement message from the Visited Network (VN), which we denote *CIPHMODOK*, to know that the VN received correctly the *CIPHMODCOM* message.

When the Mobile times-out, i.e., does not receive *CIPHMODOK* in time, then the Mobile retransmits the *CIPHMODCOM* to the Visited Network; this scenario is shown in Figure 5.16 (where we show the case where the first retransmission is successful). This happens after a very short time-out, of much less than a second - few milliseconds. As shown in Figure 5.16, each retransmission uses a distinct sequence number (e.g., 1 and 2, in the figure). This, again, is an important fact for the downgrade attack.

Fact 5.5 (*CIPHMODCOM* message and its retransmission). *After a Mobile receives the CIPHMODCMD (‘cipher mode command’) message, instructing the Mobile to use a specific cipher mode, then the Mobile encrypts and sends CIPHMODCOM (‘cipher mode completed’) to the VN; this message contains 456 bits (including the ECC). The Mobile then waits for an acknowledgement -*

a *CIPHMODOK* ('cipher mode Ok') message from the VN. If this isn't received after time-out of few milliseconds, the Mobile re-transmits, with a new counter value.

A failure may also occur in the reverse direction, i.e., the Visited Network (VN) may time-out while waiting for the *CIPHMODCOM* ('cipher mode completed') from the client. In this case, the VN *aborts* the Key Exchange. Again, this happens after a timeout of few milliseconds. Similarly, a failure may occur also in the earlier phases of the Key Exchange, i.e., between sending i and receiving r , or between sending r and receiving s . However, GSM allows for much larger delays in these early phases - up to 12 seconds! This fact is also significant.

Fact 5.6 (GSM Key Exchange allows 12 second delays till *CIPHMODCMD*). *The Mobile and the VN abort the Key Exchange if they do not receive the expected responses, after timeout of about 12 seconds; this holds for the messages until the CIPHMODCMD is sent (and received), from which point, responses are expected to be almost instantaneous (few milliseconds). If responses are not accepted by the timeout, the Key Exchange is aborted.*

Simplified, unrealistic downgrade attack. We first present a simplified, unrealistic downgrade attack against GSM in Figure 5.17. In this attack, the client supports A5/1 and A5/2, but the MitM attacker 'removes' A5/1 and only offers A5/2 to the Visited Network. As a result, the entire session between Visited Network and client is only protected using the (extremely vulnerable) A5/2 cipher.

However, the attack of Figure 5.17 fails, for the following reasons:

1. As per Fact 5.6, the Visited Network would time-out when it does not receive the *CIPHMODCOM* message within few milliseconds - while the fastest ciphertext-only cryptanalysis process of A5/2, from [20], takes about a second (Fact 5.4).
2. The length of the *CIPHMODCOM* message is only 456 bits (including the ECC), while the cryptanalysis attack of [20] requires 900 bits. Hence, the attack will not succeed to find the key at all.

We next present the 'real' GSM downgrade attack, which overcomes these challenges.

The 'real' downgrade attack on GSM Key Exchange. In Figure 5.18, we finally present the 'real' downgrade attack on the GSM Key Exchange. This attack addresses the two challenges presented above, by:

1. As per Fact 5.6, GSM allows delays of about 12 seconds *until* the Visited Network sends *CIPHMODCMD*. So this attack delays the s response from the client - this gives the MitM attacker 12 seconds, much more than

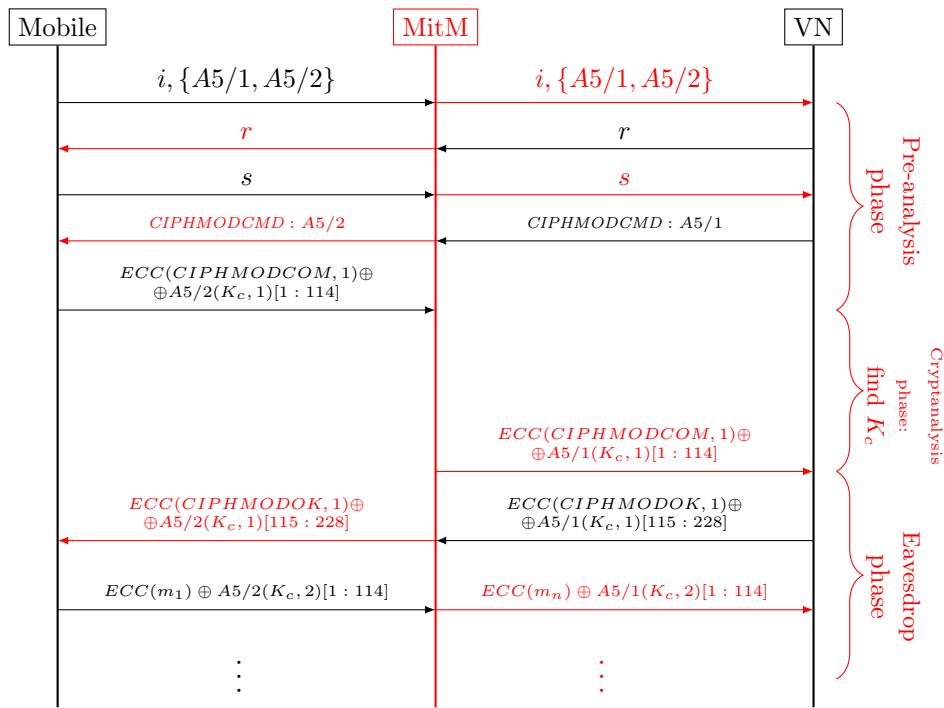


Figure 5.17: Simplified downgrade attack on GSM Key Exchange.

enough for the ciphertext-only cryptanalysis process of $A5/2$ from [20], which takes only about a second (Fact 5.4)!

2. To obtain a sufficient number of ciphertext bits, this attack *intentionally causes* the mobile client to time-out while waiting for the $CIPHMODOK$ message, resulting in rapid retransmission of the $CIPHMODCOM$ message from the client - *re-encrypted*, since the counter value is *modified*, as follows from Fact 5.5. Each of these retransmissions contains 456 bits, providing together more than the 900 bits required for the attack of $A5/2$ from [20] (Fact 5.4).

Several additional variants of this attack are possible; see, for example, the following exercise.

Exercise 5.7 (GSM combined replay and downgrade attack). Consider an attacker who eavesdrops and records the entire communication between mobile and Visited Network during a connection which is encrypted using a ‘strong’ cipher, say $A5/3$. Present a sequence diagram, like Figure 5.15, showing a ‘combined replay and downgrade attack’, allowing this attacker to decrypt all of that ciphertext communication by later impersonating as a Visited Network, and performing a downgrade attack.

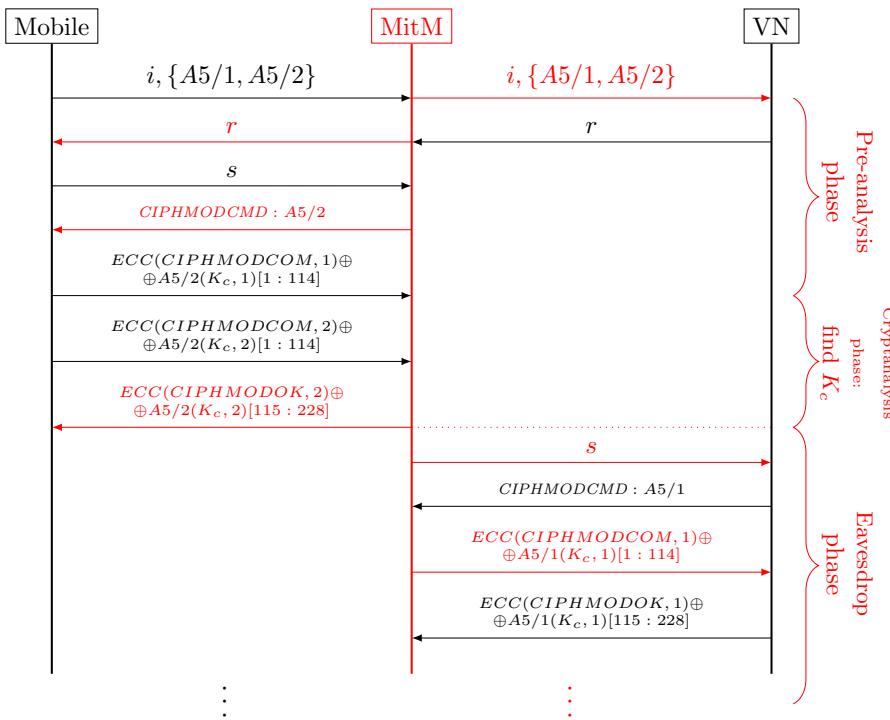


Figure 5.18: A ‘real’ downgrade attack on GSM Key Exchange.

Hint: the attacker will resend the value of r from the eavesdropped-upon communication (encrypted using a ‘strong’ cipher) to cause the mobile to re-use the same key - but with a weak cipher, allowing the attacker to expose the key. \square

Protecting GSM against downgrade attacks. Downgrade attacks involve modification of information sent by the parties - specifically, the possible and/or chosen ciphers. Hence, the standard method to defend against downgrade attacks is to *authenticate* the exchange, or at least, the ciphersuite-related indicators.

Note that this requires the parties to agree on the authentication mechanism, typically, a MAC scheme. It may be desirable to also negotiate the authentication mechanism. In such case, the negotiation should be bounded to reasonable time, and the use of the authentication scheme and key limited to a few messages, to foil downgrade attacks on the authentication mechanism. Every authentication mechanism supported should be secure against this (weak) attack.

It is also necessary to avoid the use of the same key for different encryption schemes, as done in GSM (Fact 5.3), and exploited, e.g., by the attacks of Figure 5.18 and Exercise 5.7. Using separate keys is quite easy, and does not

require any significant resources - it seems that there was no real justification for this design choice in GSM, except for the fact that this allows the Home Network to send just one key K_c , without knowing which cipher would be selected by the mobile and Visited Network.

Deployed defense. The ‘real’ attack is not ‘real’ anymore - but it is prevented in a rather ‘crude’ way: the GSM consortium abolished support for the insecure A5/2. Note that downgrading between other versions, e.g., from A5/3 to A5/1, is still possible.

Learning from GSM Key Exchange vulnerabilities. We have discussed several significant failures of the GSM Key Exchange: the MitM downgrade attack of this subsection, the VN-impersonation attack of Figure 5.15, the use of ‘ECC-then-encrypt’ which allows a ciphertext-only attack on A5/1 or A5/2 [20]. There are more, e.g., efficient known-plaintext attacks. What are the root causes of these vulnerabilities and what can we learn, to avoid such vulnerabilities?

We believe that many of these problems are due to the fact the designers violated several basic principles. Most notably, the GSM design violates the Kerckhoffs’s principle (Principle 2): it relies on the use of ‘secret’ algorithms such as A38, A5/1 and A5/2. The GSM design also did not undergo careful public security analysis, and in particular, its attack model was never clearly stated, violating Principle 1, *clear attack model*; and its design did not carefully apply well-defined, standard cryptographic building blocks, violating Principle 3, *conservative design*, and Principle 8, *cryptographic building blocks*.

5.7 Resiliency to key exposure: forward secrecy and recover security

One of the goals of deriving pseudorandom keys for each session was to reduce the damage due to exposure of one or some of the session keys. A natural question is, *can we improve the resiliency to exposure?* In particular, can a Key Exchange protocol provide some security, even when an adversary may sometimes expose also the master key, or, more generally, *the entire state* of the parties? Notice that with all Key Exchange protocols we studied, exposure of the master key, at any time, allows an adversary to easily expose *all* (past and future) session keys.

One approach to this problem was already mentioned: place the master key κ within a *Hardware Security Module (HSM)*, so that it is assumed *not* to be part of the state exposed to the attacker. However, often, the use of an HSM is not a realistic, viable option. Furthermore, cryptographic keys may be exposed even when using an HSM - by cryptanalysis or by some weakness of the HSM, such as side-channels allowing (immediate or gradual/partial) exposure of keys.

In this section, we discuss a different approach to provide security with resiliency to key exposures. This approach is to design the Key Exchange

protocol to ensure *some* security, even after the adversary obtains the master key (or the contents of the entire storage). We mostly focus on two notions of resiliency to key exposure: *forward secrecy* and *recover security*. We explain these two notions and present Key Exchange protocols satisfying them. Both of these notions can be achieved using shared-key only.

We also briefly discuss additional, even stronger notions of resiliency to key exposures, mainly, an extension for each of the two notions: *perfect forward secrecy (PFS)* and *perfect recover security (PRS)*. For these stronger notions of resiliency, it seems necessary to use *public key cryptography*, which we introduce in Chapter 6. We present PFS and PRS protocols in Section 6.3; later, in Chapter 7, we discuss how PFS is provided by the TLS protocol.

Terms for resilient security. Note that the notions of *Recover Security* and *Perfect Recover Security* are not widely used in the literature. Also, the term *Forward Secrecy* is not always used as we define it; e.g., often it is used to refer to the notion commonly (and here) referred to as *Perfect Forward Secrecy*.

5.7.1 Forward Secrecy 2PP Key Exchange

We use the term *forward secrecy* to refer to Key Exchange protocols where exposure of the entire storage of the communicating party in some future time period, including every (master and session) key kept at that future time, would not expose the keys used in previous time periods, or the plaintext encrypted (and sent) during these previous periods. This should hold although all previous communication could have been intercepted and recorded by the attacker.

To ensure forward secrecy, each period i would use a *separate master key* k_i^M . For simplicity, we will map sessions to time periods, i.e., run the Key Exchange protocol once at the beginning of every period. At the beginning of period/session i , we must erase any previous master key (e.g., k_{i-1}^M). Definition follows. Note that some authors refer to this notion as *weak forward secrecy*, to emphasize the distinction from the stronger notion of *perfect forward secrecy* (which we present later).

Definition 5.4 (Key Exchange with Forward Secrecy). *A Key Exchange protocol \mathcal{P} ensures forward secrecy if once session i terminated, exposure of the state of the entity will not compromise the confidentiality of information sent by the entity or sent to the entity in session i .*

We next discuss *forward secrecy 2PP Key Exchange*, a forward-secrecy variant of the Key Exchange 2PP extension, which we discussed and presented earlier, in subsection 5.4.1. The difference is that instead of using a single master key k , received during initialization, the forward-secrecy Key Exchange uses a *sequence of master keys* k_0^M, k_1^M, \dots ; for simplicity, assume that each master key k_i^M is used only for the i^{th} Key Exchange, with k_0^M received during initialization.

The key to achieving the forward secrecy property is to allow easy derivation of the future master keys k_{i+1}^M, \dots from the current master key k_i^M , but

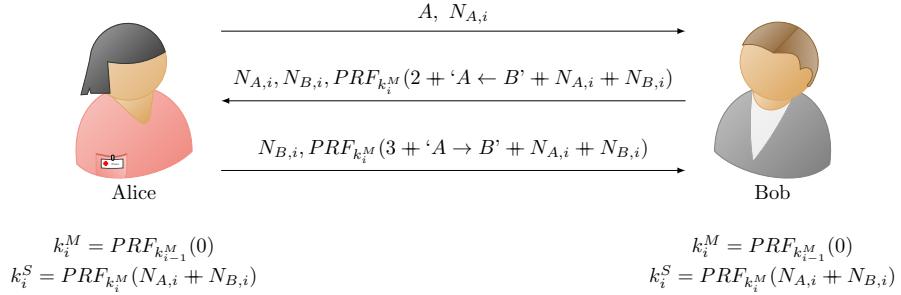


Figure 5.19: The Forward-Secrecy 2PP Key Exchange protocol. This protocol is similar to the 2PP Key Exchange protocol (Figure 5.12). The main difference is that this protocol uses a different master key k_i^M for each period i ; the initial master key, shared by the two parties, is k_0^M .

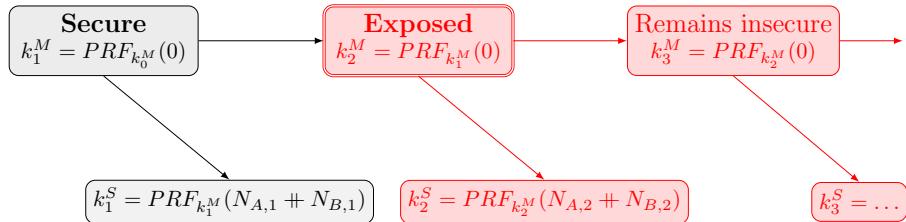


Figure 5.20: Result of running the Forward-Secrecy 2PP Key Exchange for three periods, with the keys exposed in the second period. Periods prior to the exposure (in this example, only the first period) remain secure even after the period where keys are exposed. Periods from the exposure onward are insecure.

prevent the reverse, i.e., maintain the previous master keys $k_{i-1}^M, k_{i-2}^M, \dots, k_0^M$ pseudorandom, even for an adversary who knows k_i^M, k_{i+1}^M, \dots . A simple way to achieve this is by using a PRF, namely:

$$k_i^M = \text{PRF}_{k_{i-1}^M}(0) \quad (5.6)$$

The session key k_i^S for the i^{th} session can be derived using the corresponding minor change to Equation 5.5, namely:

$$k_i^S = \text{PRF}_{k_i^M}(N_A \parallel N_B) \quad (5.7)$$

The resulting Forward-secrecy 2PP Key Exchange protocol is illustrated in Figure 5.19. The use of N_A and N_B in Eq. (5.7) is not really necessary, since each master key is used only for a single Key Exchange.

The Forward-Secure 2PP Key Exchange protocol ensures that the communication in any period that completed before any key exposure, remains secure regardless of key exposures in later periods. See Figure 5.20.

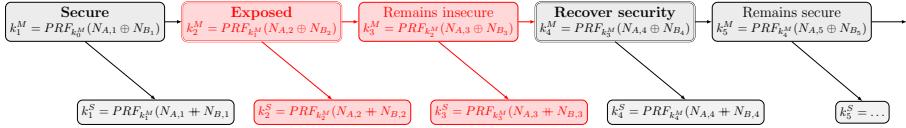


Figure 5.21: Example of running the *recover-security Key Exchange* protocol for five periods, with the keys exposed in the second period, and no attack (even eavesdropping) in the fourth period, allowing recovery of security. Periods prior to the exposure (in this example, only the first period) remain secure even after the period where keys are exposed. Periods after exposure remain insecure, until a ‘recovery period’ (in this example, period 4) where there is no attack. Following recovery period, security is maintained till next exposure.

5.7.2 Recover-Security Key Exchange Protocol

We use the term *recover security* to refer to key setup protocols where a single session *without eavesdropping or other attacks*, suffices to recover security from previous key exposures. Definition follows.

Definition 5.5 (Recover security Key Exchange). *A Key Exchange protocol recovers security if session i is secure, when either of the following holds:*

No attack: *during session i , there was no exposure, and all messages were delivered correctly, without eavesdropping, injection or modification.*

Preserve security: *during session i there was no exposure, and the previous session ($i - 1$) was secure.*

The forward-secure 2PP Key Exchange protocol (Figure 5.19) ensures *forward secrecy* - but not *recover security*. This is since the attacker can use one exposed master key, say k_j^M , to derive all the following master keys, including k_i^M for $i > j$, using Equation 5.6; in particular, $k_{j+1}^M = \text{PRF}_{k_j^M}(0)$.

However, a simple extension suffices to ensure recover security, *as well as* forward secrecy. The extension is simply to use the random values exchanged in each session, i.e., $N_{A,i}, N_{B,i}$, in the derivation of the next master key, i.e.:

$$k_i^M = \text{PRF}_{k_{i-1}^M}(N_{A,i} \oplus N_{B,i}) \quad (5.8)$$

We call this protocol the *recover-security Key Exchange protocol*, and illustrate its operation in Figure 5.21.

By computing the new master key using these three values, it is secret as long as at least *one* of these three values is secret. Since the recover security requirement assumes at least one session where the attacker does not eavesdrop or otherwise interfere with the communication, then both $N_{A,i}$ and $N_{B,i}$ are secret, hence the new master key k_i^M is secret. Indeed, we could have used just one of $N_{A,i}$ and $N_{B,i}$; by XOR-ing with both of them, we ensure secrecy of the master key even if the attacker is able to capture one of the two flows, i.e., even stronger security.

Note that the *recover-security Key Exchange* protocol requires the parties to have a source of randomness which is secure - even if the party is broken-into (and keys exposed). In reality, many systems only rely on pseudo-random generators (PRGs), whose future values are computable using a past value. In such case, it becomes critical to use also the input from the peer ($N_{A,i}$ or $N_{B,i}$), and these values should be also used to re-initialize the PRG, so that new nonces ($N_{A,i}, N_{B,i}$) are pseudorandom and not predictable.

5.7.3 Stronger notions of resiliency to key exposure

Forward secrecy and recover security significantly improve the resiliency against key exposure. There are additional and even stronger notions of resiliency to key exposure, which are provided by more advanced Key Exchange protocols; we only cover a few of these in this textbook - specifically, the ones in Table 5.2.

All known protocols that achieve more advanced notions of resiliency use public key cryptology, and in particular, key-exchange protocols such as the Diffie-Hellman (DH) protocol. Indeed, it seems plausible that public-key cryptography is necessary for many of these notions. This includes the important notions of *Perfect Forward Secrecy (PFS)* and *Perfect Recover Security (PRS)*, which, as the names imply, are stronger variants of forward and recover security, respectively. We now briefly discuss PFS and PRS, to understand their advantages and why they require more than the protocols we have seen in this chapter; we discuss these notions further, with implementations, in Section 6.3.

Perfect Forward Secrecy (PFS). PFS, like Forward Secrecy, also requires resiliency to exposures of state, including keys, occurring in the future. However, on top of that, PFS also requires *resiliency to exposure of the previous state*, again including keys, as long as this exposure occurs only *after* the session ends. We next define this notion. PFS was apparently first coined by Gunther [152]; unfortunately, the term is not always used with a consistent meaning and definition, but the following definition seems to capture the meaning usually used by experts.

Definition 5.6 (Perfect Forward Secrecy (PFS)). *A Key Exchange protocol \mathcal{P} ensures perfect forward secrecy (PFS) if data sent during session i is confidential (indistinguishable), provided that either (1) there is not MitM attack during session i , or (2) the master key of session i and of any previous session, is not given to the adversary - or given only after session i .*

We discuss some PFS Key Exchange protocols in the next chapter, which deals with asymmetric cryptography (also called public-key cryptography, PKC). All known PFS protocols are based on PKC.

Exercise 5.8 (Forward Secrecy vs. Perfect Forward Secrecy (PFS)). *Present a sequence diagram, showing that the forward-secrecy 2PP Key Exchange protocol presented in subsection 5.7.1, does not ensure Perfect Forward Secrecy (PFS).*

<i>Notion</i>	<i>Session i is secure, when:</i>	Crypto
Secure key-setup	Attacker is given <i>session</i> keys of other sessions, but <i>master</i> key is never exposed.	Shared key
Forward Secrecy (FS)	Attacker is given <i>all</i> keys, but only of sessions <i>after</i> session i .	Shared key
Perfect Forward Secrecy (PFS)	Attacker is given all keys of sessions <i>except i</i> , but only <i>after</i> session i ends.	Public key
Recover Security (RS)	Attacker is given keys of other sessions, but session $i - 1$ is secure (or <i>no attack during session i</i>).	Shared key
Perfect Recover Security (PRS)	Attacker is given keys of other sessions, but <i>either</i> session $i - 1$ is secure, or <i>only eavesdropping</i> in session i .	Public key

Table 5.2: Notions of resiliency to key exposures of key-setup Key Exchange protocols. See implementations of forward and recover security in subsection 5.7.1 and subsection 5.7.2 respectively, and for the corresponding ‘perfect’ notions (PFS and PRS) in subsection 6.3.1 and subsection 6.3.2, respectively.

Perfect Recover Security (PRS). We introduce the term *perfect recover security* to refer to Key Exchange protocols where a single session without exposure or *MitM* attacks suffices to recover security from previous key exposures. Definition follows.

Definition 5.7 (Perfect Recover Security (PRS) Key Exchange). *A Key Exchange protocol ensures Perfect Recover Security (PRS), if security (confidentiality and authentication) is ensured for messages exchanged during session i , provided that there is no exposure during session i and either (1) session $i - 1$ is secure, or (2) there is no MitM attack during session i (session i is a recovery session).*

Note the similarity to PFS, in allowing only eavesdropping during the ‘recovery’ session i . Similarly to PFS, we also discuss some PRS Key Exchange protocols in the next chapter, which deals with asymmetric cryptography. Known PRS protocols are all based on asymmetric cryptography.

Comparison of the four notions of resiliency. We compare the four notions of resiliency (forward secrecy, PFS, recover security and PRS) in Table 5.2, along with ‘regular’ secure Key Exchange protocols. We also present the relationships between the five notions in Figure 5.22.



Figure 5.22: Relations between notions of resiliency to key exposures. An arrow from notion A to notion B indicates that notion A implies notion B . For example, a protocol that ensures Perfect Forward Secrecy (PFS) also ensures Forward Secrecy.

Additional notions of resiliency. The research in cryptographic protocols includes additional notions of resiliency to key and state exposures, which we do not cover in this textbook. These include *threshold security* [97], which ensures that the entire system remains secure even if (up to some threshold) of its modules are exposed or corrupted, *proactive security* [74], which deals with recovery of security of some modules after exposures, and *leakage-resiliency* [116], which ensures resiliency to gradual leakage of parts of the storage.

5.8 Additional Exercises

Exercise 5.9 (Attack against SNA with ‘fixed roles’). *Show that the SNA handshake protocol does not ensure concurrent mutual authentication, also for a scenario where each party is only willing to act in one role, i.e., either as an initiator or as a responder, but not as both.*

Hint: as in Figure 5.5, the attack will involve two sessions; but you are not required that both sessions will terminate correctly - one of them may fail. \square

Exercise 5.10. *Some applications require only one party (e.g., a door) to authenticate the other party (e.g., Alice); this allows a somewhat simpler protocol. We describe in the two items below two proposed protocols for this task (one in each item), both using a key k shared between the door and Alice, and a secure symmetric-key encryption scheme (E, D) . Analyze the security of the two protocols.*

1. *The door selects a random string (nonce) n and sends $E_k(n)$ to Alice; Alice decrypts it and sends back n .*
2. *The door selects and sends n ; Alice computes and sends back $E_k(n)$.*

Repeat the question, when E is a block cipher rather than an encryption scheme.

Exercise 5.11. *Consider the following mutual-authentication protocol, using shared key k and a (secure) block cipher (E, D) :*

1. *Alice sends N_A to Bob.*
2. *Bob replies with $N_B, E_k(N_A)$.*
3. *Alice completes the handshake by sending $E_k(N_B \oplus E_k(N_A))$.*

Show an attack against this protocol, and identify the design principles which were violated by the protocol, and which, if followed, should have prevented such attacks.

Exercise 5.12 (GSM). *In this exercise we study some of the weaknesses of the GSM handshake protocol, as described in Section 5.6. In this exercise we ignore the existence of multiple types of encryption and their choice (‘ciphersuite’).*

1. *In this exercise, and in usual, we ignore the fact that the functions A8, A3 and the ciphers E_i were kept secret; explain why.*
2. *Present functions A3, A8 such that the protocol is insecure when using them, against an eavesdropping-only adversary.*
3. *Present functions A3, A8 that ensure security against MitM adversary, assuming E is a secure encryption. Prove (or at least argue) for security. (Here and later, you may assume a given secure PRF function, f .)*

4. To refer to the triplet of a specific connection, say the j^{th} connection, we use the notation: $(r(j), sres(j), k(j))$. Assume that during connection j' attacker received key $k(\hat{j})$ of previous connection $\hat{j} < j'$. Show how a MitM attacker can use this to expose, not only messages sent during connection \hat{j} , but also messages sent in future connections (after j') of this mobile.
5. Present a possible fix to the protocol, as simple and efficient as possible, to prevent exposure of messages sent in future connections (after j'). The fix should only involve changes to the mobile and the Visited Network, not to the home.

Exercise 5.13 (Downgrade to A5/1 attack on GSM). Consider a mobile client and a visited network that both support A5/3 (or some other strong stream cipher). Present a sequence diagram showing how a MitM attacker can cause them to use the (weaker) A5/1 protocol.

Exercise 5.14. Fig. 5.23 illustrates a simplification of the SSL/TLS session-security protocol; this simplification uses a fixed master key k which is shared in advance between the two participants, Client and Server. This simplified version supports transmission of only two messages, a ‘request’ M_C sent by the client to the server, and a ‘response’ M_S sent from the server. The two messages are protected using a session key k' , which the server selects randomly at the beginning of each session, and sends to the client, protected using the fixed shared master key k .

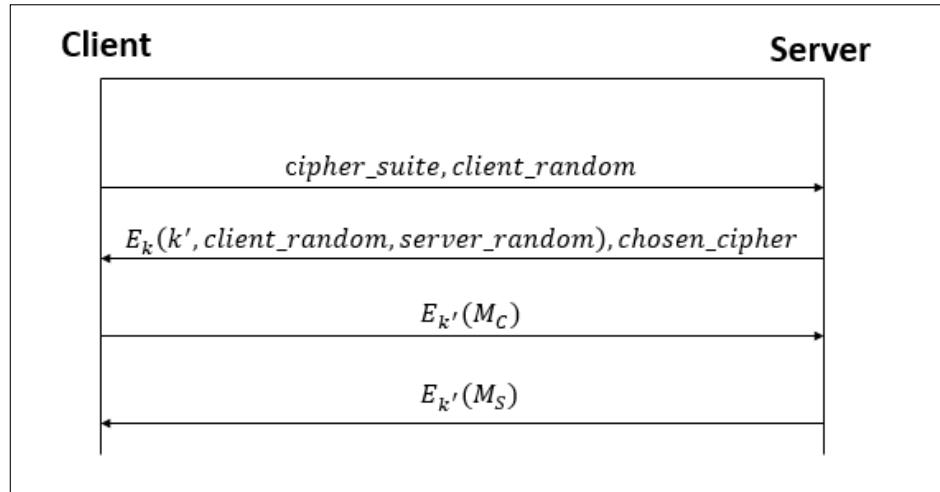


Figure 5.23: Simplified SSL

The protocol should protect the confidentiality and integrity (authenticity) of the messages (M_C, M_S), as well as ‘replay’ of messages, e.g., client sends M_C in one session and server receives M_C on two sessions.

1. The field `cipher_suite` contains a list of encryption schemes ('ciphers') supported by the client, and the field `chosen_cipher` contains the cipher in this list chosen by the server; this cipher is used in the two subsequent messages (a fixed cipher is used for the first two messages). For simplicity consider only two ciphers, say E_1 and E_2 , and suppose that both client and server support both, but that they prefer E_2 since E_1 is known to be vulnerable. Show how a MitM attacker can cause the parties to use E_1 anyway, allowing it to decipher the messages M_C, M_S .
2. Suggest a minor modification to the protocol to prevent such 'downgrade attacks'.
3. Ignore now the risk of downgrade attacks, e.g., assume all ciphers supported are secure. Assume that M_C is a request to transfer funds from the clients' account to a target account, in the following format:

Date (3 bytes)	Operation type (1 byte)	Comment (20 bytes)	Amount (8 bytes)	Target account (8 bytes)
-------------------	----------------------------	-----------------------	---------------------	-----------------------------

Assume that E is CBC mode encryption using an 8-bytes block cipher. The solution should not rely on replay of the messages (which will not work since only one message is sent in each direction on each usage).

Mal is a (malicious) client of the bank, and eavesdrops on a session where Alice is sending a request to transfer 10\$ to him (Mal). Show how Mal can abuse his Man-in-the-Middle abilities to cause transfer of larger amount. Explain a simple fix to the protocol to prevent this attack.

Exercise 5.15. Consider the following protocol for server-assisted group-shared-key setup. Every client, say i , shares a key k_i with the server. Let G be a group of users; user $i \in G$ can send, at any time, a request to the server for the (fixed) group key k_G ; the request consists of the list of users in G , the time t (in seconds) according to the clock of user i , and an authenticator $MAC_{k_i}(G, t)$. If the value of t is within one minute from its own clock value, the server responds by sending to i the encrypted key: $x_G(t) = k_G + \prod_{j \in G} PRF_{k_j}(t)$, where $\prod_{j \in G} PRF_{k_j}(t)$ is multiplication of the values $PRF_{k_j}(t)$ for every user j in G , including $j = i$. User i then computes $k_G^{(i)} = x_G \bmod PRF_{k_i}(t)$.

1. Draw a sequence diagram showing the operation of the protocol.
2. Let i, j be two users in group G , i.e., $i, j \in G$. Suppose user i sends request at time t_i and user j sends request at time t_j . Explain the conditions for them to receive the same key k_G .
3. Present an attack allowing a malicious user $m \notin G$ to learn the key k_G for a group it does not belong to. User m may eavesdrop to all messages, and request the key $k_{G'}$ for any group (set) of users G' s.t. $m \in G'$.

Exercise 5.16. In the GSM protocol, the home sends to the Visited Network one or more authentication triplets (r, K, s) . The Visited Network and the

mobile are to use each triplet only for a single handshake; this is somewhat wasteful, as often the mobile has multiple connections (and handshakes) while visiting the same Visited Network.

1. Suppose a Visited Network decides to re-use the same triplet (r, K, s) in multiple handshakes, for efficiency (less requests to home). Present message sequence diagram showing that this may allow an attacker to impersonate as a client. Namely, that client authentication fails.
2. Suggest an improvement to the messages sent between mobile and Visited Network, that will allow the Visited Network to reuse the (r, K, s) triplet received from Visited Network, for multiple secure handshakes with the mobile. Your improvement should consist of a single additional challenge r_B which the Visited Network selects randomly and sends to the mobile, together with the challenge r received in the triplet from the home; and a single response s_B which the mobile returns to the server, instead of sending the response s as in the original protocol. Show the computation of s_B by mobile and Visited Network: $s_B = \underline{\hspace{2cm}}$. Your solution may use an arbitrary pseudo-random function PRF.
3. GSM sends frames (messages) of 114 bits each, by bit-wise XORing the n^{th} plaintext frame with 114 bits output from $A5/i_K(n)$. Here, $A5/i$, for $i = 1, 2, \dots$, is a cryptographic function, n is the frame number, and K was a key received from the home. $A5/1$ and $A5/2$ are described in the specifications - and both are known to be vulnerable; other functions can be agreed between mobile and Visited Network. Both $A5/1$ and $A5/2$ are insecure; for this question, assume the use of a secure cipher, say A5/5. Suppose, again, that a Visited Network decides to re-use the same triplet (r, K, s) in multiple handshakes. A mobile has two connections to the Visited Network, sending message m_1 in the first connection and message m_2 in the second connection. Assume that the Visited Network re-uses the same triplet (r, K, s) in both connections, and that the attacker knows the contents of m_1 . Show how the attacker can find m_2 .

Note: the improvement suggested in the previous item (r_B, s_B) does not have significant impact on this item - you can solve with it or without it.

4. To prevent the threat presented in the previous item, the mobile and Visited Network can use a different key $K' = \underline{\hspace{2cm}}$ (instead of using K).
5. Design a Visited Network-only forward secrecy improvement to A5/5. Namely, even if attacker is given access to the entire memory of the Visited Network after the j^{th} handshake using the same r , the attacker would still not be able to decipher information exchanged in past connections. Your design may send the value of j together with r from Visited Network to mobile, and may change the stored value of s at the end of every handshake; let s_j denote the value of s at the j^{th} handshake, where the initial value is

s received from the home (i.e., $s_1 = s$). Your solution consists of defining the value of s_j given s_{j-1} , namely: $s_j = \underline{\hspace{2cm}}$.

Exercise 5.17 (GSM). Many GSM mobile phones use an encryption algorithm referred to as A5/3, when supported by the visited network, since it is considered more secure than A5/1 (and certainly more than A5/2, which was discontinued). The MAL organization records millions of A5/3 encrypted connections by different people ‘of interest’. MAL cryptanalysts find an effective attack against the GSM A5/1 algorithm; the attack exposes the key in few minutes, requiring only one ciphertext message. Suppose now Alice tries to communicate using her mobile and the GSM protocol, and the connection setup is intercepted by MAL. Show a sequence diagram showing how MAL may use this interception connection attempt and the attack found against A5/1, to decrypt prior GSM communication by Alice, which was encrypted using A5/3.

Exercise 5.18. Consider the following key establishment protocol between any two users with an assistance of a server S , where each user U shares a secret key K_{US} with a central server S .

$A \rightarrow B : (A, N_A)$

$B \rightarrow S : (A, N_A, B, N_B, \mathcal{E}_{K_{BS}}(A \# N_A))$

$S \rightarrow A : (A, N_A, B, N_B, \mathcal{E}_{K_{AS}}(N_A \# sk), \mathcal{E}_{K_{BS}}(A \# sk), N_B)$

$A \rightarrow B : (A, N_A, B, N_B, \mathcal{E}_{K_{BS}}(A \# sk))$

Assume that \mathcal{E} is an authenticated encryption. Show an attack which allows an attacker to impersonate one of the parties to the other, while exposing the secret key sk .

Exercise 5.19 (Hashing vs. Forward Secrecy). We discussed in §5.7.1 the use of PRG or PRF to derive future keys, ensuring Forward Secrecy. Could a cryptographic hash function be securely used for the same purpose, as in $\kappa_i = h(\kappa_{i-1})$? Evaluate if such design is guaranteed to be secure, when h is a (1) CRHF, (2) OWF, (3) bitwise-randomness extracting.

Exercise 5.20 (PFS definitions). Below are informal definitions for PFS from the literature. Compare them to our definitions for PFS: are they equivalent? Are they ‘weaker’ - a protocol may satisfy them yet not be PFS as we define, or the other way around? Or are they incomparable (neither is always weaker)? Can you give an absurd example of a protocol meeting the definition, which is ‘clearly’ not sensible to be claimed to be PFS? Any other issue?

From Wikipedia, [321] An encryption system has the property of forward secrecy if plain-text (decrypted) inspection of the data exchange that occurs during key agreement phase of session initiation does not reveal the key that was used to encrypt the remainder of the session.

From [227, 256] A protocol has Perfect Forward Secrecy (PFS) if the compromise of long-term keys does not allow an attacker to obtain past session keys.

Chapter 6

Public Key Cryptography

As we discussed in subsection 1.5.1, cryptography has been applied for over two millennia. However, until relatively recently, cryptography was always based on the use of *symmetric keys*. In particular, in Chapter 2, we studied symmetric cryptosystems, also called shared-key cryptosystems, which use the same key k for encryption ($c \leftarrow E_k(m)$) and for decryption ($m \leftarrow D_k(c)$); see in Figure 1.4. Similarly, in Chapter 4, we focused on shared-key (symmetric) Message Authentication Code (MAC) schemes, which also used only one key k to compute the authenticator (tag) that we will send with a message to prove its authenticity, and later to compute the authenticator for a message received and confirm it is identical to the one received with the message, proving its authenticity.

This changed quite dramatically by the publication, in 1976, of [101], a seminal paper by Diffie and Hellman introducing *public key cryptography*, also known as *asymmetric cryptography*. Asymmetric cryptography is built on the idea that we may use *different keys* for different functions, e.g., for encryption and for decryption. Of course, the keys may be *related*; for example, if we use a key e to encrypt, and a key d to decrypt, the pair (e, d) should be related to properly retrieve the plaintext: $m = D_d(E_e(m))$.

The advantage in asymmetric cryptography is that, for many applications, *one key can be public*, and only the other kept private. For example, Alice can publish her encryption key $A.e$, allowing everyone to encrypt messages to her by computing $c = E_{A.e}(m)$, but only Alice knows the corresponding decryption key $A.d$ such that $m = D_{A.d}(c)$. We refer to such asymmetric cryptosystems as *Public Key Cryptosystem (PKC)*; see illustration in Figure 1.5.

In [101], Diffie and Hellman identified three types of public-key schemes: *public-key cryptosystem*, *digital signatures* and *key exchange*. However, they only presented a design for the last one, namely the *DH key exchange* protocol. This discovery of the revolutionary concept of asymmetric cryptography is recreated in Figure 6.1¹.

¹Thanks to Whit and Marty for blessing this invented dialog.

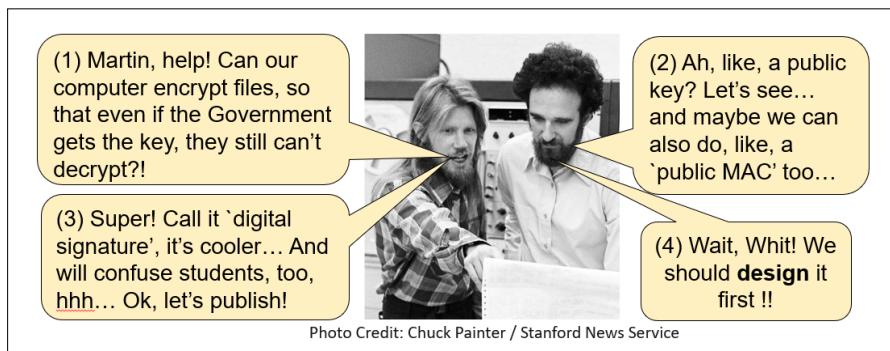


Figure 6.1: The discovery of Public-Key Cryptography by Whitfield Diffie and Martin Hellman.

In this chapter, we introduce public key cryptography. We begin, in the following section, with a brief introduction to public key cryptography. We then discuss key-exchange protocols, and later also public-key cryptosystems, mainly El-Auth- h -DH and RSA. We already discussed signature schemes and their security in subsection 1.4.1, but in subsection 6.6.1 we discuss the specific case of RSA-based public key signatures.

6.1 Introduction to PKC

The basic observation leading to asymmetric cryptography is quite simple in hindsight: *security requirements are asymmetric*. For example, to protect confidentiality, an encryption scheme should prevent an attacker from *decrypting ciphertext*, requiring the key used for decryption to be secret. However, confidentiality is not broken if the attacker can *encrypt* messages. In fact, in Def. 2.10 of security against chosen plaintext attack (CPA), we allow the adversary to *encrypt* plaintexts without any restriction; but we do not facilitate *decrypting* ciphertexts. Giving the attacker a key which allows encryptions but not decryptions, will not change security under this definition. Let us first discuss the three basic types of public key schemes introduced in [101], which are still the most important types of public key schemes: *public key cryptosystem (PKC)*, *digital signature* schemes, and *key exchange* protocols.

6.1.1 Public key cryptosystems

Public key cryptosystems (PKC) are encryption schemes consisting of three algorithms, (KG, E, D) , which use a pair of keys: a *public key* e for encryption, and a *private key* d for decryption. Both keys are generated by the *key generation* algorithm KG . The encryption key is not secret; namely, we assume that it is known to the attacker.

Let us now define a public key cryptosystem, similarly to Definition 2.1 for shared-key cryptosystems. As in Definition 2.1, we require *correctness*, i.e., that decryption of an encrypted message will recover that message. One notable difference is that a public key cryptosystem includes a *key generation* algorithm KG , since the encryption and decryption keys must be related - we cannot just choose them at random.

Definition 6.1 (Public-key cryptosystem (PKC)). A public-key cryptosystem (PKC) is triplet of (probabilistic) algorithms, (KG, E, D) and a set M (of plaintext messages), ensuring correctness, i.e., for every message $m \in M$ and key-pair $(e, d) \xleftarrow{\$} KG(1^l)$ holds:

$$D_d(E_e(m)) = m \quad (6.1)$$

See the illustration of a public key cryptosystem in Figure 1.5.

Other terms for public key cryptosystems include *asymmetric cryptosystems* and *public key encryption schemes*. We will try to stick to the term ‘public key cryptosystems’, often using just the acronym PKC.

We further discuss public key cryptosystems in sections 6.4.2 and 6.5.

6.1.2 Signature schemes

We introduced *signature schemes*, in Figure 1.7 and subsection 1.4.1,

Both signature schemes and Message Authentication Code (MAC) functions are used for *authentication* of messages. However, MAC functions use a single secret key k for authenticating and for validating messages, while signature schemes use a distinct private signing key s for signing (authenticating), and a distinct public verification key v for validating authenticity.

The correctness requirement of signature schemes is similar to the one for MAC schemes (Section 4.3), namely for security parameter 1^l , message m and key-pair $(s, v) \xleftarrow{\$} KG(1^l)$ holds:

$$V_v(m, S_s(m)) = \text{TRUE} \quad (6.2)$$

We presented constructions for one-time signatures in subsection 3.4.3. In Section 6.6, we discuss constructions of ‘regular’ signature schemes, i.e., schemes which may be used to sign arbitrary number of messages.

6.1.3 Public-Key-based Key Exchange Protocols

Public-key based Key Exchange protocols establish a shared secret key among two (or more) parties, based on the use of public keys. Such protocols may be *authenticated*, using keys shared in advance between the parties, or *unauthenticated*, not using any pre-shared keys. In this chapter, we focus on different variants of the *Diffie-Hellman (DH) key-exchange* protocol. We first discuss the unauthenticated Diffie-Hellman (DH) key-exchange protocols, in subsection 6.2.3, and later, in Section 6.3, discuss authenticated Diffie-Hellman (DH) key-exchange protocols.

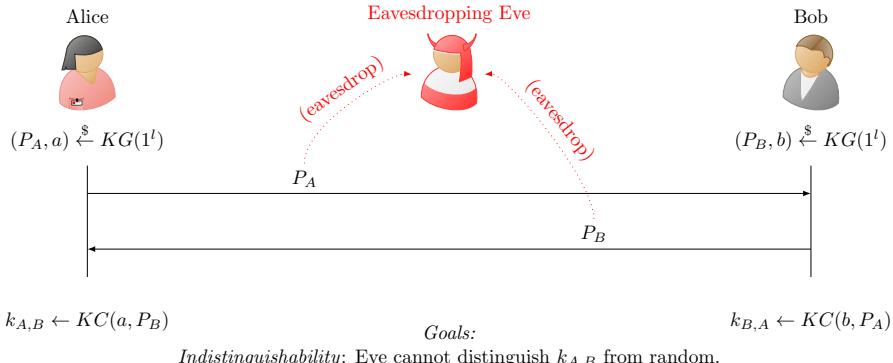


Figure 6.2: Operation of an arbitrary two-flows unauthenticated public-key based key exchange protocol, such as the *Diffie-Hellman* protocol (presented later). Such protocol is defined by two efficient probabilistic algorithms: *Key Generation* (KG), to generate (private, public) key-pairs, and *Key Combining* (KC), to combine the exchanged public keys (P_A and P_B) into the shared key: $k = KC(P_A, P_B)$. An unauthenticated key exchange protocol should be secure against an *eavesdropping adversary Eve*, but is vulnerable to a *Man-in-the-Middle* adversary.

Like other key-exchange protocols, the final output is a shared key, known to both (or all) participants; the goal is that the key would be completely hidden from the attacker. Public-key-based key exchange protocols are much more computationally-demanding than the shared-key key-exchange protocols we discussed in Chapter 5, and therefore, they are typically used only periodically, and the shared-key they output is usually referred to as a *master key*, and used later to derive shared *session keys*, possibly using (efficient) shared-key key-exchange protocols, as discussed in Section 5.4. Or, a combined protocol can be used to both share the master key and then to share session-keys based on the master key; in particular, this is done by the widely-used SSL/TLS protocols, which we study in Chapter 7.

The basic operation and goals of unauthenticated public-key-based key exchange protocols are illustrated in Figure 6.2, focusing on the case where the protocol involves only two flows: the first from Alice to Bob, and the second from Bob to Alice. To motivate the unauthenticated scenario, think that Alice and Bob meet in public, and want to establish secure communication between them; however, it being a public place, their discussion may be overheard, e.g., by Eavesdropping Eve. The key exchange protocol would allow them to establish a shared secret key, known only to the two of them, in spite of the possible eavesdropping.

Notice that during the run of the protocol, we allow the adversary (Eavesdropping Eve) only to *eavesdrop* to the communication between the parties; such adversary cannot modify or inject messages between the parties.

Defining two-flows unauthenticated public-key-based key exchange protocols. Let us now define, informally, an unauthenticated public-key-based key exchange protocol; for simplicity, let's focus on protocols using two flows, as in Figure 6.2. Such a key exchange protocol can be defined by a pair of efficient probabilistic algorithms, (KG, KC) (for key-generation and key-combining, respectively). The key-generation algorithm is given the key length (security parameter) in unary, 1^l , and outputs a pair of strings, e.g., (a, P_A) ; we refer to the first (a) as the *private key* and to the second (P_A) as the *public key*.

The *key generation function* KG receives as input a *security parameter* 1^l , and outputs a pair of keys, e.g., (P_A, a) , where P_A is a public key and a is a private key. The *key-combining function* KC is run by each party, and receives as input a public value (from the other party) and a private value (of the party running KC); the output of KC would be used as the key shared between the two parties. The keys derived by the Alice and Bob should be the same, i.e.: $KC(a, P_A) = KC(b, P_B)$.

A key exchange protocol should ensure *correctness* and *indistinguishability*. The *correctness* requirement is that both parties will derive the same key. More precisely, for every security parameter 1^l , the following should hold. Let $(a, P_A) \xleftarrow{\$} KG(1^l)$ and $(b, P_B) \xleftarrow{\$} KG(1^l)$ be two key-pairs generated, using the key-generation algorithm KG with security parameter 1^l , for Alice and Bob respectively. Then we have:

$$KC(a, P_B) = KC(b, P_A) \quad (6.3)$$

Namely, applying the key-combining algorithm KC to combine Alice's private key a with Bob's public key P_B , results in the same symmetric key as the one resulting from combining Bob's private key b with Alice's public key P_A .

Indistinguishability requires, intuitively, that an eavesdropping adversary, who 'sees' P_A and P_B , cannot learn anything about the shared key; equivalently, it requires that the adversary cannot *distinguish* between being given randomly-generated P_A, P_B and the key derived from them, versus being given randomly-generated P_A, P_B and a random string of the same length as the key. The following definition states this requirement more precisely.

Definition 6.2 (The indistinguishability requirement). *Let (KG, KC) be a key-exchange protocol, and \mathcal{A} be an efficient (PPT) adversary. We say that (KG, KC) ensures key-indistinguishability if for every PPT adversary \mathcal{A} and for sufficiently-large security parameter 1^l , holds:*

$$\Pr \left[\begin{array}{c} \mathcal{A}(P_A, P_B, KC(a, P_A)) = 1 \\ \text{where} \\ (a, P_A) \xleftarrow{\$} KG(1^l), \\ (b, P_B) \xleftarrow{\$} KG(1^l) \end{array} \right] - \Pr \left[\begin{array}{c} \mathcal{A}(P_A, P_B, r) = 1 \\ \text{where} \\ (a, P_A) \xleftarrow{\$} KG(1^l), \\ (b, P_B) \xleftarrow{\$} KG(1^l), \\ r \xleftarrow{\$} \{0, 1\}^{|KC(a, P_A)|} \end{array} \right] \in NEGL(1^l) \quad (6.4)$$

6.1.4 Advantages of Public Key Cryptography (PKC)

Public key cryptography is not just a cool concept; it is very useful, allowing solutions to problems which symmetric cryptography fails to solve, and making it easier to solve other problems.

We first identify three important challenges which *require* the use of asymmetric cryptography:

Signatures provide evidence. Only the owner of the private key can digitally sign a message, but everyone can validate this signature. This allows a recipient of a signed message to know that once he validated the signature, he has the ability to convince other parties that the message was signed by the sender. This is impossible using (shared-key) MAC schemes, and allows many applications, such as signing an agreement, payment order or recommendation/review. An important special case is signing a *public key certificate*, linking an entity and its public key.

Security without assuming shared key. Using public key cryptography, we can establish secure communication between parties, without requiring them to previously share a secret key between them, or to share a secret key *and* communicate with an additional party (such as a *KDC*, see Section 5.5). One method to do so is to use unauthenticated *key-exchange* protocol; this is secure if the attacker only has eavesdropping capabilities during the exchange (this is not secure against a MitM attacker). Another alternative is when one party (e.g., the client) knows, or can securely receive, the public key of the other party (e.g., the server); in this case, the client can encrypt a shared key and send it to the server. To allow a party, e.g., the client (Alice), to validate the public key of the other party, e.g., the server (Bob), we can send the public key P_B signed by a trusted party. We refer to the signed public key as a *public key certificate*; public key certificates are a very important aspect of applied cryptography, and we discuss them extensively in Chapter 8.

Stronger resiliency to exposure. In Section 5.7 we discussed the goal of resiliency to exposure of secret information, in particular, of the ‘master key’ of shared-key key-setup protocols, and presented the *forward secrecy key-setup handshake*. In subsection 5.7.3, we also briefly discussed some stronger resiliency properties, including *Perfect Forward Secrecy (PFS)*, *Threshold security* and *Proactive security*. Designs for achieving such stronger resiliency notions are all based on public key cryptography; we discuss these in Section 6.3.

Public key cryptography (PKC) also makes it *easier* to design and deploy secure systems. Specifically:

Easier key distribution: public keys are easier to *distribute*, since they can be given in a public forum (such as directory) or in an incoming message; note that the public keys still need to be authenticated, to be sure we

are receiving the correct public keys, but there is no need to protect their secrecy. Distribution is also easier since each party only needs to distribute one (public) key to all its peers, rather than setting up different secret keys, one per each peer.

Easier key management: public keys are easier to maintain and use, since they may be kept in non-secure storage, as long as they are validated before being used.

Less keys: Only one public key is required for each party, compared to a total of $\frac{n \cdot (n-1)}{2} = O(n^2)$ shared keys required for each pair of n entities. Namely, we need to maintain - and refresh - less keys.

Considering all these advantages, one may wonder why not *always* use public key cryptography. The reason is that there is also a price to the use of PKC - as we next discuss.

6.1.5 The price of PKC: assumptions, computation costs and length of keys and outputs

With all the advantages listed above, it may seem that we should *always* use public key cryptography. However, PKC has three significant drawbacks: computation time, key-length and potential vulnerability. We discuss these in this subsection.

All of these drawbacks are due to the fact that when attacking a PKC scheme, the attacker has the *public key* which corresponds to the private key. The private key is closely related to the public key - for example, the private decryption key ‘reverses’ encryption using the public key; yet, the public key should not expose (information about) the private key. It is challenging to come up with a scheme that allows this relationship between the encryption and decryption keys, and yet where the public key does not expose the private key. In fact, as discussed in Section 1.5, the *concept* of PKC was ‘discovered’ twice!

Considering the challenge of designing asymmetric cryptosystems, it should not be surprising that all known public-key schemes have considerable drawbacks compared to the corresponding shared-key (symmetric) schemes. There are two types of drawbacks: *overhead* and *required assumptions*.

PKC assumptions and quantum cryptanalysis Applied PKC algorithms, such as RSA, DH, El-Gamal and elliptic-curve PKCs, all rely on specific computational assumptions, mostly on the hardness of specific number-theoretic problems, mainly two: *factoring* and *discrete logarithm*.

These specific hardness assumptions, and several others, are usually considered *well-founded*. This is due to the extensive efforts of mathematicians and other experts to find efficient algorithms for these problems. In particular, factoring and discrete logarithms have been studied for many years, long before

their use for PKC was proposed; and efforts increased by far as PKC became known and important.

However, it is certainly conceivable that an efficient algorithms exists - and would someday be found. Such a discovery may even occur suddenly and soon - such unpredictability is the nature of algorithmic and mathematical breakthroughs. In particular, a recent draft [291] presented a new factoring method, which claimed to be fast enough to be practical for significant key-lengths, and specifically to ‘destroy the RSA cryptosystem’. As the time of writing, this draft was withdrawn, and may be incorrect; but it will not be shocking if such an algorithm were to be found, indicating that RSA security may be considerably less than currently estimated, requiring the use of either longer keys or other schemes.

Furthermore, since all of the widely-used PKC algorithms are so closely related, it is even possible that some, potentially related, advances in cryptanalysis would apply to all of them - leaving us without any practical PKC algorithm. PKC algorithms are the basis for the security of many systems and protocols; if suddenly there were no viable, practical and unbroken PKC, that would be a major problem.

And if all that is not alarming enough, efficient algorithms to solve both the factoring and the discrete logarithm problems *are known*, requiring an appropriate *quantum computer*. There has been many efforts to develop quantum computers, with significant progress - but results are still far from the ability to cryptanalyze these PKC schemes, when used with key-lengths which are considered secure (against known attacks, using standard computing devices). However, that may change with improvements in quantum computing.

Cryptographers work hard to identify additional candidate PKC systems, which will rely on other, ‘independent’ or - ideally - more general assumptions, as well as schemes which are secure even if large-scale quantum computing becomes feasible, which are referred to as *post-quantum cryptography*. We discuss the impact of quantum computing on cryptography, including both use for cryptanalysis and development of post-quantum cryptography, in Section 10.4.

One particularly interesting approach to the development of cryptographic schemes robust to advances in algorithms for specific problems, is the design of PKC schemes based on *lattice* problems. Lattice problems seem resilient to quantum-computing; furthermore, some of the results in this area have proofs of security based on the general and well-founded complexity assumption of NP-completeness. Details are beyond our scope; see, e.g., [12, 259].

PKC overhead: key-length and computation. Another drawback of asymmetric cryptography, is that all of the proposed schemes - definitely, all proposed schemes which were not broken - have much higher overhead, compared to the corresponding shared-key schemes. There are two main types of overhead: *computation time* and *key-length*.

The system designers choose the key-length of the cryptosystems they use, based on the *sufficient effective key length* principle (principle 5). These decisions

	Symmetric Cryptography			Factoring (RSA), Discrete-log (DH)			Elliptic curves (ECIES)		
Estimation Year	LV 2002	NIST 2014	BSI 2017	LV 2002	NIST 2014	BSI 2017	LV 2002	NIST 2014	BSI 2017
2020	86	112	128	1881	2048	2000	161	224	250
2030	93	112	128	2493	2048	3000	176	224	250
2040	101	128	128	3214	3072	3000	191	256	250
Crypto++ [92]	$4.5 \cdot 10^9$ bytes/sec 128-bits AES			$3 \cdot 10^5$ bytes/sec 2048-bits RSA/DH			$3 \cdot 10^4$ bytes/sec 256-bits ECIES		

Table 6.1: Comparison of key length and computing time for asymmetric and symmetric cryptography. Table shows three recommendations for key-length, in bits, required for confidentiality against ‘commercial’ adversaries. The recommendations are given for widely-deployed public-key (asymmetric) and shared-key (symmetric) cryptosystems; the rows refer to the year in which confidentiality is to be preserved (2020, 2030 and 2040). The recommendations are based on predictions of advances in both computing power and cryptanalysis. The LV recommendations are from a 2002 paper [215], the NIST recommendations are from a 2014 publication [21] and the BSI values are from a 2017 publication [73]. The bottom row compares the performance of the schemes for the Crypto++ implementation, based on [92].

are based on the perceived resources and motivation of the attackers, on their estimation or bounds of the expected damages due to exposure, and on the constraints and overheads of the relevant system resources. Finally, a critical consideration is the estimates of the required key length for the cryptosystems in use, based on known and estimated future attacks. Such estimates and recommendations are usually provided by experts proposing new cryptosystems, and then revised and improved by experts and different standardization and security organizations, publishing key-length recommendations.

We present three well-known recommendations in Table 6.1. These recommendations are marked in the table as *LV'01*, *NIST2014* and *BSI'17*, and were published, respectively, in a seminal 2001 paper by Lenstra and Verheul [215], by NIST in 2014 [21] and by the German BSI organization in 2017 [73]. See these and much more online at [138].

Recommendations are usually presented with respect to a particular *year* in which the ciphertexts are to remain confidential (the three rows for 2020, 2030 and 2040 in Table 6.1). Experts estimate the expected improvements in the cryptanalysis capabilities of attackers over years, due to improved hardware speeds, reduced hardware costs, reduced energy costs (due to improved hardware), and, often more significantly but hardest to estimate, improvements in methods of cryptanalysis. Such predictions cannot be done precisely, and hence, recommendations differ, sometimes considerably.

Table 6.1 presents the recommendations for four typical, important cryptosystems (in columns two to four). Column two presents the recommendations for a *symmetric cryptosystem* such as AES. The recommendations for sym-

metric cryptosystems are not limited to AES; they apply to *any* symmetric (shared-key) cryptosystem. They only require that the best attacks against the system are generic attacks such as *exhaustive search* (subsection 2.3.1) or *table lookup* (subsection 2.3.2); symmetric cryptosystem against which there is a more effective attack are typically considered insecure and avoided.

Column three presents the recommendations for RSA and El-Gamal, the two oldest and most well-known public-key cryptosystems; we discuss both cryptosystems, in sections 6.5 and 6.4.2. This column also applies to the Diffie-Hellman (DH) key-exchange protocol; in fact, the El-Gamal cryptosystem is essentially a variant of the DH protocol, as we explain in subsection 6.4.2. RSA and El-Gamal/DH are based on two different number-theoretic problems: the factoring problem (for RSA) and the discrete-logarithm problem (for DH/El-Gamal); but the best-known attacks against both are related, with running time which is exponential in *half* the key-length. We briefly discuss these problems in subsection 6.1.7.

The fourth column of table 6.1 presents the recommendations for elliptic-curve based public-key cryptosystems such as ECIES. As the table shows, the recommended key-lengths for elliptic-curve based public-key cryptosystems are, quite consistently, much lower than the recommendations for the ‘older’ RSA and El-Gamal/DH systems; this makes them attractive in applications where longer keys are problematic, due to storage and/or communication overhead. We do not cover elliptic-curve cryptosystems in this textbook; these are covered in other courses and books, e.g., [12, 158, 303].

Table 6.1 shows that the required key-length is considerably higher for public-key schemes, compared to shared-key (symmetric) schemes. Symmetric cryptography requires only about half of the key-length required by Elliptic-curve cryptosystems, and only about 5% of the key length required, for the same level of security, when using the RSA and DH public-key schemes. The lower key-length recommendations for Elliptic-curve cryptography, makes these schemes attractive in the (many) applications where key-length is critical, such as when communication bandwidth and/or storage are limited.

The bottom row of Table 6.1 compares the *running time* of implementations of AES with 128 bit key in counter (CTR) mode, RSA with 1024 and 2048 bit key, and 256 bit ECIES elliptic curve cryptosystem. We see that the symmetric cryptosystem (AES) is many orders of magnitude faster. It supports about $4.5 \cdot 10^9$ bytes/second, compared with about $3 \cdot 10^5$ bytes/second for the comparably-secure 2048-bit RSA, and less than $3 \cdot 10^4$ bytes/second for ECEIS. We used the values reported for one of the popular Cryptographic libraries, Crypto++ [92].

The minimize use of PKC principle. In this subsection we have seen several serious concerns with the use of asymmetric (public key) cryptography. First, practical, deployed public key cryptographic algorithms, are secure only with specific assumptions - which have held many years, true, but still may be broken, e.g., by new, faster factoring algorithms [291]. Second, applied

public-key techniques may be vulnerable to further improvements in quantum computing. Finally, as Table 6.1 shows, asymmetric (public key) cryptography has much higher overhead compared to symmetric cryptography. From all of this, we conclude the following principle:

Principle 12 (Minimize use of public-key cryptography). *Designers should avoid, or, where absolutely necessary, minimize the use of public-key cryptography.*

In particular, consider that typical messages are much longer than the size of inputs to the public-key algorithms. If we ignored the high costs of asymmetric cryptography, we could split the input into ‘blocks’ whose size is the allowed input-size of the public-key algorithms, and then use ‘modes of operations’, like these presented for encryption and MAC, for applying the public-key algorithms to multiple blocks. However, the resulting computation costs would have been absurd. Even *more* absurd, although theoretically possible, would be to modify the public-key operation to directly support longer inputs. Luckily, there are simple and efficient solutions, to both encryption and signatures, which are used essentially universally, to apply these schemes to long (or VIL) messages:

Signatures: use the *Hash-then-Sign (HtS)* paradigm, see subsection 3.2.6.

Encryption: use the *hybrid encryption* paradigm, see the following subsection (subsection 6.1.6).

6.1.6 Hybrid Encryption

The huge performance overhead of asymmetric cryptosystems implies that they are typically used mainly when the parties do not share a symmetric key. Furthermore, even when the parties do not share a symmetric key, we usually do not use directly the widely-used, ‘classical’ *asymmetric* cryptosystems (KG^A, E^A, D^A) , e.g., RSA. Instead, we usually combine such ‘classical’ asymmetric cryptosystem (KG^A, E^A, D^A) , with an efficient *symmetric* cryptosystem (E^S, D^S) , e.g., AES. Namely, we construct a new, *hybrid* asymmetric cryptosystem, which we denote (KG^H, E^H, D^H) . Note our use of mnemonic superscripts to distinguish between the three cryptosystems: A for the ‘classical’ asymmetric cryptosystem (KG^A, E^A, D^A) , S for the symmetric cryptosystem (E^S, D^S) and H for the *hybrid* cryptosystem (KG^H, E^H, D^H) .

Hybrid encryption is to obtain the benefits of asymmetric (public key) cryptosystems, yet with much reduced overhead, for the typical case where the plaintext is much longer than the input size of the ‘classical’ public-key encryption. For example, when using RSA with 4000-bit keys, the input size must be less than 4000 bits; with hybrid encryption, we can encrypt much longer messages, with only a single (4000-bit) RSA encryption, plus the number of symmetric-key operations required to encrypt the plaintext.

Almost universally, hybrid encryption is achieved using the simple construction illustrated in Figure 6.3. Note that Figure 6.3 shows only the encryption

and decryption processes of the hybrid encryption scheme; this is since in this common construction, the hybrid encryption scheme (KG^H, E^H, D^H) uses the same key-generation function as that of the underlying asymmetric encryption scheme, i.e., $KG^H(1^l) = KG^A(1^l)$.

Let us explain the hybrid encryption and decryption processes, as illustrated in Figure 6.3.

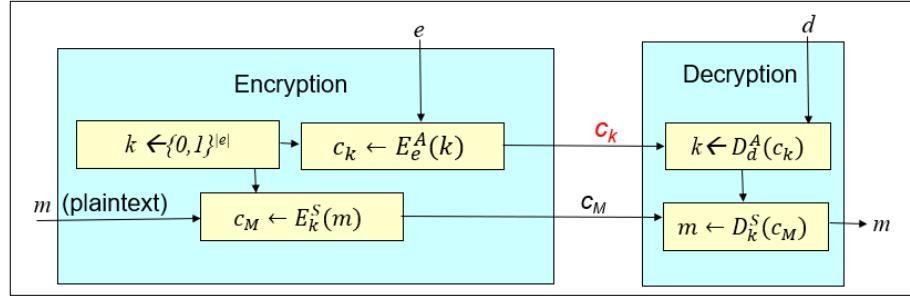


Figure 6.3: Hybrid encryption (KG^H, E^H, D^H) , defined as a combination of an asymmetric (public key) cryptosystem (KG^A, E^A, D^A) with shared key cryptosystem E^S, D^S , to allow efficient public key encryption of long message m using public key e and security parameter 1^l .

The hybrid encryption process $E_e^H(m)$. We now explain how we perform hybrid encryption, given message m and using the public key e , as illustrated in Figure 6.3. We first select a random l -bit symmetric key $k \xleftarrow{\$} \{0, 1\}^{|e|}$; note that, for simplicity, we use the length of the public key as the length of the shared key too (in practice, a shorter key usually suffices). We then use this key k to encrypt the message using symmetric encryption, i.e., compute the *cipher-message* $c_M = E_k^S(m)$. Finally, we then use the public-key encryption, to encrypt the symmetric key k , i.e., compute the *cipher-key* c_K , as: $c_K = E_e^A(k)$. The ciphertext of the hybrid encryption is the pair of cipher-message and cipher-key, i.e., (c_M, c_K) . More formally, we define:

$$E_e^H(m) \leftarrow (c_M, c_K) \text{ where } \begin{bmatrix} k \xleftarrow{\$} \{0, 1\}^{|e|} \\ c_M \leftarrow E_k^S(m) \\ c_K \leftarrow E_e^A(k) \end{bmatrix} \quad (6.5)$$

The hybrid decryption process $D_d^H((c_M, c_K))$. We now explain how we perform hybrid decryption, given the private key d and the ciphertext, which should be a pair (c_M, c_K) of cipher-message and cipher-key. The hybrid decryption process is illustrated in Figure 6.3. We first decrypt the symmetric key, by: $k \leftarrow D_d^A(c_K)$, and then use this key k to decrypt the message, using the symmetric decryption function: $m \leftarrow D_k^S(c_M)$. More formally, we define

the hybrid decryption process as:

$$D_d^H((c_M, c_K)) \leftarrow D_k^S(c_M), \text{ where } k \leftarrow D_d^A(c_K) \quad (6.6)$$

Exercise 6.1. *Prove, or present counterexample to, the following claim: if the asymmetric (KG^A, E^A, D^A) and symmetric (E^S, D^S) cryptosystems ensure correctness (per definitions 6.1 and 2.1, respectively), then the hybrid cryptosystem (KG^H, E^H, D^H) , defined as above, is also an asymmetric cryptosystem (PKC) that ensures correctness.*

6.1.7 The Factoring and Discrete Logarithm Hard Problems

As discussed in Section A.1, cryptography, and in particular public-key cryptography is based on the theory of complexity, and specifically on (computationally) *hard problems*. Intuitively, a hard problem is a family of computational problems, with two properties:

Easy to verify: there is an efficient (PPT) algorithm to verify solutions.

Hard to solve: there is no known efficient algorithm that solves the problem (with significant probability). We refer here to *known* algorithms; it is unreasonable to expect a *proof* that there is no efficient algorithm to a problem for which there is an efficient (PPT) verification algorithm. The reason for that is that such a proof would also solve the most important, fundamental open problem in the theory of complexity, i.e., it would show that $NP \neq P$. See Section A.1 or relevant textbooks, e.g., [140].

Intuitively, public key schemes use hard problems, by having the secret key provide the solution to the problem, and the public key provide the parameters to verify the solution. To make this more concrete, we briefly discuss *factoring* and *discrete logarithm*, the two hard problems which are the basis for many public key schemes, including the oldest and most well known: *RSA*, *DH*, *El-Gamal*. For more in-depth discussion of these and other schemes, see courses and books on cryptography, e.g., [303]. Note that while so far, known attacks are equally effective against both systems (see Table 6.1), there is not yet a proof that an efficient algorithm for one problem implies an efficient algorithm against the second.

Note: both the factoring and the discrete logarithm problems are well-known problems from the domain of number theory. Properly understanding these problems, as well as the operation and security of the public key cryptosystems we present which are related to these problems, may require familiarity with some basic notions of number theory. These are summarized in Section A.2.

Factoring. The factoring problem is one of the oldest problems in algorithmic number theory, and is the basis for RSA and other cryptographic schemes. Basically, the factoring problem involves finding the prime divisors (factors) of a

large integer. However, most numbers have small divisors - half of the numbers divide by two, third divide by three and so on... This allows efficient *number sieve* algorithms to factor most numbers. Therefore, the factoring hard problem refers specifically to factoring of numbers which have only *large* prime factors.

For the RSA cryptosystem, in particular, we consider factoring of a number n computed as the product of two large random primes: $n = pq$. The factoring hard problem assumption is that given such n , there is no efficient algorithm to factor it back into p and q .

Verification consists simply of multiplying p and q , or, if given only one of the two, say p , of dividing n by p and confirming that the result is an integer q with no residue.

Discrete logarithm. The *discrete logarithm* problem is another important, well-known problem from algorithmic number theory - and the basis for the DH (Diffie-Hellman) key-exchange protocol, the El-Gamal cryptosystem, elliptic-curve cryptography, and additional cryptographic schemes.

Discrete logarithms are defined for a given *cyclic group* \mathbb{G} and a generator g of \mathbb{G} ; see background in subsection A.2.4. Given a generator g of a finite cyclic group \mathbb{G} , and an element $x \in \mathbb{G}$, an integer y is called the *discrete logarithm* of x over \mathbb{G} with respect to g , if $x = g^y$. Note that multiplication (and exponentiation) are done using the group operation of \mathbb{G} ; e.g., for the modulo p group $\mathbb{Z}_p^* \equiv \{1, 2, \dots, p-1\}$, we require $x \equiv g^y \pmod{p}$.

Discrete logarithms are similar to the ‘regular’ logarithm function $\log_b(x)$ over the real numbers \mathbb{R} , which returns the number $y \in \mathbb{R}$ s.t. $y = b^y$. Discrete logarithms, unlike ‘regular’ logarithms, are computed over the finite cyclic group \mathbb{G} rather than over the real numbers, and use the group operation of \mathbb{G} rather than multiplication over the real numbers.

Intuitively, an algorithm that outputs a discrete logarithm a given an element $x \in \mathbb{G}$ and the generator g is said to solve the *discrete logarithm problem* for \mathbb{G} . We say that the discrete logarithm problem is *hard* for finite cyclic group \mathbb{G} , if there is no efficient (PPT) algorithm \mathcal{A} that solves the discrete logarithm problem for \mathbb{G} (with significant probability of success). This is in contrast to the logarithm function over the real numbers, which is efficiently computable. Note, however, that for any group \mathbb{G} , it only requires an exponentiation to verify whether $x = g^y$, and exponentiation can be computed quite efficiently.

This discussion is only intuitive, since we did not clearly define the input of the algorithm \mathcal{A} . This may suffice for most readers; however, for interested readers, we present also a precise definition of the discrete logarithm problem. In this definition, we consider a PPT algorithm Gen which receives, as input, a security parameters 1^l , and *generates* (outputs) the generator g the order q of \mathbb{G} .

Definition 6.3 (The discrete logarithm problem). *Let Gen be a PPT algorithm that, on input 1^l , outputs (g, q) such that $\{1, g, \dots, g^q\}$ is a cyclic group (using a given group operation). We say that the discrete logarithm problem is hard*

for groups generated by Gen , if for every PPT algorithm \mathcal{A} holds:

$$\Pr \left[(g, q) \leftarrow \text{Gen} \left(1^l \right) ; a \xleftarrow{\$} \{1, \dots, q\} : a = \mathcal{A}(g^a) \right] \in \text{NEGL}(1^l) \quad (6.7)$$

In practical cryptography, the discrete logarithm problem is used mostly for (cyclic) groups defined by multiplications modulo a prime p , often the cyclic group $\mathbb{Z}_p^* \equiv \{1, 2, \dots, p-1\}$. However, for some primes p , the discrete-logarithm problem is *easy* for \mathbb{Z}_p^* . In particular:

Fact 6.1. *Let p be a prime. If $p - 1$ has only ‘small’ prime factors, then there are known algorithms, such as the Pohlig-Hellman algorithm [262], that efficiently compute discrete logarithms.*

This motivates the use of a modulus p which is a prime without small factors. In this textbook, we focus on a special case called *safe prime*, as we next define.

Definition 6.4 (Safe prime). *A prime number $p \in \mathbb{N}$ is called a safe prime, if $p = 2q + 1$ for some prime $q \in \mathbb{N}$. If p is a safe prime, we say that the group \mathbb{Z}_p^* , containing the numbers from 1 to $p - 1$, with the modular multiplication operation, is a safe prime group.*

Many efforts have failed to find an efficient algorithm to compute discrete-logarithms for *safe prime groups*. As a result, the discrete-logarithm problem is widely believed to be *hard* for the mod- p group, \mathbb{Z}_p^* , if p is a safe prime.

For efficiency and/or security considerations, some designs use other finite cyclic groups for which the discrete logarithm problem is considered hard, which are not safe prime groups; one example are groups defined using elliptic curves.

6.1.8 The secrecy implied by the discrete logarithm assumption

Suppose that the discrete logarithm assumption for safe prime groups holds, i.e., it is computationally-hard to find the discrete-log a , given $g^a \bmod p$, where g is a generator of the safe prime group. Does this mean that the attacker cannot learn *any* information on a ?

The answer is *no*. Furthermore, we show that the attacker can efficiently learn *some* information about a - specifically, its *least-significant bit* (*Lsb*), i.e., if a is even ($\text{Lsb}(a) = 0$) or odd ($\text{Lsb}(a) = 1$). As we will see later, this has important implications on the design of some discrete logarithm-based cryptographic schemes, such as the ‘secure’ way to use the Diffie-Hellman protocol; see Claim 6.3.

Learning $\text{Lsb}(a)$ is based on the notion of *quadratic residue modulo p*, which has many uses in the mathematics of cryptography.

Definition 6.5 (Quadratic residue). *Let p be a prime number, and let y be a positive integer. We say that y is a quadratic residue modulo p , if there is some integer z s.t. $y \equiv z^2 \pmod{p}$.*

We first claim, without proof, that quadratic residuosity² can be efficiently determined.

Claim 6.1. *Given a prime p , there is an efficient algorithm that can determine if a given positive integer y is a quadratic residue modulo p .*

Proof: omitted; see, e.g., [171]. \square

We next show that $y = g^x \pmod{p}$ is a quadratic residue modulo p if and only if $LSb(x) = 0$, i.e., the least significant bit of x is zero, or equivalently, x is even. Combined with Claim 6.1, this shows that we can efficiently find the least-significant bit of the exponent x .

Claim 6.2. *Let p be a prime, g be a generator for \mathbb{Z}_p^* , and x be a positive integer. Then $y \equiv g^x \pmod{p}$ is a quadratic residue mod p , if and only if $LSb(x) = 0$, i.e., x is even.*

Proof: Let us first prove that if x is even, i.e., $LSb(x) = 0$, then $y = g^x \pmod{p}$ is a quadratic residue. First observe, that if x is even, then there is some integer z s.t. $x = 2z$. Hence, $y \equiv g^{2z} \equiv (g^z)^2 \pmod{p}$; i.e., y is, indeed, a quadratic residue.

We now the other direction, i.e., let $y = g^x \pmod{p}$ be a quadratic residue mod p , where x is an integer; we prove that $LSb(x) = 0$, i.e., x is even. This proof uses basic facts from number theory, which we present in subsection A.2.3.

For any odd number m , there exists an integer k such that $m = 2k + 1$. Let us assume, to the contrary, that g^m is a quadratic residue mod p , for some odd integer m ($LSb(m) = 1$); namely, $g^m \equiv z^2 \pmod{p}$ for some integer z . From Fermat's theorem (Theorem A.1) follows that:

$$z^{p-1} \equiv 1 \pmod{p} \quad (6.8)$$

However, on the other hand:

$$z^{p-1} \equiv z^{2 \cdot \frac{p-1}{2}} \equiv (z^2)^{\frac{p-1}{2}} \equiv (g^m)^{\frac{p-1}{2}} \equiv g^{(2k+1) \cdot \frac{p-1}{2}} \equiv g^{k \cdot (p-1)} \cdot g^{\frac{p-1}{2}} \pmod{p} \quad (6.9)$$

Now, again from Fermat's theorem, we have:

$$g^{k \cdot (p-1)} \equiv (g^{p-1})^k \equiv 1^k \equiv 1 \pmod{p} \quad (6.10)$$

By combining Equations (6.8-6.10), we have:

$$g^{\frac{p-1}{2}} \equiv 1 \cdot g^{\frac{p-1}{2}} \equiv g^{k \cdot (p-1)} \cdot g^{\frac{p-1}{2}} \equiv z^{p-1} \equiv 1 \pmod{p} \quad (6.11)$$

²Determination of quadratic residuosity is equivalent to computation of the *Legendre symbol*, defined as

$$\frac{y}{p} \equiv \begin{cases} 1 & \text{if } y \text{ is a quadratic residue modulo } p \text{ and } 0 \neq a \pmod{p} \\ -1 & \text{if } y \text{ isn't a quadratic residue modulo } p \\ 0 & \text{if } y = 0 \pmod{p}. \end{cases}$$

where p is a prime and y is an integer.

Namely, $g^{\frac{p-1}{2}} \equiv 1 \pmod{p}$.

However, g is a generator of \mathbb{Z}_p^* , i.e., $g^k \not\equiv 1$ for every integer k s.t. $1 \leq k < p - 1$, and in particular for $k = \frac{p-1}{2}$. This contradicts $g^{\frac{p-1}{2}} \equiv 1 \pmod{p}$, i.e., Equation 6.11. \square

Claim 6.2 shows that by (efficiently) finding if $g^x \pmod{p}$ is a quadratic residue modulo p (Claim 6.1), we can find the least-significant bit of x ($LSb(x)$), indicating if x is even or odd. Namely, while it may be hard to compute the entire discrete logarithm (x , given $g^x \pmod{p}$), it is possible to efficiently find at least one bit of x - the least significant bit.

6.2 The DH Key Exchange Protocol

A major motivation for public key cryptography, is to secure communication between parties, without requiring the parties to previously agree on a shared secret key. In their seminal paper [101], Diffie and Hellman introduced the *concept* of public key cryptography, including public-key cryptosystem (PKC), which indeed allows secure communication without a preshared secret key. However, this paper did not contain a proposal for *implementing* a PKC.

Instead, [101] introduced the *key exchange* problem, and present the *Diffie-Hellman (DH) key exchange protocol*, often referred to simply as the DH protocol. Although a key-exchange protocol is not a public key cryptosystem, yet it also allows secure communication - without requiring a previously shared secret key. In fact, the *goal* of a key exchange protocol, is to *establish* a shared secret key.

In this section, we explain the DH protocol, by developing it in three steps - each in a subsection. In subsection 6.2.1 we discuss a ‘physical’ variant of the DH protocol, which involves physical padlocks and exchanging a box (locked by one or two locks).

6.2.1 Physical key exchange

To help understand the Diffie-Hellman key exchange protocol, we first describe a *physical* key exchange protocol, illustrated by the sequence diagram in Fig. 6.4. In this protocol, Alice and Bob exchange a secret key, by using a *box*, and two padlocks - one of Alice and one of Bob. Note that initially, Alice and Bob do not have a shared key - and, in particular, Bob cannot open Alice’s padlock and vice versa; the protocol nevertheless, allows them to securely share a key.

Alice initiates the protocol by placing the key to be shared in the box, and locking the box with her padlock. When Bob receives the locked box, he cannot remove Alice’s padlock and open the box. Instead, Bob *locks* the box with his own padlock, *in addition to Alice’s padlock*. Bob now sends the box, locked by *both* padlocks, to Alice.

Upon receiving the box, locked by both padlocks, Alice removes her own padlock and sends back the box, now locked only by Bob’s padlock, back to Bob. Finally, Bob removes his own padlock, and is now able to open the box and find the key sent by Alice. We assume that the Man in the Middle adversary

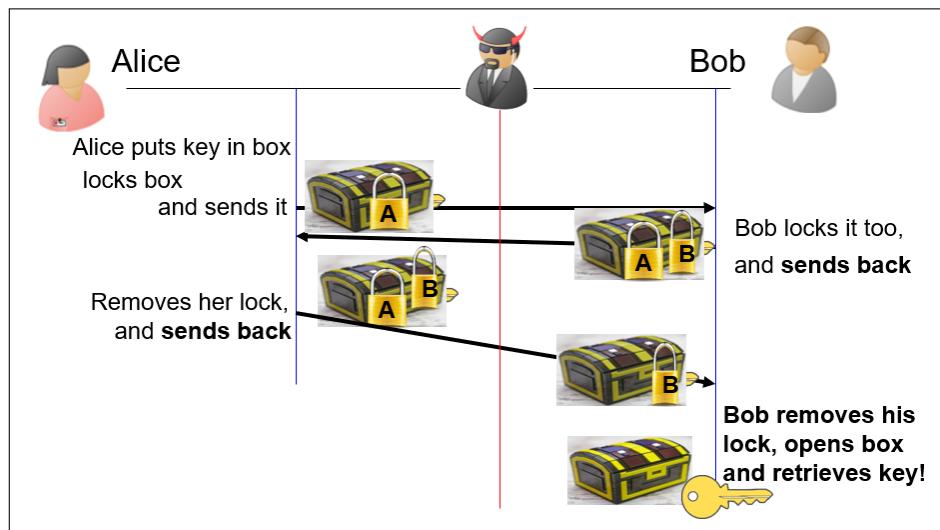


Figure 6.4: Physical Key Exchange Protocol

cannot remove Alice's or Bob's padlocks, and hence, cannot learn the secret in this way. The Diffie-Hellman protocol replaces this physical assumption, by appropriate cryptographic assumptions.

However, notice that there is a further limitation on the adversary, which is crucial for the security of this physical key exchange protocol: the adversary should be unable to send a *fake padlock*. Note that in Figure 6.4, both padlocks are stamped by the initial of their owner - Alice or Bob. The protocol is not secure, if the adversary is able to put her own padlock on the box, but stamp it with A or B, and thereby making it appear as if the padlock is Alice's or Bob's, respectively. This corresponds to the fact that the Diffie-Hellman protocol is only secure against an *eavesdropping adversary*, but insecure against a *MitM adversary*.

The critical property that facilitated the physical key exchange protocol, is that Alice can remove her padlock, even after Bob has added his own padlock. Namely, the 'padlock' operation is 'commutative' - it does not matter if Alice placed her padlock first and Bob second, she can still remove her padlock as is it was applied last. In a sense, the key to cryptographic key exchange protocols such as Diffie Hellman, is to perform a mathematical operation which is also commutative; of course, there are many commutative operations. We next discuss 'insecure prototypes' key-exchange protocols based on three commutative operations: addition, multiplication and XOR. However, before that, let us briefly discuss the *definition* of a secure key exchange protocol.

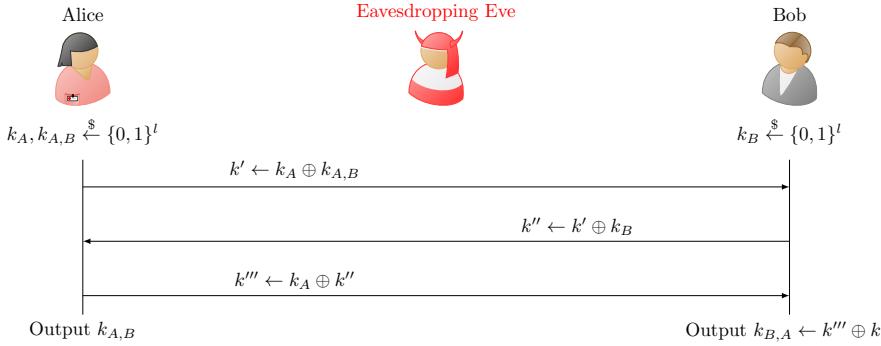


Figure 6.5: The (insecure) XOR Key Exchange Protocol; this protocol ensures correctness $k_{A,B} = k_{B,A}$, but is insecure. Specifically, by eavesdropping to the three exchanged messages (k' , k'' and k'''), Eve can find the key $k_{A,B}$. Can you find out how? See Exercise 6.2.

6.2.2 Some candidate key exchange protocols

In this subsection, we present few ‘prototype’ key-exchange protocols, which help us to properly explain the Diffie-Hellman protocol. Unlike the physical key exchange protocol of subsection 6.2.1, these are ‘real protocols’, i.e., involve only the exchange of messages - no physical objects or assumptions. We begin with three insecure ‘prototypes’, each using a different commutative operation: XOR, Addition and Multiplication.

The XOR, Addition and Multiplication key exchange protocols. The sequence diagram in Figure 6.5 presents the first prototype: the *XOR key exchange protocol*. This prototype tries to use the XOR operator, to ‘implement the padlocks’ of Figure 6.4.

XOR is a natural candidate, since we know that XOR can provide confidentiality when used ‘correctly’, e.g., in the one-time pad construction. Furthermore, XOR is commutative, and it is easy to see that this suffices to ensure the correctness of the XOR key exchange, i.e., the fact that $k_{A,B} = k_{B,A}$, as follows:

$$\begin{aligned}
 k_{B,A} &= k''' \oplus k_B \\
 &= (k'' \oplus k_A) \oplus k_B \\
 &= ((k' \oplus k_B) \oplus k_A) \oplus k_B \\
 &= (((k_{A,B} \oplus k_A) \oplus k_A) \oplus k_A \\
 &= k_{A,B}
 \end{aligned}$$

However, as the next exercise shows, the XOR key exchange protocol is insecure. In fact, not only it does not satisfy indistinguishability, but worse: an eavesdropper can easily find the exchanged key.

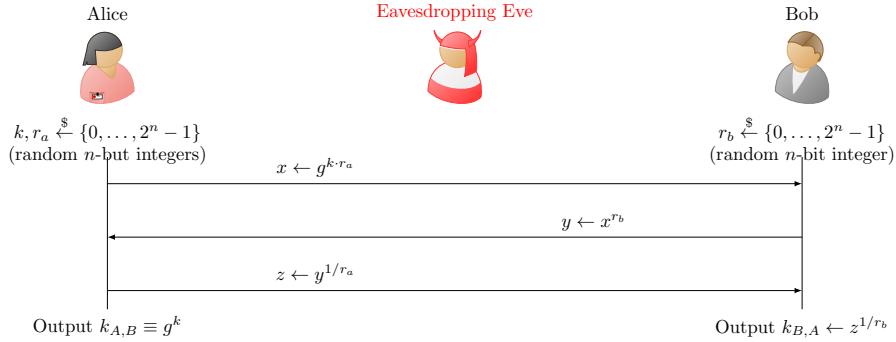


Figure 6.6: The (insecure and inefficient) Exponentiation Key Exchange Protocol, using some random integer g . If the resulting shared key $k_{A,B} = k_{B,A}$ is too long, use only some bits. This protocol, like the XOR key exchange protocol, ensures correctness $k_{A,B} = k_{B,A}$ but is insecure, as we show in the text.

Exercise 6.2 (XOR key exchange protocol is insecure). *Show how an eavesdropping adversary may find the secret key exchanged by the XOR Key Exchange protocol, by (only) using the values sent between the two parties.*

Solution (sketch): attacker XORs all three messages, to obtain: $k = (k \oplus k_A) \oplus (k \oplus k_A \oplus k_B) \oplus (k \oplus k_B)$. \square

Exponentiation key exchange. The attack on the XOR key exchange was due to the fact that the attacker was able to ‘remove’ elements by applying the XOR again, due to the combination of XOR’s commutativity and the fact that for XOR, every element is its own inverse, i.e., $(\forall x \in \{0, 1\}^l) x \oplus x = 0^l$.

So, let us try to use a different mathematical operation, that also ensures commutativity (for correctness), but where elements are (typically) not their own inverses: exponentiation. In Fig. 6.6, we show the resulting Exponentiation Key Exchange protocol. This protocol is obviously very inefficient, but let us ignore the inefficiency; we just present it to show its correctness and *vulnerability*, as motivation and to build intuition to the *modular exponentiation key exchange* protocol that we show afterwards.

Let us first show that the Exponentiation Key Exchange protocol ensures correctness, i.e., that $k_{A,B} = k_{B,A}$:

$$k_{B,A} = z^{1/r_b} \quad (6.12)$$

$$= \left(y^{1/r_a} \right)^{1/r_b} \quad (6.13)$$

$$= (x^{r_b})^{1/(r_b \cdot r_a)} \quad (6.14)$$

$$= (g^{k \cdot r_a})^{1/r_a} \quad (6.15)$$

$$= g^k = k_{A,B} \quad (6.16)$$

We relied on the commutativity of exponentiation in Equation 6.15.

Let us now explain an attack recovering the key $k_{A,B}$ exchanged, similarly to the attack on the XOR key exchange. The attack uses the fact that the exponentiation operation may be *removed* to find the exponent, by computing the inverse operation, i.e., logarithm (base g). The logarithm function is less efficient than exponentiation, but, over the integers or real numbers, it is still considered an efficient operation, since it can be computed in polynomial time.

Namely, an eavesdropper can simply compute the logarithm (base g) to remove the exponentiations from all flows, which reduces the protocol to the multiplication key exchange, shown insecure in Ex. 6.12 (and similarly to the attack on XOR key exchange in Exercise 6.2). Namely, the attacker applies the logarithm operator, with basis g , to the three messages of Fig. 6.6 - resulting in the values $k \cdot r_a$, $k \cdot r_a \cdot r_b$ and $k \cdot r_b$. The attacker can now combine these three values to find k , by computing $k = \frac{(k \cdot r_a) \cdot (k \cdot r_b)}{(k \cdot r_a \cdot r_b)}$.

Of course, this attack used the value of g . This is justified, since in a key exchange protocol, the parties do not have any preshared secret input (see Figure 6.2); indeed, if the parties already share a secret key, why not use it directly? Note that even if g is a preshared secret, the protocol is still vulnerable, with a modified attack (Exercise 6.13).

Exercise 6.12 shows that a similar vulnerability occurs if we use multiplication or addition instead of XOR or exponentiation. However, we will not give up - and we next show how we ‘fix’ this protocol, and finally present a protocol against which may be secure. Note that we do not claim that this protocol *is* secure; indeed, like other designs based on supposedly computationally-hard problem, a *proof* that the design is secure is unlikely - it would imply a proof that $P \neq NP$; see Section A.1.

Modular-Exponentiation Key Exchange. We now ‘fix’ the Exponentiation Key Exchange Protocol (Fig. 6.6). The attack against it used the fact that the computations in Fig. 6.6 are done over the field of the real (or natural) numbers - \mathbb{R} (or \mathbb{N}), where there are efficient algorithms to compute logarithms. This motivates changing this protocol, to use, instead, operations over a group in which the (discrete) logarithm problem is considered hard. Such groups exist, e.g., the ‘mod p ’ group, for a safe prime p .

We present this protocol in Fig. 6.7. Notice that this protocol uses multiplicative inverses in the $(p - 1)$ modular group, e.g., $a^{-1} \bmod (p - 1)$ is the number in $\mathbb{Z}_{p-1}^* \equiv \{1, \dots, p - 2\}$ such that: $a \cdot a^{-1} = 1 \bmod (p - 1)$.

The correctness of the Modular-Exponentiation Key Exchange Protocol follows from the commutativity of modular-exponentiation, much like the correctness of the preceding protocols:

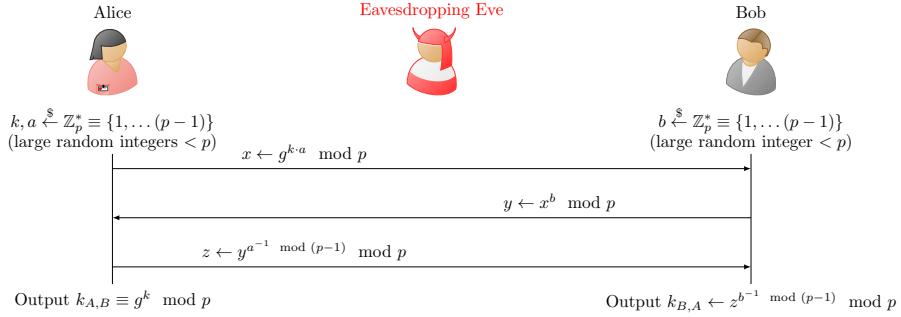


Figure 6.7: The Modular-Exponentiation Key Exchange Protocol, where p is a prime and g is a generator of \mathbb{Z}_p^* . The values k, a and b are chosen randomly from \mathbb{Z}_p^* , i.e., integers between 1 and $p - 1$ (why? see Exercise 6.14). Alice derives $k_{A,B} = g^k \pmod p$, and Bob derives $k_{B,A} = z^{b^{-1}} \pmod{(p-1)} \pmod p$; Equation 6.17 shows both derive the same key, i.e., $k_{A,B} = k_{B,A}$.

$$\begin{aligned}
k_{B,A} &= z^{b^{-1} \bmod (p-1)} \bmod p \\
&= \left(y^{a^{-1} \bmod (p-1)} \right)^{b^{-1} \bmod (p-1)} \bmod p \\
&= (x^b)^{b^{-1} \cdot a^{-1} \bmod (p-1)} \bmod p \\
&= (g^{k \cdot a})^{a^{-1} \bmod (p-1)} \bmod p \\
&= g^k \bmod p = k_{A,B}
\end{aligned} \tag{6.17}$$

Is this protocol secure, i.e., does it ensure indistinguishability? This may depend on the prime p used. One way to try to break the Modular-Exponentiation Key Exchange Protocol of Fig. 6.7, is to compute the (discrete) logarithm of the three values exchanged by the protocol - like the attack above against the ‘regular’ Exponentiation Key Exchange Protocol. This works when discrete logarithm can be computed efficiently, e.g., when $p - 1$ is *smooth*, i.e., has only small prime factors, e.g., $p = 2^x + 1$.

However, this attack requires computing discrete logarithms - and this is believed to be computationally-hard for certain modulus. In particular, discrete logarithm is assumed to be computationally hard when the modulus p is a large *safe prime*, i.e., $p = 2q + 1$ for some prime q ; see Definition 6.4 and Definition 6.3.

The attacker can easily detect if $a = 1$ or $b = 1$, and then find k . However, the probability of this choice is only $\frac{1}{p-2}$, which is exponentially small in the number of bits in p - i.e., negligible (for sufficiently large p). The attacker can also guess some specific value chosen by the parties, say $\tilde{a} \in \{1, \dots, p-1\}$ as a guess for a , compute $\tilde{a}^{-1} \bmod (p-1)$, and check if the guess for a was correct (i.e., whether $\tilde{a} = a \bmod p$), by comparing z to $y^{\tilde{a}^{-1}} \bmod p$. If the guess was correct, i.e., if $z = y^{\tilde{a}^{-1}} \bmod p$, then $\tilde{a} = a \bmod (p-1)$, and the attacker computes the key: $k_{A,B} = x^{a^{-1} \bmod (p-1)} \bmod p$. Note that there is

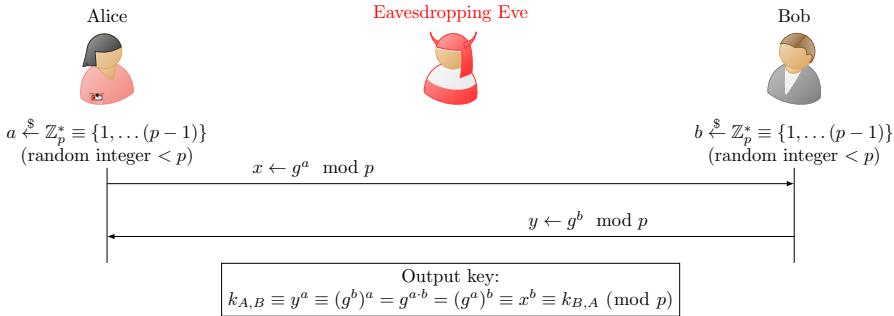


Figure 6.8: The Diffie-Hellman Key Exchange Protocol. The protocol uses \pmod{p} computations, where p is a prime. It is believed to be hard to compute the resulting key $g^{ab} \pmod{p}$, when p is a safe prime ($p = 2q + 1$ where q is a prime).

no advantage for the parties to select the a, b or k exponents from a larger set (not limited to $\{1, \dots, p - 1\}$); all the values sent by the protocol, as well as the key, will be exactly the same as when using the corresponding exponents $\pmod{p - 1}$.

In the following subsection, we present the Diffie-Hellman protocol - which is essentially an improved and simplified variant of the Modular-Exponentiation Key Exchange Protocol of Fig. 6.7.

6.2.3 The Diffie-Hellman Key Exchange Protocol and Hardness Assumptions

Fig. 6.8 presents the Diffie-Hellman (DH) key exchange protocol. The protocol uses the (safe) prime group \mathbb{Z}_p^* , i.e., using multiplications modulo p , where p is a (safe) prime p . The protocol assumes a given (public) choice of parameters: the (safe) prime p and the generator g . Recall that the order q of \mathbb{Z}_p^* is $q = p - 1$, i.e., $g^q \equiv 1 \pmod{p}$ and $\{1, \dots, p - 1\} = \{g^i\}_{i=1}^q$.

The protocol consists of only two flows: in the first flow, Alice sends $g^a \pmod{p}$, where $a \in \{1, \dots, p - 1\}$ is a *private key* chosen randomly by Alice; and in the second flow, Bob responds with $g^b \pmod{p}$, where b is a private key chosen randomly by Bob. The result of the protocol is a shared secret value $g^{ab} \pmod{p}$, computed by Alice as $k_{A,B} = (g^b \pmod{p})^a \pmod{p} = g^{ba} \pmod{p}$, and by Bob as $k_{B,A} = (g^a \pmod{p})^b \pmod{p} = g^{ab} \pmod{p}$.

The Diffie-Hellman key-exchange protocol is, essentially, a simplified, and slightly optimized, variant of the Modular-Exponentiation Key Exchange Protocol of Fig. 6.7; and in particular, the security of both protocols relies on the difficulty of computing discrete logarithms, and may fail if p has only small factors; the choice of *safe primes* ($p = 2q + 1$ for prime q) is hoped to be ‘safe’, i.e., to ensure security.

The basic difference between the two protocols is that in the Diffie-Hellman protocol, the key output is not $g^k \pmod p$ for some random k , as happens for the Exponentiation key exchange. Instead, the key being output (exchanged) is g^{ab} . This makes the protocol a bit simpler, and more efficient: only two flows instead of three, no need to compute inverses ($a^{-1}, b^{-1} \pmod p$), and one less exponentiation.

The correctness of the Diffie-Hellman key exchange protocol, i.e., the fact that $k_{A,B} = k_{B,A}$, follows from the commutativity of exponentiation (and modular exponentiation), as follows:

$$k_{A,B} \equiv y^a \pmod p \quad (6.18)$$

$$\equiv (g^b)^a \pmod p \quad (6.19)$$

$$\equiv g^{ab} \pmod p \quad (6.20)$$

$$\equiv x^b \pmod p \quad (6.21)$$

$$= k_{B,A} \quad (6.22)$$

The following exercise may help to get a better feeling for the protocol and how it works.

Exercise 6.3 (Diffie-Hellman (DH) Key Exchange). *Let $p = 7$.*

1. *Is $p = 7$ a safe prime?*
2. *Find a generator g for \mathbb{Z}_p^* ; show that g is a generator and how you found it.*
3. *Alice and Bob run the DH protocol with the prime p and generator g . Alice selects $a = 3$, and Bob selects $b = 6$. Compute the values sent by Alice and Bob, and show the computation of the shared key by each of them, resulting in the same value.*

DH is vulnerable to MitM attacker. Both the Diffie-Hellman and the Modular-Exponentiation key exchange protocols insecure against a MitM attacker; they are designed only against an *eavesdropping adversary*. In fact, as shown in Figure 6.9, all a MitM attacker needs to do is to fake the message from a party, allowing it to impersonate that party (establishing a shared key with the other party). Indeed, in practice, we (almost) always use *authenticated* variants of the DH protocol, as we discuss in subsection 6.3.1.

Security of DH and the Computational DH Assumption. Ok, so the DH protocol is vulnerable against MitM; but can we safely use it against an *eavesdropper*? Namely, can we assume that the key output by the DH protocol cannot be computed by an eavesdropping adversary, when DH is computed over a group in which discrete logarithm is assumed to be a computationally-hard problem, e.g., the ‘mod p ’ group where p is a safe prime? So far, this has not

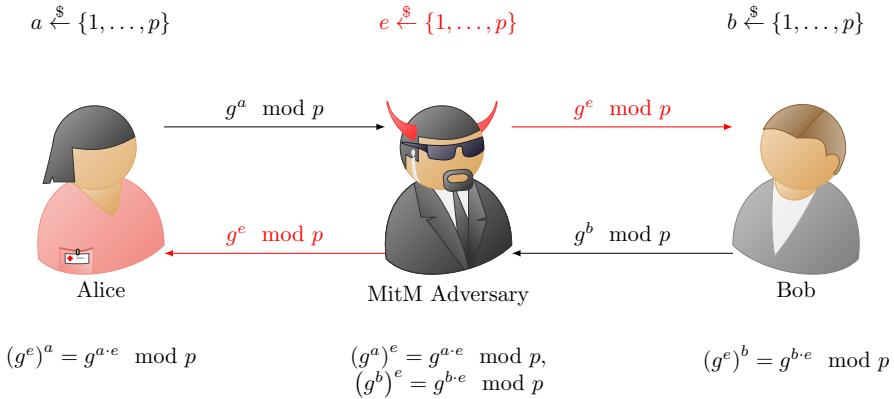


Figure 6.9: MitM attack on the DH key-exchange protocol. The DH protocol is believed to be secure against an eavesdropping adversary - or if the messages are authenticated.

been *proven*; there is no proof that if DH is computed (using a safe prime p), then the resulting key cannot be efficiently computed by an adversary. There isn't even a proof showing that such attack against DH would imply an efficient method to compute discrete logarithms (modulo a safe prime p). In fact, these are still important open questions.

The common approach is to *assume* that when DH protocol is run using safe prime p , then an eavesdropping adversary cannot guess the resulting shared key. This assumption, which is stronger than the assumption of hardness of discrete-log, is called the *Computational DH (CDH)* assumption. The CDH assumption essentially means that it is infeasible to compute the DH shared secret $g^{ab} \bmod p$, given the two exchanged values $g^a \bmod p$ and $g^b \bmod p$, if p is a sufficiently-long safe prime (i.e., $p = 2q + 1$ for a prime q). Both the DH and the discrete-log problems are easy for some other values of p , for a ‘smooth’ p , i.e., when $p - 1$ has only small factors, e.g., $p - 1 = 2^i$ for some integer i .

Definition 6.6 (*Computational DH (CDH) for safe prime groups*). *The Computational DH (CDH) assumption for safe prime groups holds, if there is no efficient (PPT) adversary \mathcal{A} that, given a random n -bit safe prime p (i.e., $p = 2q + 1$ for prime q) and generator g , and the values $(g^a \bmod p, g^b \bmod p)$ for random $a, b \xleftarrow{\$} \{1, \dots, p - 1\}$, returns, with non-negligible probability, $g^{ab} \bmod p$.*

Namely, for every PPT algorithm \mathcal{A} and random n -bit safe prime p and

generator g holds:

$$\Pr_{\substack{a,b \in \mathbb{Z}_p^*}} [\mathcal{A}(g^a \bmod p, g^b \bmod p) = g^{ab} \bmod p] \in NEGL(n) \quad (6.23)$$

Note that the definition allows the random choice of $a = 1$ (or $b = 1$), although for $a = 1$ holds $g^{ab} = g^b$. However, the number of possible values in \mathbb{Z}_p^* is exponential in n , i.e., the probability of such ‘bad choice’ is negligible.

At least one bit of $g^{ab} \bmod p$ is exposed! Even assuming that the CDH assumption holds for safe prime groups, an eavesdropper is still be able to learn (at least) one bit about $g^{ab} \bmod p$. Specifically, an attacker, observing $g^a \bmod p$ and $g^b \bmod p$ from a run of the Diffie-Hellman protocol, can efficiently find whether $g^{ab} \bmod p$ is a *quadratic residue* modulo p , i.e., if there exists some $z \in \mathbb{Z}_p^*$ such that $g^{ab} \equiv z^2 \bmod p$ (Definition 6.5). Let us show how.

Claim 6.3. *Let p be a prime, g be a generator for \mathbb{Z}_p^* , a, b be integers, and $y \equiv g^{ab} \bmod p$. Given $g^a \bmod p$ and $g^b \bmod p$, we can efficiently deduce if y is a quadratic residue modulo p .*

Proof: From Claim 6.1, we can efficiently find if $g^a \bmod p$ and $g^b \bmod p$ are quadratic residues modulo p . From Claim 6.2, this gives the least significant bit (parity) of a and of b ; obviously, ab is even if either a or b is even. Again from Claim 6.2, the least significant bit of ab indicates the quadratic residuosity of $g^{ab} \bmod p$. \square

6.2.4 Secure derivation of keys from the DH protocol

An eavesdropper to the DH key exchange can observe $g^a \bmod p$ and $g^b \bmod p$; hence, from Claim 6.3, the attacker can know if $y \equiv g^{ab} \bmod p$ is a quadratic residue modulo p . Therefore, using $y \equiv g^{ab} \bmod p$ directly as a key may not be advisable, as even assuming that the CDH assumption is true, still an eavesdropper can learn partial information about y (i.e., if it is a quadratic residue). Notice that while we show only exposure of this information - the quadratic residuosity of $g^{ab} \bmod p$ - there could be ways to expose more³ information without violating the CDH assumption.

So, how can we use the DH protocol to securely exchange a key? One could simply ignore this concern; but let us discuss two other, more prudent, options.

First option: generalized DH protocol, using DDH groups. The generalized DH protocol can ensure that the value of the derived key k_{AB} is secret, without any leakage. This protocol uses (and requires) a cyclic group \mathbb{G} where the (stronger) *Decisional DH (DDH) Assumption* is believed to hold. Let us first define this assumption (a bit informally).

³It may possible to expose even %80 of the bits [65].

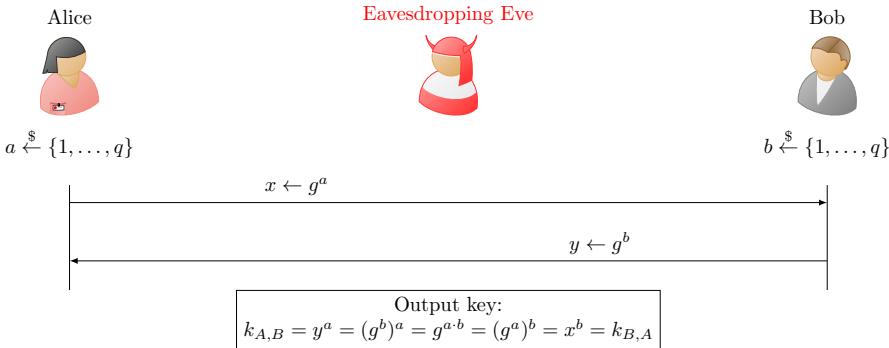


Figure 6.10: The Generalized Diffie-Hellman Key Exchange Protocol, for group \mathbb{G} with order q . All operations are group operations, denoted like the usual multiplication notation. For some types of groups, and sufficiently-large order q , it is believed that it is infeasible not only to compute g^{ab} but even to distinguish between g^{ab} and a random group member, i.e., *DDH* security. The protocol reduces to the ‘regular’ (mod- p) Diffie-Hellman protocol, when the group \mathbb{G} is \mathbb{Z}_p^* , the modular- p group for prime p .

Definition 6.7 (The Decisional DH (DDH) Assumption). *Group \mathbb{G} , with order q , satisfies the Decisional DH (DDH) Assumption if there is no PPT algorithm \mathcal{A} that can distinguish, with significant advantage compared to guessing, between $[g^a, g^b, g^{ab}]$ and $[g^a, g^b, g^c]$, for a, b and c selected randomly from $\{1, \dots, q\}$. A group for which the DDH assumption is believed to hold is called a DDH group.*

In Figure 6.10, we present the Generalized Diffie-Hellman protocol, using cyclic group \mathbb{G} , rather than the specific group \mathbb{Z}_p^* used in the original DH protocol. We use the group operation of \mathbb{G} , denoted as multiplication, in lieu of the mod- p multiplication used by \mathbb{Z}_p^* . The protocol ensure key secrecy, if \mathbb{G} is a *DDH group*, i.e., if the DDH problem is believed to be computationally infeasible (‘hard’) for \mathbb{G} .

The generalized DH protocol assumes agreed-upon DDH group \mathbb{G} and generator g , and known order q for \mathbb{G} . Like the original DH protocol (Figure 6.8), the protocol has only two flows. In the first flow, Alice sends g^a , where $a \in \{1, \dots, q\}$ is a secret chosen randomly by Alice; and in the second flow, Bob responds with g^b , where $b \in \{1, \dots, q\}$ is a secret chosen randomly by Bob. Both ‘exponentiations’ (g^a and g^b) are done by repeatedly applying the group operation (instead of modular multiplication, as in the original DH protocol).

The result of the protocol is a shared secret value g^{ab} , computed by Alice as $k_{A,B} = (g^b)^a = g^{ba}$, and by Bob as $k_{B,A} = (g^a)^b = g^{ab}$. All ‘exponentiations’ are repeated application of the group operation of \mathbb{G} . Notice that the secret value exchanged, g^{ab} , is an element of the group \mathbb{G} , i.e., it is not a uniformly random string; this requires mapping of g^{ab} into a random string.

One group where DDH is assumed to hold is \mathbb{Q}_p , the subgroup of \mathbb{Z}_p^* consisting of the quadratic residues in \mathbb{Z}_p^* , for a safe prime p (i.e., $p = 2q + 1$,

for prime q). Certain elliptic-curve groups are also believed to be DDH groups. See other examples in [65].

Second option: extract a secret, random key from the partially-random $g^{ab} \bmod p$. The second option is to use the DH protocol as described, i.e., with a safe prime group \mathbb{Z}_p^* , but to securely *extract* (derive) a shared key k from $g^{ab} \bmod p$. Section 3.5 discusses the two common ways to extract a shared key from a mostly-random shared secret data: using either a *randomness extractor hash function* or a *Key Derivation Function (KDF)*. This approach requires, basically, that the $g^{ab} \bmod p$ contains a ‘sufficient’ randomization, to ensure that the output of the extractor or KDF is pseudorandom - basically, a variant of DDH with respect to the specific extractor or KDF used.

6.3 Using DH for Resiliency to Exposures: the (PFS) Auth-h-DH and (PRS) DH-Ratchet protocols

As discussed above, and demonstrated in Ex. 6.9, the DH protocol is vulnerable to a MitM attacker; its security is only against a passive, eavesdropping-only attacker. In most practical scenarios, attackers who are able to eavesdrop, have some or complete ability to also perform active attacks such as message injection; it may seem that DH is only applicable to the relatively few scenarios of eavesdropping-only attackers.

In this section, we discuss extensions of the DH protocol, extensively in practice to improve resiliency to adversaries which have MitM abilities, combined with key-exposure abilities. Specifically, these extensions allow us to ensure *Perfect Forward Secrecy (PFS)* and *Perfect Recover Security (PRS)*, the two strongest notions of resiliency to key exposures of secure key setup protocols as presented in Table 5.2 (Section 5.7).

6.3.1 The Authenticated DH (Auth-h-DH) protocol: ensuring Perfect Forward Secrecy (PFS)

Assuming that the parties share a secret master key MK , it is quite easy to extend the DH protocol in order to protect against MitM attackers. All that is required is to use a *Message Authentication Code (MAC)* scheme to authenticate the DH flows. To ensure security without requiring the use of DDH group (i.e., the DDH assumption), we may extract the key using an extractor hash function h (or a KDF). See Fig. 6.11, showing the Auth-h-DH, the resulting *authenticated* variant of the DH protocol using hash h to extract the key.

Correctness follows similarly to the argument for ‘regular’ Diffie–Hellman protocol:

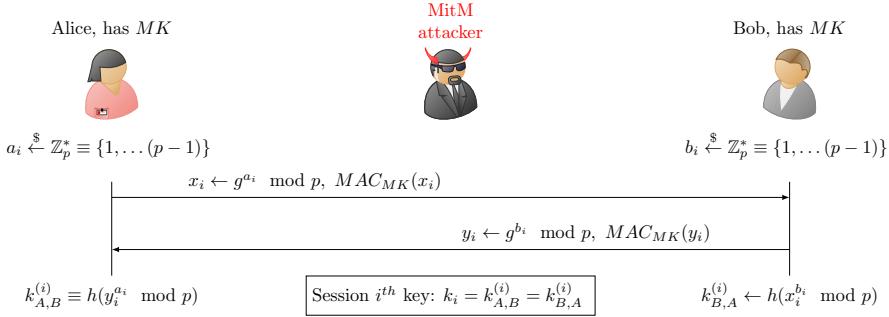


Figure 6.11: The Auth-*h*-DH Protocol, showing *i*th exchange. This protocol is secure against MitM attackers; furthermore, it ensures *Perfect Forward Secrecy* (*PFS*), i.e., exposure of current keys does not expose past keys. The protocol, as presented, uses both a MAC function (*MAC*) and a keyless randomness extractor hash function *h*.

$$k_i \equiv k_{A,B}^{(i)} \equiv h(y_i^{a_i} \pmod p) \quad (6.24)$$

$$= h((g^{b_i})^{a_i} \pmod p) \quad (6.25)$$

$$= h((g^{b_i})^{a_i} \pmod p) \quad (6.26)$$

$$= h((g^{a_i})^{b_i} \pmod p) \quad (6.27)$$

$$= h(x_i^{b_i} \pmod p) \quad (6.28)$$

$$\equiv k_{B,A}^{(i)} \quad (6.29)$$

The next informal lemma presents the security properties of Auth-*h*-DH. We only give an informal argument for the validity of the Lemma.

Lemma 6.1 (Informal Auth-*h*-DH). *Assuming the extended CDH assumption, the Auth-*h*-DH protocol ensures indistinguishability and PFS.] The Auth-*h*-DH protocol (Fig. 6.11) ensures secure key-setup (indistinguishability) and Perfect Forward Secrecy (PFS), provided that MAC is a secure MAC function and that *h* is a randomness extractor hash function.*

Argument: Let us first consider a run where the key *MK* is unknown to the attacker, i.e., was not exposed. Hence, since *MAC* is a secure *Message Authentication Code* (*MAC*), then the protocol outputs a key only if the message it receives was sent by the peer (or by itself); in any case, the exponent used (*a_i* or *b_i*) was selected randomly by Alice or Bob and unknown to the attacker. From the *extended CDH* assumption (subsection 6.2.4), and the assumption that *h* is a randomness extractor hash, it follows that the session key *k_i* (set by Alice to *k_{A,B}⁽ⁱ⁾*, and by Bob to *k_{B,A}⁽ⁱ⁾*) is indistinguishable from random, i.e., the

Auth- h -DH protocol ensures secure key-setup. Note that the protocol cannot ensure that both parties will generate the same k_i , since a MitM attacker may change messages, or simply drop one of the messages (causing the key to be output by only one party).

The PFS property also follows, since it requires that key k_i is secure, even if MK is exposed, as long as the exposure occurs *after* session i was completed; and our analysis did not exclude such exposure *after* the session was over. \square

KDF-based variants of the Auth- h -DH protocol. As mentioned in Section 3.5, keyless generic extractors do not exist, and it is preferable to avoid their use and rely on a *Key Derivation Function (KDF)*, which can also output as many pseudorandom bits as needed. Let us discuss briefly two variants of the Auth- h -DH protocol, which use a KDF instead of a (keyless) extractor.

Variant 1: a two-keyed KDF-based variant of Auth- h -DH. The first variant simply replaces the keyless randomness extractor hash h of Figure 6.11, with the KDF-extract function, i.e., the key is derived as KDF_{salt} , where $salt$ is a random and non-secret (known) key, which does not expose the master key MK used by the MAC function.

Variant 2: a secure variant of Auth- h -DH, using a combined KDF/PRF. Another variant of Auth- h -DH uses the same function f and the same master key MK to replace both the MAC function and the KDF function. This variant requires f to satisfy both the MAC and the KDF functionalities. This is a stronger assumption, but it makes the design simpler and more efficient; therefore, it is often preferred.

See Exercise 6.19 for questions related to these and other variants of the Auth- h -DH protocol. And here is a more basic exercise about the Auth- h -DH protocol.

Exercise 6.4. Alice and Bob share master key MK and perform the Auth- h -DH protocol daily, at the beginning of every day i , to set up a ‘daily key’ k_i for day i . Assume that Mal can eavesdrop on communication between Alice and Bob every day, but perform MitM attacks only every even day (i s.t. $i \equiv 0 \pmod{2}$). Assume further that Mal is given the master key MK , on the fifth day. Could Mal decipher messages sent during day i , for $i = 1, \dots, 10$? Write your responses in a table.

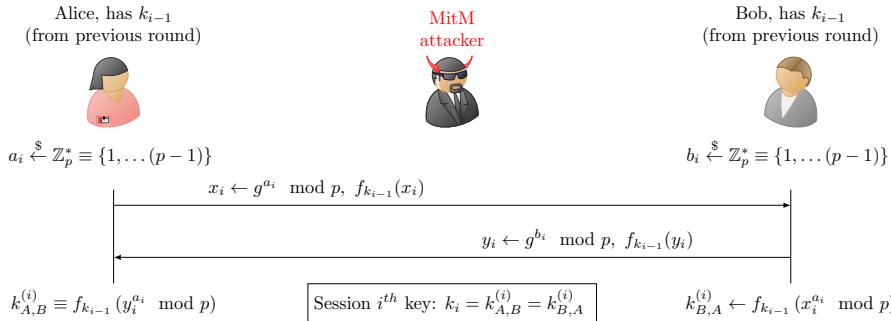
Note that the results of Ex. 6.4 imply that the Auth- h -DH protocol does *not* ensure the *Recover Security* property. We next show extensions that improve resiliency to key exposures, and specifically recover security after exposure, provided that the attacker does not deploy the MitM ability for one handshake.

6.3.2 The DH-Ratchet protocol: Perfect Forward Secrecy (PFS) and Perfect Recover Security (PRS)

The Auth- h -DH protocol ensures perfect forward secrecy (PFS), but does *not* ensure recover security, and definitely not perfect recover security (PRS).

Protocol	Section	Secure key setup	Forward secrecy (FS)	Perfect Forward Secrecy (PFS)	Recover Security (RS)	Perfect Recover Security (PRS)
2PP-Key Exchange	5.4.1	✓	✗	✗	✗	✗
FS-ratchet	5.7.1	✓	✓	✗	✗	✗
RS-ratchet	5.7.2	✓	✓	✗	✓	✗
Auth-h-DH	6.3.1	✓	✓	✓	✗	✗
DH-ratchet	6.3.2	✓	✓	✓	✓	✓

Table 6.2: Resiliency to key exposures of Key Exchange protocols.

Figure 6.12: The DH-Ratchet key exchange protocol, ensuring PFS and PRS against MitM attacker, assuming that f is both a PRF and KDF. The figure shows round i of the protocol.

In fact, a single key exposure, at some point in time, suffices to make all future handshakes vulnerable to a MitM attacker - even if there has been some intermediate handshakes without (any) attacks, i.e., during which the attacker had neither MitM nor eavesdropper capabilities. To see that the Auth-h-DH protocol does not ensure recover security, see Exercise 6.4.

Note that the (shared key) RS-Ratchet protocol presented in subsection 5.7.2 (Fig. 5.21), achieved recovery of security - albeit, not Perfect Recover Security (PRS). Namely, the Auth-h-DH protocol does not even strictly improve resiliency compared to the RS-Ratchet protocol (subsection 5.7.2); see Table 6.2.

In this subsection we show how to achieve *both* PFS and Perfect Recover Security (PRS). Specifically, we present the *DH-Ratchet* protocol, as illustrated in Fig. 6.12, which ensures *both* PFS and PRS. This protocol uses a function f , which is assumed to be *simultaneously* both a PRF and a KDF; this is similar to one of the variants of the Auth-h-DH protocol, discussed in subsection 6.3.1.

Like the Auth-h-DH protocol presented above, the DH-ratchet protocol also authenticates the DH exchange; hence, as long as the authentication key is unknown to the attacker *at the time when the protocol is run*, then the key exchanged by the protocol is secret. The improvement, compared to the Auth-

h-DH protocol, is in the key used to authenticate the DH exchange; instead of using a fixed master key (*MK*) as done by the Auth-*h*-DH protocol (Fig. 6.11), the DH-ratchet protocol authenticates the i^{th} DH exchange using the *session key* exchanged in the *previous* round (exchange), i.e., k_{i-1} . An initial shared secret key k_0 is used to authenticate the first round, i.e., $i = 1$.

Lemma 6.2 (DH-Ratchet ensures PFS and PRS). *The DH-Ratchet protocol (Fig. 6.12) ensures secure key-setup with perfect forward secrecy (PFS) and perfect recovery security (PRS), assuming that f is both a PRF and KDF.*

Sketch of proof: The PFS property follows, like in Lemma 6.1, from the fact that k_i , the session key exchanged during session i , depends on the result of the DH protocol, i.e., is secure against an eavesdropping-only adversary. The protocol also ensures secure key setup, since a MitM adversary cannot learn k_{i-1} and hence cannot forge the DH messages.

The PRS property follows from the fact that if at some session i' there is only an eavesdropping adversary, then the resulting key $k_{i'}$ is secure, i.e., unknown to the attacker, since this is assured when running DH against an eavesdropping-only adversary. It follows that in the following session ($i'+1$), the key used for authentication is unknown to the attacker, hence the execution is again secure - and results in a new key $k_{i'+1}$ which is again secure (unknown to attacker). This continues, by induction, as long as the attacker is not (somehow) given the key k_i to some session i , before the parties receive the messages of the following session $i+1$. \square

Many instant messaging applications use a slightly more advanced version of the DH-Ratchet protocol, usually referred to as the *Double Ratchet* protocol (or algorithm). The Double-Ratchet protocol does not use the DH-derived keys k_i directly to protect the traffic; instead, it derives from k_i a series of keys used to protect the traffic. The standard double-ratchet protocol is also *asynchronous*, allowing the two parties to change keys independently and without dependency on time synchronization. The following exercise presents the *Synchronous Double-Ratchet protocol*, a slightly simplified example of the Double-Ratchet protocol which retains much of its security benefits but assumes synchronized clocks.

Exercise 6.5 (The Synchronous Double-Ratchet protocol). *Alice and Bob use low-energy devices to communicate. To ensure secrecy, they run, daily, the DH-Ratchet protocol (Fig. 6.12), but want to further improve security, by changing keys every hour. However, to save energy and time, the hourly process should use only very efficient computations - and no exponentiations. Let k_i^j denote the key they share after the j^{th} hour of the i^{th} day, where $k_i^0 = k_i$ (the key exchanged in the ‘daily exchange’ of Fig. 6.12).*

1. *Show how Alice and Bob should set their hourly shared secret key k_i^j .*
2. *Identify the security benefits of your solution, compared to the ‘regular’ DH-Ratchet protocol.*

Solution:

1. $k_i^j = f_{k_i^{j-1}}(1)$ (the value 1 is arbitrary of course).
2. The protocol uses k_i^j as the ‘session key’, i.e., the key used to protect the traffic in the j^{th} hour of the i^{th} day. Assume the first hour of the day is numbered $j = 1$. The advantage is that exposure of k_i^j , for any hour $j > 0$, does not expose the ‘ratchet master key’ of that day $k_i = k_i^0$. Hence, such exposure only exposes traffic sent during this hour but not traffic sent in any other hour (or day). This is in contrast with the DH-Ratchet protocol, where exposure of the session key k_i , used throughout day i , exposes all the traffic of day i , and furthermore, exposes future traffic until the key is recovered (in a day in which the attacker does not eavesdrop).

6.4 The DH and El-Gamal Public Key Cryptosystems

In this section, we discuss two related public-key cryptosystems, both based on the discrete-logarithm problem: the (well-known) El-Gamal PKCS and the Diffie-Hellman (DH) PKCS, which is basically a transformation of the DH key exchange protocol into a public-key cryptosystem. In the next section (Section 6.5), we present a third system, the (well-known) RSA public key cryptosystem.

6.4.1 The DH PKC and the Hashed DH PKC

In their seminal paper [101], Diffie and Hellman presented the concept of public-key cryptosystems - but did not present an implementation. On the other hand, they did present the (Figure 6.8). We next show that a minor tweak allows us to turn the DH key exchange protocol into a PKC; we accordingly refer to this PKC as the DH PKC, and a variant of it that also uses a hash function h as the DH- h PKC.

The DH PKC. Let us first present the *DH public key cryptosystem (DH PKC)*, illustrated in Figure 6.13. As can be seen, this public key cryptosystem is essentially an adaptation of the DH key exchange protocol (Figure 6.8), using safe prime p . Essentially, instead of Alice selecting random secret a and sending $g^a \bmod p$ to Bob in the first flow of the DH protocol, Alice selects a *fixed private key* d_A , exactly in the same way, i.e., $d_A \xleftarrow{\$} \{1, \dots, p-1\}$. Next, Alice computes her *public key* e_A , as: $e_A \equiv g^{d_A} \bmod p$.

Bob encrypts a message m using Alice’s public key e_A by selecting a random value $b \in [2, p-2]$, and computing $e_A^b \bmod p$, and sending the pair to Alice; note that this is essentially Bob’s role in the DH protocol. Namely, Bob computes the ciphertext according to Equation 6.30:

$$E_{e_A}(m) = \begin{cases} b \xleftarrow{\$} [1, p-1] \\ \text{Return } (g^b \bmod p, m \oplus (e_A)^b \bmod p) \end{cases} \quad (6.30)$$

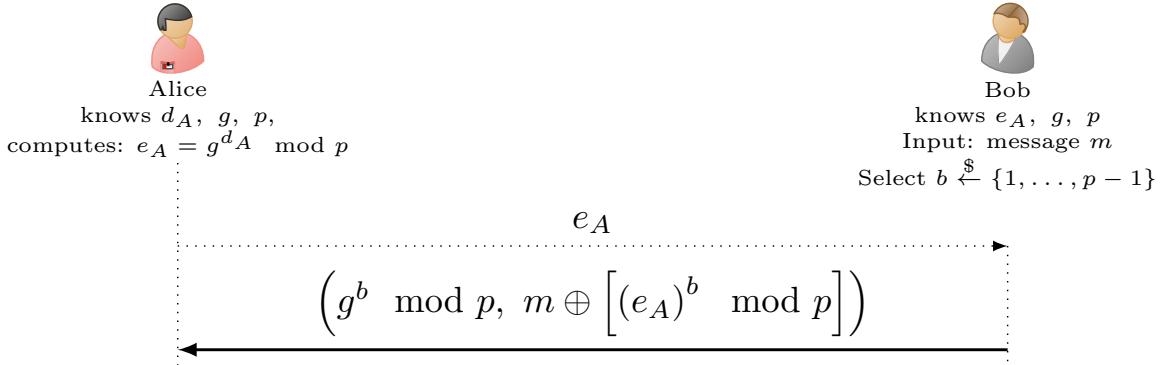


Figure 6.13: The DH public key cryptosystem (DH PKC). Bob encrypts plaintext message m , using Alice's public key e_A and the public parameters: a safe prime p and generator g of the group \mathbb{Z}_p^* . The ciphertext consists of the *pair of strings* $(g^b \bmod p, m \oplus [(e_A)^b \bmod p])$. The value b is selected randomly and known only to Bob, who should select a new b for each encryption.

Notice that *the ciphertext is a pair of values*.

Upon receiving such a ciphertext, which we denote (c_b, c_m) , Alice can decrypt it by computing:

$$D_{d_A}(c_b, c_m) = c_m \oplus [(c_b)^{d_A} \bmod p] \quad (6.31)$$

To see that the DH PKC ensures correctness, i.e., that decryption recovers the plaintext, we observe that:

$$\begin{aligned} D_{d_A}(E_{e_A}(m)) &= D_{d_A}\left(g^b \bmod p, m \oplus (e_A)^b \bmod p\right) \\ &= \left(m \oplus (e_A)^b \bmod p\right) \oplus \left[(g^b \bmod p)^{d_A} \bmod p\right] \\ &= m \oplus (g^{b \cdot d_A} \bmod p) \oplus (g^{b \cdot d_A} \bmod p) \\ &= m \end{aligned}$$

The security of DH PKC and the Hashed DH PKC. Intuitively, the security of DH PKC seems to follow from the *CDH* assumption (Definition 6.6). Let us present a *reduction* argument which supports this intuition. Assume an attacker \mathcal{A}^{DHPKC} is able to learn a random message m from e_A and $E_{e_A}(m)$. Then we can design an attacker \mathcal{A}^{CDH} that will be able to compute $g^{ab} \bmod p$, given $g^a \bmod p$ and $g^b \bmod p$, as follows. Given $g^a \bmod p$ and $g^b \bmod p$, the attacker \mathcal{A}^{CDH} will define $e_A = g^a \bmod p$ and $c_b = g^b \bmod p$, and select a random message m and compute $c_m = m \oplus (g^a \bmod p)^b \bmod p$.

However, this argument has a flaw; even if the CDH assumption holds, the DH PKC may not be a secure encryption. Specifically, the argument was based on the assumption that the attacker \mathcal{A}^{DHPKC} is able to learn the message m .

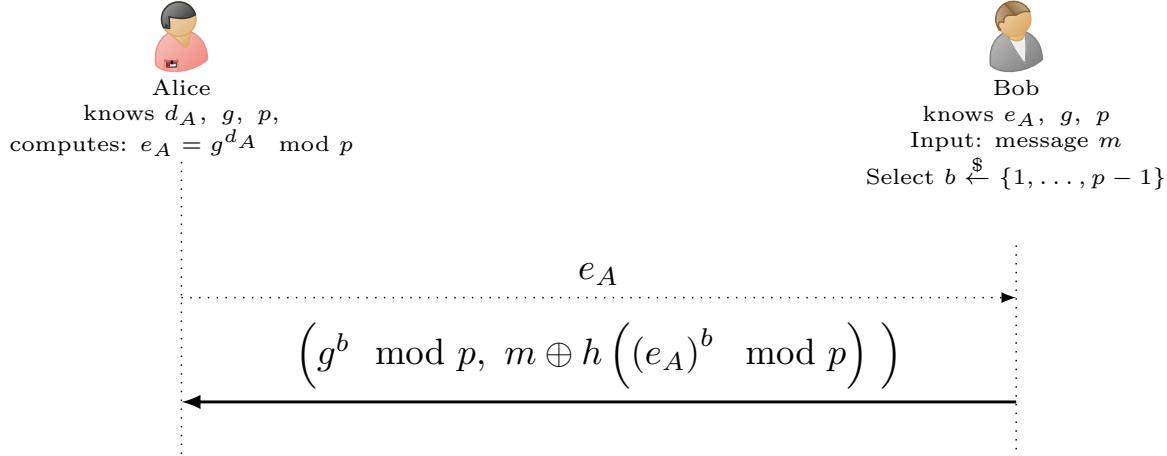


Figure 6.14: The Hashed DH Public Key Cryptosystem (Hashed DH PKC): same as the DH PKC (Figure 6.13, except for hashing the ‘one-time key’ $e_A^b \pmod{p}$).

However, a secure encryption scheme should also prevent *disclosure of partial information* about the plaintext, as formalized by the *indistinguishability test for public key cryptosystems, PKC IND-CPA* (Definition 2.11).

In fact, Claim 6.3 shows that an attacker may learn partial information about $g^{ab} \pmod{p}$ from the public values $g^a \pmod{p}$ and $g^b \pmod{p}$ - even if the CDH assumption holds. This may, therefore, expose partial information about m when using the DH public key cryptosystem (DH-PKC) as presented in Figure 6.13.

One solution to this is to modify the design. Recall that DH-PKC uses the \pmod{p} group (with a safe prime p). We could, instead, use a *DDH group* (Definition 6.7), i.e., a cyclic group \mathbb{G} that is believed to *hide all partial information*, as in subsection 6.2.4.

Fig. 6.14 presents a different solution: the *Hashed DH PKC*. In the Hashed DH PKC, we apply a cryptographic hash function h to e_A^b , before XORing it with the message. Namely, we compute $c_m = m \oplus h(e_A^b \pmod{p})$. Specifically, this should be a *randomness-extractor hash* function. The output of a randomness-extractor hash h should be indistinguishable from random; hence, it should hide all partial information about the key. The ciphertext is, therefore, the pair:

$$E_{e_A}(m) \equiv (g^b \pmod{p}, m \oplus h((e_A)^b \pmod{p}))$$

Yet another variant of the DH PKC uses a keyed key-derivation function $KDF_s(g^{d_A b} \pmod{p})$, with a uniformly-random key/salt s . See discussion of randomness extractor hash functions and key derivation functions KDF in subsection 3.5.3.

Exercise 6.6. Show the correctness of the DH-h PKC.

6.4.2 The El-Gamal PKC

The El-Gamal PKC is another encryption scheme based on the *DH key exchange protocol*, which is closely related to the DH PKC. As for the DH PKC, we discuss three variants of the El-Gamal PKC: using the $\text{mod } p$ group (for safe prime p), using DDH group, and *Hashed El-Gamal*.

The original design of the El-Gamal PKC, in [133], uses multiplications $\text{mod } p$ where p is a safe prime, like the DH PKC. Key generation is also done as in DH PKC, i.e., Alice's selects its private key randomly as $d_A \xleftarrow{\$} \{2, \dots, p-1\}$ and computes her *public key* e_A as: $e_A \equiv g^{d_A} \pmod{p}$. Even the encryption process is similar to DH PKC: Bob selects a random value $b \in [2, p-1]$, and computes two values: $c_b \leftarrow g^b \pmod{p}$ and e_A^b .

$$E_{e_A}(m) = (g^b \pmod{p}, e_A^b \pmod{p}) \quad (6.32)$$

The difference between the original El-Gamal PKC and the DH PKC is in how Bob *uses* the e_A^b value to encrypt the message m . In the original El-Gamal PKC, Bob *multiples* m , i.e., computes $c_m \leftarrow m \cdot e_A^b \pmod{p}$, while in DH PKC, Bob uses exclusive-or, i.e., computes $c_m \leftarrow m \oplus e_A^b$. Decryption is also modified accordingly, by using (modular) division instead of exclusive-or, i.e.:

$$D_{d_A}(c_b, c_m) = \frac{c_m}{c_b^{d_A}} \pmod{p} = c_m \cdot c_b^{-d_A} \pmod{p} \quad (6.33)$$

Correctness follows similarly to DH PKC.

Unfortunately, similarly to DH PKC, the original El-Gamal PKC may expose partial information about the plaintext message m . And, like for DH PKC, there are two solutions, both similar to the corresponding DH-PKC solution: *Hashed El-Gamal* or using El-Gamal with a *DDH* group (Definition 6.7).

The first solution, Hashed El-Gamal, is similar to hashed DH PKC. Namely, in the encryption process, we hash the ‘one-time pad’ e_A^b before using it to hide the message m . Namely, encryption is, as usual, a pair (c_b, c_m) , except c_m is computed as: $c_m \leftarrow m \oplus h(e_A^b)$. We also compute the hash for decryption:

$$D_{d_A}(c_b, c_m) = \frac{c_m}{h(c_b^{d_A})} \pmod{p} = c_m \cdot \left(h(c_b^{d_A}) \right)^{-1} \pmod{p} \quad (6.34)$$

The second solution, using El-Gamal with a *DDH* group, is similar to the use of a DDH group with the DH PKC. Namely, computations are done over a cyclic group \mathbb{G} believed to ensure the DDH *assumption* (Definition 6.7). Using the usual convention where we denote the operation of group \mathbb{G} in the same ways that we normally denote multiplication over the reals (or integers), the encrypt and decrypt operations of El-Gamal become even a bit simpler (compared to Equation 6.32 and Equation 6.33):

$$E_{e_A}(m) = (g^b, e_A^b) ; D_{d_A}(c_b, c_m) = c_m \cdot c_b^{-d_A} \quad (6.35)$$

The El-Gamal with a *DDH* group is believed to be secure, i.e., to prevent disclosure of any partial information about the plaintext. In addition, the DDH

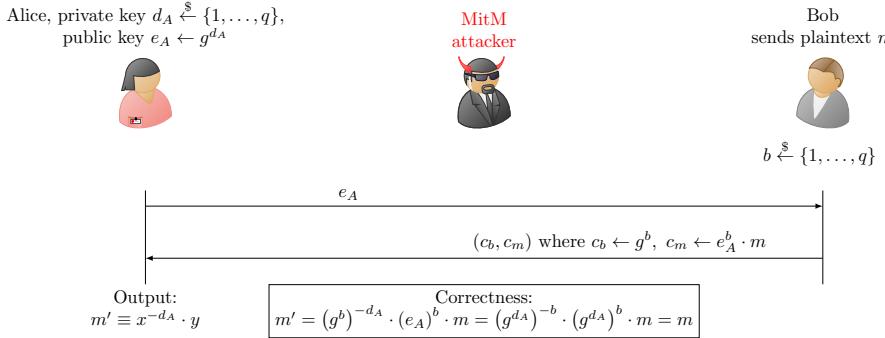


Figure 6.15: The El-Gamal Public-Key Cryptosystem using a DDH cyclic group \mathbb{G} , whose order we denote by q ; exponentiations and multiplications are done using the corresponding operations of \mathbb{G} . The private key of Alice, denoted d_A , and the value b used by Bob, are both selected randomly from the set $\{1, \dots, q\}$. Alice computes her public key as $e_A = g^{d_A}$.

El-Gamal PKC is *multiplicative homomorphic*; we explain this property and some of its important applications in the following section.

Let us describe the DDH El-Gamal PKC in more details; we also illustrate it in Fig. 6.15. We use g to denote a generator of \mathbb{G} , and q to denote the *order* of \mathbb{G} ; i.e., $\mathbb{G} = \{g^1, \dots, g^q\}$. Alice selects her private key d_A to be a random element in the group, i.e., $d_A \xleftarrow{\$} \{g^1, \dots, g^q\}$; and her public key is simply $e_A = g^{d_A}$. Notice that we use the standard notations of multiplication and exponentiation for the corresponding group operations of \mathbb{G} . As shown in Fig. 6.15, the El-Gamal encryption of plaintext $m \in \mathbb{G}$, denoted $E_{e_A}(m)$, is computed as follows:

$$E_{e_A}(m) \leftarrow \begin{cases} b \xleftarrow{\$} [1, q] \\ \text{Return } (g^b, m \cdot e_A^b) \end{cases} \quad (6.36)$$

El-Gamal decryption is defined as:

$$D_{d_A}(c_b, c_m) = c_b^{-d_A} \cdot c_m \quad (6.37)$$

The correctness property holds since for every message $m \in \mathbb{G}$ holds:

$$\begin{aligned} D_{d_A}(g^b, m \cdot e_A^b) &= [(g^b)^{-d_A} \cdot (m \cdot (g^{d_A})^b)] \\ &= [g^{-b \cdot d_A} \cdot m \cdot g^{b \cdot d_A}] \\ &= m \end{aligned} \quad (6.38)$$

Exercise 6.7. In this exercise, use the $\mod p$ modular group (where p is a prime) to compute (the original) El-Gamal encryption. Let $p = 5$.

1. Find a generator for \mathbb{Z}_p^* . (There are only three candidates to try!)

2. Let's select the private key of Alice as $d_A = 2$. Compute Alice's public key, $e_A = g^{d_A} \bmod p$.
3. Compute El-Gamal encryption of 4 and of 3: $c_4 \equiv E_{e_A}(4)$, $c_3 \equiv E_{e_A}(3)$.
Comment: this is a randomized encryption, so another encryption may result in a different output!
4. Compute the decryptions of c_4 and of c_3 .
5. Explain why El-Gamal encryption using $\bmod p$ group - even for large safe prime p - does not satisfy the requirements of secure encryption.

6.4.3 El-Gamal is Multiplicative-Homomorphic Encryption

An encryption scheme (E, D) is *multiplicative homomorphic* if there is an operation, which we denote \times , defined over a pair of ciphertext messages, such that for every public-private key pair (e, d) and every pair of plaintext message m_1, m_2 holds that $E_e(m_1) \times E_e(m_2)$ is an encryption of $m_1 \cdot m_2$, namely:

$$m_1 \cdot m_2 = D_d(E_e(m_1) \times E_e(m_2)) \quad (6.39)$$

Where $m_1 \cdot m_2$ is integer multiplication.

In this section, we show that the (non-hashed) El-Gamal cryptosystem is *multiplicative homomorphic*, and discuss some of the applications of this property. We focus on the use of a DDH group \mathbb{G} , i.e., a cyclic group which is believed to satisfy the DDH assumption (Definition 6.7).

It is convenient to think of \times as a ‘multiplication of ciphertexts’ operation. Following this, the homomorphic property basically means, that the multiplication of two ciphertexts, $E_e(m_1) \times E_e(m_2)$, is equivalent to the encryption of the multiplication of the two messages, $E_e(m_1 \cdot m_2)$. Notice that $m_1 \cdot m_2$ is done using the group operation rather than normal multiplication.

The El-Gamal \times operation is also similar to multiplication. Recall that in the El-Gamal PKC, ciphertexts consist of pairs (c_b, c_m) of elements from the group \mathbb{G} . The \times operator applied to a pair of ciphertexts, (c_b, c_m) and $(c_{b'}, c_{m'})$, is defined as:

$$(c_b, c_m) \times (c_{b'}, c_{m'}) \equiv (c_b \cdot c_{b'}, c_m \cdot c_{m'}) \quad (6.40)$$

The following lemma shows that the \times operator correctly computes the encryption of the multiplication of the two plaintext messages m and m' whose El-Gamal encryptions are (c_b, c_m) and $(c_{b'}, c_{m'})$, respectively.

Lemma 6.3. *Let $d_A \xleftarrow{\$} [1, p - 1]$ and $e_A = g^{d_A}$. Then for any two messages $m, m' \in \mathbb{G}$ holds and any encryption of them $E_{e_A}(m), E_{e_A}(m')$ holds:*

$$m \cdot m' = D_{d_A}(E_{e_A}(m) \times E_{e_A}(m')) \quad (6.41)$$

Proof: Let b be the random exponent used to compute $E_{e_A}(m)$ and b' be the random exponent used to compute $E_{e_A}(m')$. Then:

$$E_{e_A}(m) = (g^b, m \cdot e_A^b) \quad (6.42)$$

$$E_{e_A}(m') = (g^{b'}, m' \cdot e_A^{b'}) \quad (6.43)$$

Hence:

$$E_{e_A}(m) \times E_{e_A}(m') = (g^{b+b'}, m \cdot m' \cdot e_A^{b+b'}) \quad (6.44)$$

And:

$$D_{d_A}(E_{e_A}(m) \times E_{e_A}(m')) = (g^{b+b'})^{-d_A} \cdot m \cdot m' \cdot (g^{d_A})^{b+b'} \quad (6.45)$$

$$= m \cdot m' \quad (6.46)$$

□

Exercise 6.8. Use the values of p, g from Exercise 6.7, and perform all multiplications mod p .

1. Compute $c_M \equiv c_3 \times c_4$.
2. Compute the decryption of c_M . Explain why the result is as expected from the lemma.

Note: it is not secure to use the mod p group for El-Gamal (last item in Exercise 6.7).

6.4.4 Types and Applications of Homomorphic Encryption

In the next section we discuss another multiplicative-homomorphic, the *textbook RSA* PKC. Like we defined multiplicative-homomorphic encryption, we can define other types of homomorphic encryption. One obvious example is *additive-homomorphic encryption*. There are also encryption schemes which are homomorphic with respect to multiple operations - each using a different operator, of course.

In particular, encryption schemes which are homomorphic with respect to *both* multiplication *and* addition are referred to as *Fully Homomorphic Encryption (FHE)*. FHE is a very powerful tool; it allows computation of the encryption of the *arbitrary function* of the plaintext, given only the ciphertexts. For example, we can encrypt the (secret) inputs, send them to an untrusted computation server which will compute the encryption of the function over the inputs, and then decrypt the results - without exposing the inputs to the untrusted server. Namely, FHE allows *arbitrary computations over encrypted data*. Such schemes have different applications, e.g., in *cloud computing*, where an untrusted cloud service is performing some computation on encrypted values.

However, known FHE schemes are complex and have significant overhead, in terms of computation time and/or key/ciphertext length. This is in comparison

to *Partially-Homomorphic Encryption (PHE)* schemes that are homomorphic with respect to only one operation, e.g., multiplication. In some applications, a single operations suffices. For example, multiplicative-homomorphic encryption allows *multiplication of ciphertexts*, as we have just seen; given two ciphertexts, we can compute the encryption of their multiplication - *without knowing the plaintexts or the decryption key*.

In this section, we will give a glimpse of the important applications of homomorphic encryption, limiting ourselves to multiplicative-homomorphic encryption. We focus on the El-Gamal multiplicative homomorphic encryption, and demonstrate how it can be used for applications requiring *anonymity*, and specifically, for *anonymous voting*. This is a tiny taste from the extensive research on the use of cryptography to ensure *privacy, anonymity and secure and private voting*.

Secure and private voting. Secure voting is essential for democracy; and one of the main requirements is, usually, *voting privacy*, i.e., preserving the confidentiality of the vote of each individual, and only exposing the tally of entire populations. This may be achieved by use of physical designs such as a ballot box or trusted voting machines.

There is extensive research on the use of cryptography to ensure *secure electronic voting*. We focus on *voter privacy*, i.e., ensuring the secrecy of the vote of specific individuals.

First, let us consider a trivial design for an e-voting system: voters encrypt their votes with the public key of a trusted server, to which they then send their votes; the server *decrypts* and then *tallies* the votes. This system requires complete trust in the server; in particular, the server can trivially know the vote of each voter.

To ensure voter privacy, we separate the two functions of the server and use *two servers*: a *tally server* and a *decrypt server*. We also switch the order of operations: the encrypted votes are sent first to the tally server, who aggregates all of them into a single (encrypted) value, and then sent to the decrypt server, who decrypts to produce the final outcome.

This privacy-preserving voting process is shown in Fig. 6.16. As shown, each candidate i is assigned a unique small prime number: $p_i > 1$. Each voter, e.g. Alice, selects one candidate, say i_A (with identifier p_{i_A}), and sends to the tally server $E_{e_{DS}}(p_{i_A})$. The tally server combines the encrypted votes by computing $x \equiv E_{e_{DS}}(p_{i_A}) \times E_{e_{DS}}(p_{i_B}) \times E_{e_{DS}}(p_{i_C})$ and sending x to the decrypt server. From Equation 6.39, $x = E_{e_{DS}}(p_{i_A}) \times E_{e_{DS}}(p_{i_B}) \times E_{e_{DS}}(p_{i_C}) = E_{e_{DS}}(p_{i_A} \cdot p_{i_B} \cdot p_{i_C})$, i.e., x is the encryption of $p_{i_A} \cdot p_{i_B} \cdot p_{i_C}$. Hence the decrypt server outputs $p_{i_A} \cdot p_{i_B} \cdot p_{i_C}$, i.e., the combined vote. By factoring the combined vote, we find how many votes were given to each candidate i . Note that this factoring operation is efficient, since we know exactly all possible factors. Also, note that for the factoring to provide correct result, the combined vote must always be less than p .

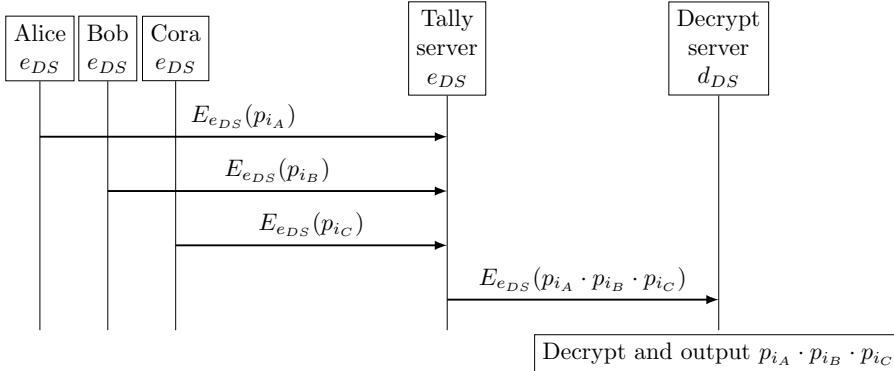


Figure 6.16: Example: Privacy-preserving voting using two servers and multiplicative-homomorphic encryption, e.g., El-Gamal PKC; $p_i > 1$ is a small prime number assigned to candidate i . Voters send encrypted using e_{DS} , the public key of the Decrypt server, the identifier of their candidate, and send to the tally server. The tally server sends $e_{DS}(p_{i_A} \cdot p_{i_B} \cdot p_{i_C})$ to the decrypt server, who outputs the combined vote $p_{i_A} \cdot p_{i_B} \cdot p_{i_C}$. By factoring this, we find how many votes were given to each candidate i .

Obviously, this is not a complete description of a secure voting system. A complete system would not only ensure voter privacy, but also prevent *cheating* by users and by the servers.

Exercise 6.9. We continue with $p = 5$ and same g and public key from Exercise 6.8; now use this as the public key of the Decrypt server, e_{DS} . Let there be two candidates: 2 and 3 (actually, we can't have more with $p = 5$ - why?).

1. Alice and Bob vote 2 and Cora votes 3; compute the encrypted votes.
2. Compute the encrypted combined vote (output of Tally).
3. Decrypt the combined vote.
4. The output may not be correct; explain why.
5. Show at least one way in which a corrupt party (voter, tally server or decrypt server) can manipulate the results.

Another application: Re-encryption. Let us point out another application of multiplicative homomorphic encryption, such as the El-Gamal PKC: *re-encryption*. Namely, consider encryption $(c_m, c_b) = E_{e_A}(m)$ of plaintext m using public key e_A . Let $(c_1, c_{b,1}) \leftarrow E_{e_A}(1)$ be an encryption of 1 using e_A , and let $(c'_m, c'_b) \leftarrow (c_m, c_b) \times (c_1, c_{b,1})$. From the homomorphic property (Equation 6.39), (c'_m, c'_b) is an encryption of $m \cdot 1 = m$, i.e., it is also an encryption of m . A further property of re-encryption is that an adversary

cannot distinguish between the re-encryption of (c_m, c_b) and an encryption of a different message $\hat{m} \neq m$. Re-encryption is used in different protocols, often to preserve anonymity.

Re-encryption preserves the same decryption key. El-Gamal also allows a similar, but different, mechanism, called *proxy re-encryption*, where another entity called *proxy* is given a special key, denoted $e_{A \rightarrow B}$ and computed by Alice, that allows the proxy to transform a ciphertext message encrypted with the key of Alice, $c = E_{e_A}(m)$, into an encryption of the same message but with Bob's key, $c' = E_{e_B}(m)$. Proxy re-encryption has been proposed for different applications, such as monitoring of encrypted traffic. For more details on this mechanism and its applications, see [59].

Re-encryption (and proxy re-encryption) require the use or awareness of the public key e_A with which the message was encrypted. In some applications, it is desirable to allow re-encryption without specifying the public key, e.g., for *recipient anonymity*. In such case, one can use an elegant extension called *universal re-encryption*, which allows re-encryption without knowledge of the encryption key e_A . This is done by appending the encryption $E_{e_A}(1)$ to each ciphertext; see details in [146].

Homomorphic encryption cannot be IND-CCA secure! We see that homomorphic encryption has some nice applications. However, there is a caveat, namely, *homomorphic encryption cannot be IND-CCA secure*. This is especially clear to see, considering the re-encryption application. Namely, an attacker can re-encrypt the encryption $c^* = E_e(m^*)$ of a challenge message m^* , resulting in ciphertext $c' \neq c^*$, which also decrypts to m^* . The CCA attack allows decryption of c' , since $c' \neq c^*$, and this provides the attacker with the challenge message m^* . This argument can be extended to show that homomorphic encryption cannot ensure ‘relaxed CCA security’ (rCCA), where the attacker cannot make a ciphertext query which decrpts to the challenge plaintext (such as c'); see Exercise 6.29.

The homomorphic property is so useful, that it has multiple applications such as these mentioned above, in spite of the fact that it cannot be IND-CCA (or even IND-rCCA) secure. However, this does require extra care and expertise. For example, we briefly mentioned that the *text book RSA* PKC is multiplicative-homomorphic. We will see that this fact was, indeed, abused in an important attack against RSA, which motivates use of RSA with a padding mechanism (which makes it non-homomorphic). With that, let us proceed to discuss RSA.

6.5 The RSA Public-Key Cryptosystem

In 1978, Rivest, Shamir and Adelman presented the first proposal for a public-key cryptosystem - as well as a digital signature scheme [276]. This beautiful scheme is usually referred to by the initials of the inventors, i.e., RSA; it was awarded the Turing award in 2002, and is still widely used. We will cover

here only some basic details of RSA; a more in-depth study is recommended, by taking a course in cryptography and/or reading one of the many books on cryptography covering RSA in depth.

The reader may want to refresh on subsection A.2.2 and Section A.2.3 before learning this section.

6.5.1 RSA key generation.

Key generation in RSA is more complex than for DH and El-Gamal. We first list the steps, and then explain them:

- Select a pair of large prime numbers p, q ; let $n = p \cdot q$ and let $\phi_n = (p - 1) \cdot (q - 1)$. To encrypt messages of up to N bits, pick p, q to be each a random $\lceil \frac{N}{2} \rceil$ bits.
- Select a value e which is co-prime to ϕ_n , i.e., $\gcd(e, \phi_n) = 1$.
- Compute d s.t. $e \cdot d \pmod{\phi_n} = 1$.
- The public key is (e, n) and the private key is (d, n) .

Selecting e to be co-prime to ϕ_n is necessary - and sufficient - to ensure that e has a multiplicative inverse d in the group $\pmod{\phi_n}$. To find the inverse d , we can use the extended Euclidean algorithm. This algorithm efficiently finds numbers d, x s.t. $e \cdot d + \phi_n \cdot x = \gcd(e, \phi_n) = 1$; namely, $e \cdot d = 1 \pmod{\phi_n}$. See subsection A.2.2.

The public key of RSA is (e, n) and the private key is (d, n) , since the modulus n is required for both encryption and decryption. However, we - and others - often abuse notation and refer to the keys simply as e and d .

6.5.2 Textbook RSA: encryption, decryption, and signing.

The RSA cryptosystem is based on the *RSA-encrypt* function $E_e^{RSA}(m)$, applied to plaintext m using public key e , and the RSA decryption function $D_d^{RSA}(c)$, applied to ciphertext c and using the private key d . These functions are computed as follows:

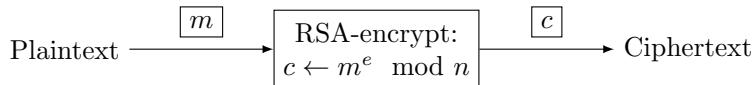
$$E_e^{RSA}(m) = m^e \pmod{n} \quad (6.47)$$

$$D_d^{RSA}(c) = c^d \pmod{n} \quad (6.48)$$

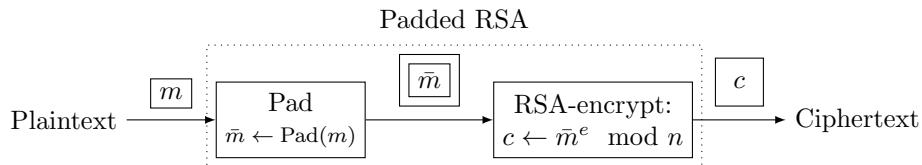
Here, the message m is encoded as a positive integer, and limited to $m < n$, ensuring that $m = m \pmod{n}$. In subsection 6.5.4 we show that this ensures *correctness*, i.e., correct decryption, namely:

$$\text{For every message } m \text{ and } c \leftarrow E_e^{RSA}(m) \text{ holds: } m = D_d^{RSA}(c) \quad (6.49)$$

We use the term *textbook-RSA* encryption, for RSA encryption performed by directly applying the RSA-encrypt function to the plaintext, without any



(a) Textbook-RSA, presented in subsection 6.5.2 and shown to be vulnerable in subsection 6.5.5.



(b) Padded RSA, usually following the PKCS#1 standard; see subsection 6.5.6. Version 1.5 of PKCS#1 is vulnerable; version 2 of PKCS#1, also referred to as OAEP, is considered secure. We use \bar{m} for the padded plaintext ($\bar{m} = \text{Pad}(m)$).

Figure 6.17: RSA encryption: (a) textbook RSA (vulnerable) vs. (b) padded RSA.

padding or other preprocessing, i.e., using E^{RSA} and D^{RSA} as defined above. As we explain in subsection 6.5.5, textbook RSA encryption has significant vulnerabilities. Therefore, in practice, the input to RSA is always processed; this preprocessing of the input is referred to as *padding*.

Padding is defined by a pair of functions $(\text{pad}, \text{unpad})$. The input to pad , and the output of unpad , are plaintext messages; and the two functions should ensure that unpadding of a padded message, recovers the message as it was before padding, namely:

$$\text{For every message } m \text{ holds: } m = \text{unpad}(\text{pad}(m)) \quad (6.50)$$

From the correctness of RSA (Equation 6.49) follows the *correctness of padded RSA*:

$$\text{For every message } m \text{ and } c \leftarrow E_e^{RSA}(\text{pad}(m)) \text{ holds: } m = \text{unpad}(D_d^{RSA}(c)) \quad (6.51)$$

We discuss standard RSA padding functions and (standard) padded RSA encryption in subsection 6.5.6. We illustrate textbook-RSA vs. padded-RSA in Figure 6.17.

Textbook RSA signatures: encrypting with private key? RSA is also the basis for a signature scheme, which we discuss in subsection 6.6.1. Signing uses the same key-generation process as explained above, except that we usually denote the public verification key by v (instead of public encryption key e) and the private signing key by s (instead of private decryption key d).

Similarly to encryption, we can also define *RSA textbook signatures*, by computing the RSA function over the message using the private signing key s ,

as in:

$$\text{Sign}_s^{\text{RSA}}(m) = m^s \pmod{n} \quad (6.52)$$

$$\text{Verify}_v^{\text{RSA}}(m, \sigma) = \begin{cases} \text{TRUE if } m = \sigma^v \pmod{n}, \\ \text{FALSE otherwise} \end{cases} \quad (6.53)$$

In practice, textbook RSA signatures are *never* used. One reason is that textbook RSA signatures can only be applied if the message m is less than n , which is rarely, if ever, the case. Therefore, in practice, RSA signatures always involve some additional processing, typically using the *Hash-then-Sign* (subsection 3.2.6), i.e., computing $\text{Sign}_s^{\text{RSA}}(h(m))$. Using textbook RSA signatures also introduces vulnerabilities, which are similar to those outlined for textbook RSA encryption in subsection 6.5.5.

Notice that the textbook RSA signing equation Equation 6.52 is exactly the same as the textbook RSA encryption equation Equation 6.47, except for the use of the private signing key s instead of the public encryption key e . Therefore, you may find reference to RSA signing a ‘encryption with the private key’. We recommend to *avoid this expression*, since it only applied to textbook RSA signatures and textbook RSA encryption, which are both vulnerable and not used in practice, and not to the secure RSA signatures and encryption (using padding and/or hashing). See further discussion of RSA signatures in subsection 6.6.1.

6.5.3 Efficiency of RSA

The RSA algorithms are conceptually simple; however, their computation requires considerable resources. The basic reason for that is that RSA security completely breaks down if an attacker is able to factor n , and find its factors (p and q); this allows computation of $\phi(n) = (p - 1) \cdot (q - 1)$, and using $\phi(n)$, the computation of the private key d from the public key e , since $d = e^{-1} \pmod{n}$ (and using $\phi(n)$ we can compute the multiplicative inverses).

Factoring is a well-studied problem, and while no *polynomial* factoring algorithm is known, there are algorithms that improve efficiency considerably. As a result, RSA keys should be quite long, as shown in Table 6.1; notice that the key-length should be chosen based on the maximal time at which the encryption should remain secret, and not based on the current time. For example, for information whose confidentiality should be preserved until 2040, the modulus n should be about 3000 bits, i.e., p and q should be random prime with about 1500 bits.

Computations, especially exponentiation, with such extremely long numbers, is computationally intensive. The computations are modulo n , which keeps the results from becoming even longer, but this requires computation of the modulus of the result (and optionally of intermediate values), and the computation of the modulus is also a computationally-intensive problem.

Therefore, efficiency is a major consideration for implementation of RSA. In this subsection, we discuss only one of the most basic optimizations: *choosing e to improve efficiency*.

The public exponent e is not secret, and, so far, we only required it to be co-prime to $\phi(n)$. This motivates choosing e that will improve efficiency - usually, to make encryption faster.

In particular, choosing $e = 3$ implies that encryption - i.e., computing $m^e \bmod n$ - requires only two multiplications, i.e., is very efficient (compared to exponentiation by larger number). Note, however, that there are several concerns with such extremely-small e ; in particular, if m is also small, in particular, if $c = m^e < n$ (without reducing $\bmod n$), then we can efficiently decrypt by taking the e -th root: $c^{1/e} = (m^e)^{1/e} = m$. This particular concern is relatively easily addressed by *padding*, as discussed below; however, there are several additional attacks on RSA with very-low exponent e , e.g., [87]. Some of these attacks are for the case where a party may encrypt and send the same message, or ‘highly related’ messages, to multiple recipients. These attacks motivate (1) the use of padding to break any possible relationships between messages , as well as (2) the choice of slightly larger e , such as 17 or $2^{16} + 1 = 65537$. The reason to choose these specific primes is that exponentiation requires only 5 or 17 multiplications, respectively; see next exercise.

Exercise 6.10. *Given integer m , show how to compute m^{17} in only five multiplications, and $m^{2^{16}+1}$ in only 17 multiplications.*

Hint: use the following idea: compute m^8 with three multiplications by $m^8 = ((m^2)^2)^2$. \square

Handling long plaintext: hybrid encryption. To complete our discussion of RSA efficiency, let us comment that RSA encryption, like other public-key cryptosystems, used *hybrid encryption* to encrypt long messages. This is necessary, since the input m to the RSA-encrypt function must be less than the modulus n , or decryption will output $m \bmod n$ which will differ from m .

Theoretically, we could have selected n to be longer than the longest message, but this would have resulted in excessive overhead. Therefore, we select n based on the security requirements (Table 6.1), which is much shorter than (normal) plaintext. To encrypt ‘normal’ messages, which are typically much longer, we apply hybrid encryption. In hybrid encryption, we use the public key encryption to encrypt a shared key k , and then use the shared key to efficiently encrypt the long message m . See subsection 6.1.6 and Figure 6.3.

6.5.4 Correctness of RSA

Does RSA decryption really work? Obviously, yes, it does; we now will explain why it does. Before we ‘really’ explain this, an exercise may give some intuition. To solve the exercise (and understand the following discussion), the reader may want to refresh on multiplicative inverses subsection A.2.2 and Euler’s function and theorem Section A.2.3.

Exercise 6.11 (Textbook RSA ensures correctness, i.e., decryption recovers message). Let $p = 7$.

1. Recall: \mathbb{Z}_p^* , for a prime p , is the group containing the numbers from 1 to $p - 1$, with the modular multiplication operation. A generator for \mathbb{Z}_p^* is a number $g \in \{1, \dots, p-1\}$ such that by multiplying g by itself enough times, each time modulus p , we get all the numbers in \mathbb{Z}_p^* . Find a generator g for \mathbb{Z}_p^* ; show that g is a generator and how you found it.
2. What is $\phi(p)$?
3. Let $q = 11$, and let $n = q \cdot p$. Compute $\phi(q)$, $\phi(p)$ and $\phi(pq)$; for each of them, compute directly from the definition, and using the relevant facts/lemmas that we learned or that appear in the textbook.
4. Let $e = 11$ be an RSA encryption key for the modulus n ; compute the corresponding private key d . Note: this is a correction; previously we had $e = 3$ - do you see why that value wasn't good?
5. Encode your initials by mapping the letters (A to Z) to the corresponding numeric values (from 1 to 26), resulting in $f, l \in \{1, 2, \dots, 26\}$. Compute $m = f + l + 7$.
6. Compute $c = E_{e,n}^{RSA}(m) = m^e \pmod{n}$.
7. Compute $m' = D_{d,n}^{RSA}(c) = c^d \pmod{n}$.
8. Explain why this encryption is insecure - yet why the use of this value $e = 11$ may be secure in other applications of RSA.

Let us now ‘really’ explain why textbook RSA decryption recovers the plaintext, i.e., the correctness of textbook RSA (Equation 6.49), namely, $(\forall m)D_d^{RSA}(E_e^{RSA}(m)) = m$.

RSA’s correctness is based on *Euler’s Theorem* (Theorem A.3), which says that for any co-prime integers m, n , holds $m^{\phi(n)} = 1 \pmod{n}$, where $\phi(n)$ is the *Euler function*, defined as the number of positive integers which are less than n and co-prime to n . See Section A.2.3. We use the theorem, to explain RSA’s correctness, i.e., why $D_d^{RSA}(E_e^{RSA}(m)) = m$.

Note that for any primes p, q holds $\phi(p) = p - 1$, $\phi(q) = q - 1$, and $\phi(p \cdot q) = (p - 1)(q - 1)$ (Lemma A.1). This is the reason for us using $\phi_n = \phi(n) = \phi(p \cdot q) = (p - 1)(q - 1)$ in the RSA key generation process.

Recall that $e \cdot d = 1 \pmod{\phi(n)}$, i.e., for some integer i it holds that $e \cdot d = 1 + i \cdot \phi(n)$. Hence:

$$\begin{aligned} m^{e \cdot d} \pmod{n} &= m^{1+i \cdot \phi(n)} \pmod{n} \\ &= m \cdot (m^{\phi(n)})^i \pmod{n} \end{aligned}$$

Recall Eq. (A.5) : $a^b \bmod n = (a \bmod n)^b \bmod n$. Assuming that m and n are co-prime ($\gcd(m, n) = 1$), we can apply Euler's theorem and can substitute $m^{\phi(n)} = 1 \bmod n$ and receive:

$$m^{e \cdot d} \bmod n = m \cdot (m^{\phi(n)} \bmod n)^i \bmod n \quad (6.54)$$

$$= m \cdot 1^i \bmod n = m \bmod n \quad (6.55)$$

It follows that:

$$\begin{aligned} D_d^{RSA}(E_e^{RSA}(m)) &= D_d^{RSA}(m^e \bmod n) \\ &= (m^e \bmod n)^d \bmod n \\ &= m^{e \cdot d} \bmod n \\ &= m \cdot (m^{\phi(n)} \bmod n)^i \bmod n \quad (6.56) \\ &= m \cdot 1^i \bmod n \\ &= m \bmod n = m \end{aligned}$$

Notice that $m \bmod n = m$ since we restricted plaintext messages m so that $m < n$. Namely, under the assumption (above) that m and n are co-primes, we have shown the correctness of textbook RSA.

What about the assumption that m and n are co-primes? Unfortunately, it does not always hold. Recall that the message m may be any positive integer s.t. $1 < m < n$. Most, but not all, of these possible messages - i.e., integers smaller than n - are co-prime to n . In fact, the number of integers smaller than n and co-prime to n is exactly the definition of $\phi(n)$, which we know to be: $\phi(n) = \phi(p \cdot q) = (p - 1) \cdot (q - 1) = n - q - p + 1$.

So *most* possible messages m are indeed co-prime to n . Still, $p + q - 2$ messages are *not* co-prime to n ; this number is much smaller than n but is still polynomial in n (roughly $n^{\frac{1}{2}}$), and our explanation does not hold for these values. We assure the reader, however, that correctness holds also for these values; it 'just' requires a slightly more elaborate argument. Such arguments, usually using the Chinese Remainder Theorem, can be found in many textbooks on cryptography and number theory, e.g., [171].

6.5.5 The RSA assumption and the vulnerability of textbook RSA

Now that we have seen that the textbook RSA PKC ensures correctness, it is time to discuss its security. We will first discuss the underlying security assumption, and then discuss several vulnerabilities of textbook RSA, which are the reason that in practice, we always use *padded RSA*, which we discuss in subsection 6.5.6.

The security of RSA encryption is based on the *RSA assumption*. Intuitively, the RSA assumption is that there is only negligible probability that an efficient adversary \mathcal{A} correctly recovers the plaintext m , given the ciphertext $m^e \bmod n$

and the public key (e, n) . Let us restate the RSA assumption a bit more formally.

Definition 6.8 (RSA assumption). *Choose n, e as explained above, i.e., $n = pq$ for p, q chosen as random l -bit prime numbers, and e is co-prime to ϕ_n . The RSA assumption is that for any efficient (PPT) algorithm \mathcal{A} and constant c , and for sufficiently large l , holds:*

$$\Pr[\mathcal{A}((e, n), m^e \bmod n) = m] \in \text{NEGL}(l) \quad (6.57)$$

Where m is chosen randomly $m \xleftarrow{\$} [2, n - 2]$.

The RSA assumption is also referred to sometimes as the *RSA trapdoor one-way permutation assumption*. The ‘trapdoor’ refers to the fact that d is a ‘trapdoor’ that allows inversion of RSA; the ‘one-way’ refers to the fact that computing RSA (given public key (e, n)) is easy, but inverting is ‘hard’; and the ‘permutation’ is due to RSA being a permutation (and in particular, invertible). See also the related concept of *one-way functions* in §3.4.

One obvious question is the relation between the RSA assumption and the assumption that factoring is hard. Assume that some adversary A_F can efficiently factor large numbers, specifically, the modulus n (which is part of the RSA public key). Hence, A_F can factor n , find q and p , compute ϕ_n and proceed to compute the decryption key d , given the public key $\langle e, n \rangle$, just like done in RSA key generation. We therefore conclude that if factoring is easy, i.e., there exists such adversary A , then the RSA assumption cannot hold (and RSA is insecure).

Textbook RSA is multiplicative-homomorphic - and vulnerable. Assume now that we are willing to accept the RSA assumption. What, then, about the security of RSA, when used as a public key cryptosystem (PKC)? In this subsection, we discuss textbook RSA, and argue that it is vulnerable; this motivates the PKCS#1 specifications for ‘padded RSA encryption’, which we discuss in the next subsection.

Before we discuss the vulnerabilities of textbook RSA, let us point out an important property of it: *textbook RSA is multiplicative-homomorphic*. Indeed, this follows quite simply:

$$\begin{aligned} E_e^{RSA}(m_1 \cdot m_2) &= (m_1 \cdot m_2)^e \bmod n \\ &= m_1^e \cdot m_2^e \bmod n \\ &= E_e^{RSA}(m_1) \cdot E_e^{RSA}(m_2) \bmod n \end{aligned}$$

As we already discussed in subsection 6.4.3, a homomorphic encryption scheme cannot be IND-CCA secure. This is a drawback, requiring extra care in design of secure applications using a homomorphic encryption scheme. However, in the case of textbook RSA, there are additional vulnerabilities, which makes its use clearly inadvisable:

1. Unlike El-Gamal PKC, the textbook RSA PKC is deterministic; hence, encryption of the same plaintext m , will always result in the same ciphertext $c = m^e \bmod n$. Suppose that the attacker guesses (or knows) a set of likely (or possible) plaintexts, say m_1 , m_2 and m_3 . The attacker can easily compute, say, $c_1 = m_1^e \bmod n$; if the plaintext message m was the same as m_1 , then $c = c_1$. Textbook RSA encryption resembles, in this sense, the insecure ECB mode of operation (Section 2.8), which has the same vulnerability. Secure encryption⁴ - shared key or public key - must be randomized and/or stateful!
2. Textbook-RSA is vulnerable to *low-exponent attacks*, especially when sending low-value messages (small m); we mentioned the trivial attack when $m^e < n$. See more elaborate low-exponent attacks in [87]); these exploit scenarios where we send identical or related messages to multiple recipients.
3. The RSA assumption does not rule out a potential *exposure of partial information* about the plaintext, e.g., a particular bit. The $\log(n)$ least-significant bits were shown to be as secure as the entire preimage [8]; however, it may be possible to expose other bits, like the one bit of the discrete log (see subsection 6.1.8).
4. Finally, while every homomorphic encryption scheme is vulnerable to CCA attacks, textbook RSA is also vulnerable to *much weakened* version of CCA attacks, where the attacker only needs to receive very limited information about the ciphertext. In subsection 6.5.7 we present Bleichenbacher's attack, which can effectively 'break' RSA given only a one-bit error indicator, specifically, indication if the result of textbook RSA decryption is a 'properly padded plaintext'. Not only this breaks 'textbook RSA', but it also breaks *padded RSA*, when using the (widely-used and very simple) RSA PKCS#1 version 1.5 padding. We discuss padded RSA in the following subsection.

6.5.6 Padded RSA encryption: PKCS#1, with v1.5 and OAEP encodings

In the previous subsection, we saw that textbook RSA has significant vulnerabilities, making it inadvisable to use it. Therefore, to improve security, practical deployments always use *padded RSA*, i.e., apply a *pad* function to the message before applying the encryption operation, and a corresponding *unpad* function to recover the plaintext after decryption. The unpad function should recover the message before padding, which ensures the *correctness of padded RSA*, i.e., for every message m we always have $m = \text{unpad}(D_d^{\text{RSA}}(E_e^{\text{RSA}}(m)))$ (Equation 6.51).

⁴However, some designs of cloud databases employ deterministic encryption, specifically to facilitate identification of encryption of the same element. Of course, this must be done with great care to avoid unintended exposure.

Practical deployments of RSA encryption, usually follow one of the versions of the PKCS #1 specifications⁵ [318].

Before version 2.0, PKCS#1 defined only one padding, which we refer to as the *v1.5 padding*; version 2.0 added *OAEPE padding*, based on the design from [36]. We briefly discuss both of these widely used padding schemes.

PKCS#1 version 1.5 padding. The v1.5 padding was designed mainly to address two of the vulnerabilities of textbook RSA: vulnerability 1 (due to deterministic output) and vulnerability 2 (low-exponent attacks). Namely:

- To prevent an attacker from identifying multiple encryptions of the same plaintext, possibly by the attacker encrypting some guesses of possible plaintexts, the padding would include a sufficient number of random bits.
- To prevent the low-exponent attacks (vulnerability 2), prepend the message bytes with the random bits. To further ensure a sufficiently large value for the padded-plaintext, prepend this with a non-zero byte.

Specifically, the *PKCS#1* version 1.5 padding algorithm, $\text{Pad}_{v1.5}(\cdot)$, is defined as follows. Given an input (pre-padding) plaintext message m , and a random string r of at least eight non-zero random bytes, we compute the padded message $M = \text{Pad}_{v1.5}(m)$ as:

$$M = \text{Pad}_{v1.5}(m) = 0x00 \parallel 0x02 \parallel r \parallel 0x00 \parallel m \quad (6.58)$$

To ensure that the binary value of M is less than n , as required for correct decryption, the (pre-padding) message m must contain less than $l - 11$ bytes, where l is the length of the modulus n (in bytes). The value of the second byte ($0x02$) prevents low-exponent attacks (by being non-zero), but further, is used to identify that the input is for the RSA encryption, and not, say, to RSA signing or RSA signature-verification.

In the decryption process, we first obtain the padded message M , from which we can easily extract and return only the plaintext m . The message should be returned only if the padding is correct, i.e., begins with $0x0002$, followed by at least eight non-zero bytes, and finally followed by a zero byte. If M deviates from this in any way, an error indicator is returned.

As the readers hopefully agree, the *PKCS#1* version 1.5 padding, defined in Equation 6.58, is simple to understand and easy to implement; indeed, it is widely deployed. However, this padding turns out to be vulnerable to some attacks. The first of these was Bleichenbacher's Padding Side Channel Attack, which we discuss in the next subsection (and see Figure 6.21).

⁵PKCS stands for *Public Key Cryptography Standards* for standard padding. The PKCS specifications were published by RSA Security LLC; versions 1.5, 2.0, 2.1 and 2.2 were define by the IETF, in RFCs [182, 185, 186] and [318], respectively.

Optimal Asymmetric Encryption Padding (OAEP). A more secure padding called *Optimal Asymmetric Encryption Padding (OAEP)* was proposed by Bellare and Rogaway [36]. From version 2.0 of the PKCS#1 standard, until the version 2.2 [318], the current version, the standard includes both OAEP as well as the PKCS#1v1.5 padding. However, OAEP should be used whenever possible: it is more secure, without noticeable extra overhead.

Intuitively, OAEP further improves the security of RSA, by addressing the two vulnerabilities not addressed by the v1.5 padding:

- To deal with the concern that RSA may expose partial information (vulnerability 3), OAEP makes it necessary to expose many bits of the input to the RSA encrypt function, in order to expose any information about the plaintext.
- To deal with the vulnerability of RSA to CCA attacks, OAEP adds *redundancy* to the plaintext. This should make it infeasible for the attacker to learn sensitive information by sending manipulated ciphertext messages, since, almost always, their decryption would not have the correct redundancy (and be rejected). Bellare and Rogaway coined the term *plaintext-aware* encryption for the property that valid ciphertext can only be sent by encrypting ‘known’ plaintext. Of course, the attacker may encrypt plaintext to obtain valid ciphertext, but in this case, the attacker knew the outcome of the decryption process in advance!

Following [36], we present first a ‘simplified OAEP padding’. The simplified padding focuses on *prevention of exposure of partial information*. Later, we describe the (non-simplified) OAEP padding, which extends the simplified padding, and also aims to *defend against CCA attacks*.

Simplified OAEP padding. The ‘simplified OAEP padding’, illustrated in Figure 6.18, is already quite clever, and therefore, let us develop it in three stages: Pad_0 , Pad_1 and then Pad_2 . First, for Pad_0 , consider an attacker which can only *expose a single bit* in the preimage of the RSA encryption; of course we don’t know *which bit*. To prevent such attacker from exposing any bit of the plaintext, we let Pad_0 first select a random string r of the same length as the plaintext m , and then output $m \oplus r \# r$. Namely, padding Pad_0 selects a random one-time pad r to m ; the padded message M consists of the ‘encrypted’ plaintext $m \oplus r$, concatenated to the ‘pad’ r .

Obviously, the Pad_0 padding is inefficient. First, it only protects against exposure of a single bit; what if we can expose two bits of the preimage, e.g., the first bit of $m \oplus r$ and the (corresponding) first bit of r ? Second, do we really need to send such a long pad (as long as the plaintext m)?

We address both issues in Pad_1 , by using a *shorter random string* r , and then applying a *cryptographic hash function* g , whose output is as long as the plaintext m ; so now we use $g(r)$ as the one-time pad. It to a longer string $g(r)$. Clearly, we reduced the overhead since r is shorter. Furthermore, if we view g as a ‘random oracle’, then security also improved. Even if the attacker

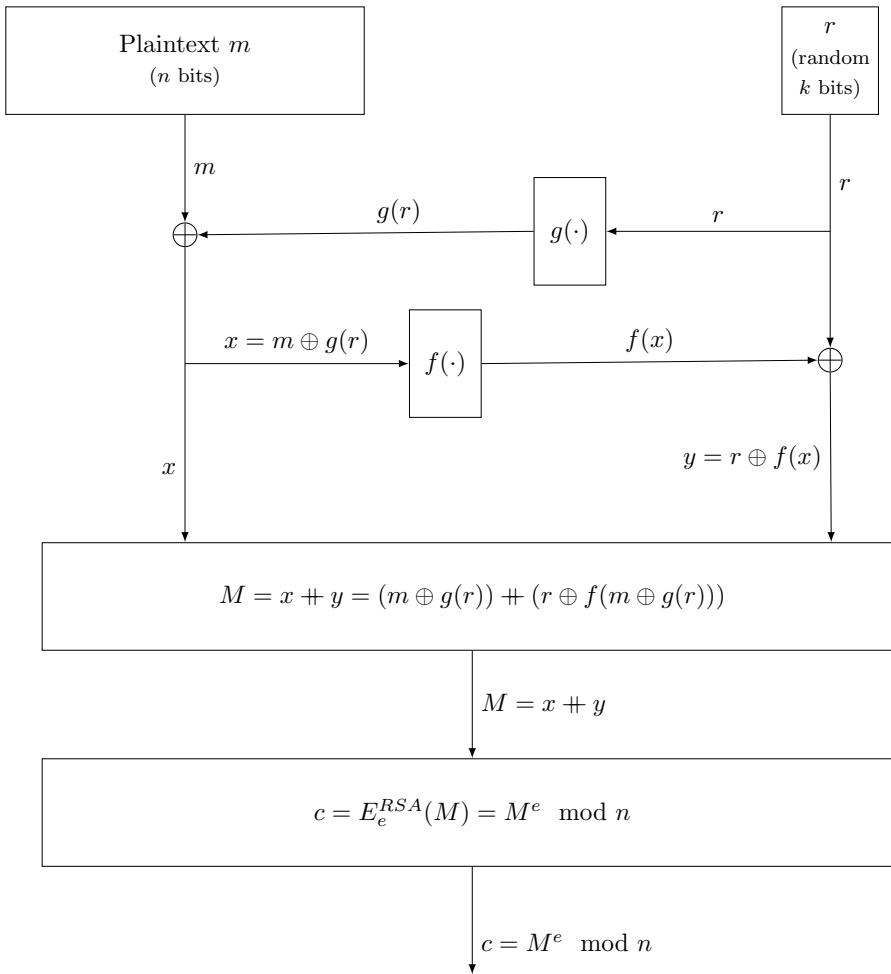


Figure 6.18: Simplified OAEP padding. This padding efficiently defends against partial exposure of RSA preimage, but, in this simplified version, does not validate that the ciphertext is the result of applying encryption, i.e., does not ensure plaintext-awareness.

knows several - but not all - of the random string r , then its output bits are still random.

Suppose, however, that the attacker can expose $|r|$ bits of the preimage of RSA - or, specifically, all the bits of the (short) random string r . Of course, r cannot be too short if we rely on it to randomize $g(r)$, so this would imply that the RSA encryption function is much weaker than we expect; but still, it would be nice to protect also against such a case - if we can do it easily - and we can.

In fact, in this third (and final) simplified padding, Pad_2 , we simply *apply again the ‘hash then one-time pad’ method of Pad_1* - this time, to ‘protect

r' . More specifically, given plaintext m , the Pad_2 algorithm first computes $x = m \oplus g(r)$, for a random string r , and then outputs the padding $x \# h(x) \oplus r$, where h is yet-another cryptographic hash function. To further clarify, let us present the corresponding UnPad_2 function, which receives the results of the RSA decryption function:

$$\text{UnPad}_2(x \# y) = x \oplus g(h(x)) \quad (6.59)$$

The reader should confirm that this ensures correctness, i.e., $m = \text{UnPad}_2(\text{Pad}_2(m))$.

The security of OAEP is analyzed using the Random Oracle Methodology (ROM), see §3.6.

OAEP padding. Finally, we describe the ‘complete’ OAEP padding, as presented in [36], and later standardized by the IETF (most recently in [318]). This padding adds to the ‘simplified padding’ (Pad_2 above) a simple *redundancy* mechanism, which suffices to ensure plaintext-awareness, and thereby, provides effective defense against CCA attacks. Our description is of the padding and its security is a bit simplified, but we believe it suffices for our (educational) purposes; there are also (minor) differences between the details of the design between the one in [36] and the one standardized by the IETF. Reader interested in details should refer to [318] and to the security analysis in [36] and in follow-up works.

OAEP adds redundancy to the plaintext, simply by appending a sufficiently-long string with known value to the plaintext message m . In fact, OAEP simply appends the string of all-zero bits. We denote it 0^k , where k should be sufficiently large for security; e.g., 128 bits. This design is illustrated in Figure 6.19.

This ‘complete’ design of OAEP, inherits the security properties of the simplified design - with one addition, i.e., *plaintext-awareness*, which provides defense against CCA attacks. This defense is due to the redundancy. Suppose the attacker sends some ciphertext c , which is not the result of legitimate encryption - by the attacker or by another party (and eavesdropped by the attacker). Namely, the attacker does not know the resulting plaintext; and even if we assume that the attacker can find some bits of the preimage, it would definitely not know all of r , and therefore, cannot ensure that the decryption will have the correct the 0^k redundancy string. Intuitively, the attacker does not gain information from CCA attacks.

6.5.7 Bleichenbacher’s Padding Side Channel Attack on PKCS#1 v1.5

In 1998, Daniel Bleichenbacher presented possibly the most important attack against public-key cryptosystems based on RSA [62]. Bleichenbacher’s attack is using the *chosen-ciphertext side-channel attack (CCSCA) model*. We begin this subsection by briefly discussing this attack model, and the important area of *side-channel* attacks in general, and then focus on Bleichenbacher’s attack.

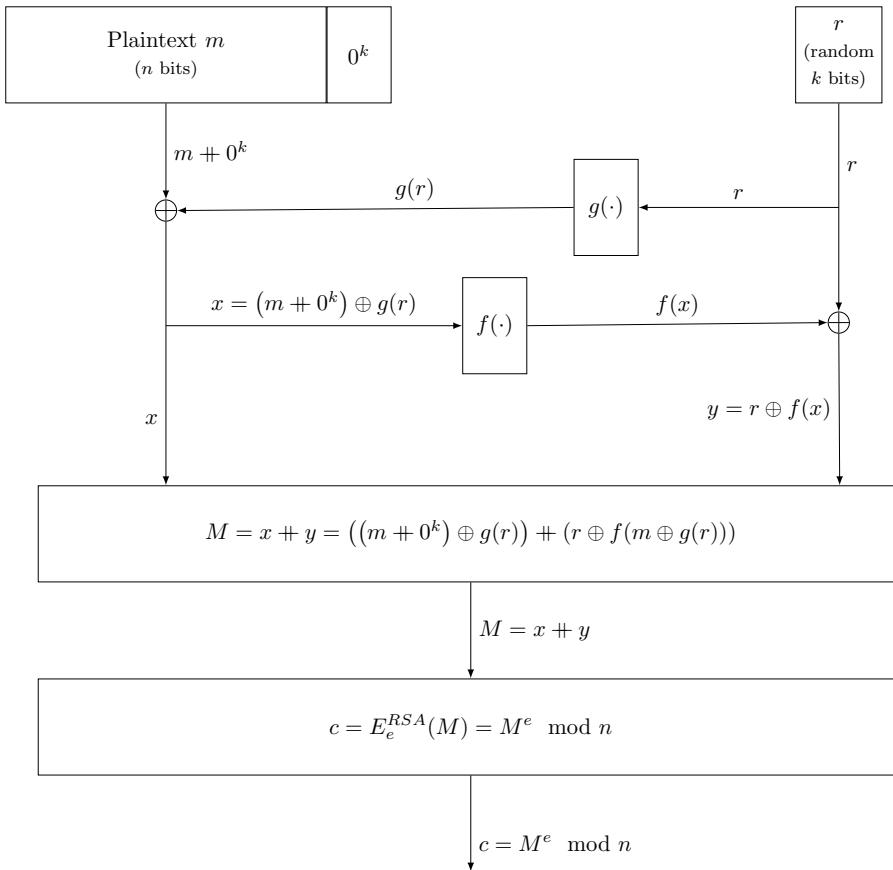


Figure 6.19: OAEP padding [36]

Side-channel attacks and the chosen-ciphertext side-channel attack (CCSCA) model. In chosen-ciphertext side-channel attacks, as in other attack models for encryption schemes, the attacker receives a *challenge ciphertext* c^* , which is the encryption of a challenge plaintext m^* . At the end of the attack, the attacker outputs a *guess* m , and we say that the attacker *wins* if its guess is correct, i.e., if $m = m^*$. What is unique about the CCSCA model are the attacker capabilities. Specifically, like in a ‘regular’ CCA attack, the attacker can give ciphertexts c_1, c_2, \dots to be decrypted; however, in a chosen-ciphertext side-channel attack (CCSCA), the attacker does *not* receive the results of the decryption. Instead, the attacker receives only some side-channel *information* regarding the decryption process and the decrypted message. See Figure 6.20.

A side-channel is *transmission of information using a non-standard channel*, which was *not* intended for communication by the system designers, and, possibly, not considered when evaluating the security of the system. Such a channel is due to some ‘side-effect’ of the operation of the system. For example, in the attack

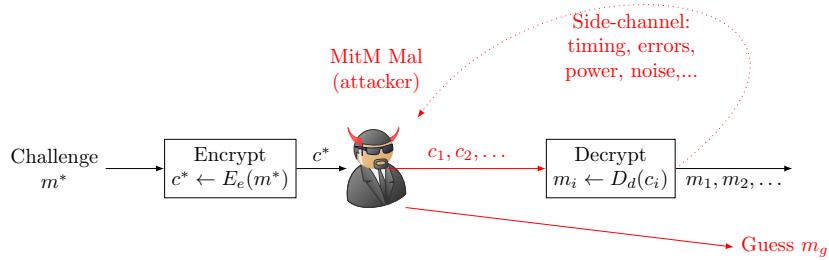


Figure 6.20: The chosen-ciphertext side-channel attack (CCSCA) model. Attacker receives ‘side-channel feedback’ from the processing of adversary-selected ciphertexts. Attacker ‘wins’ when its ‘guess’, m_g is identical to the ‘challenge’ m^* , i.e., when $m_g = m^*$.

models we discussed in Chapter 2, e.g., CPA (see Figure 2.8), the model defines clearly which information is available to the attacker; any other information is therefore excluded. But a side-channel may provide some additional information, ‘outside the model’.

There are different types of side-channels, including timing of events, power consumption, electromagnetic radiation, audio signals, error indicators and more. There are also different applications and goals for side-channels, including non-cryptographic side-channels; for example, one common use of side-channels is to infiltrate information across a firewall or other device which inspects information sent outside of an organization, to detect leakage. Even focusing on cryptographic side-channels, there are different types and goals. In particular, sometimes side-channels are viewed as *increasing* the power of the attacker, for example, a side-channel that leaks information about the computation, which may allow leakage of information regarding the secret/private key. In other side-channel attacks, the side-channel is viewed as a *weaker* assumption regarding the attacker capabilities; specifically, in Bleichenbacher’s attack, the attacker receives ‘only’ a very limited indication about the decrypted message, instead of receiving the exact plaintext (as in a CCA attack).

The information leaked in each side-channel ‘signal’ is, typically, extremely limited, e.g., only a single bit. Indeed, in many side-channel attacks, each ‘signal’ does not provide even one bit of information, since the side-channel signal is obscured by random noise. In spite of that, there have been many successful side-channel attacks on cryptographic systems and other security systems. In this introductory textbook, we only cover Bleichenbacher’s attack on RSA encryption implementations that use the PKCS#1 padding version 1.5.

Bleichenbacher’s side-channel attack. Bleichenbacher’s side-channel attack is one of the most important and well-known side-channel attacks. The attack is against RSA encryption using the PKCS#1 versions 1.5 padding;

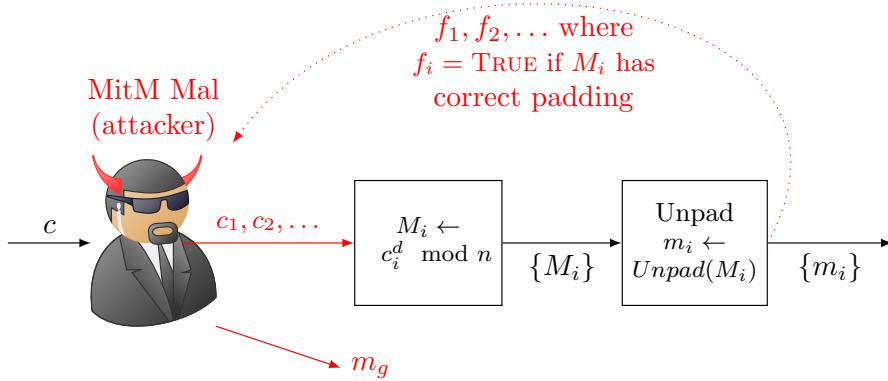


Figure 6.21: Bleichenbacher’s attack on RSA PKCS#1 version 1.5. To compute $c^d \bmod n$, for some given c , the attacker sends ciphertexts $\{c_i = c \cdot s_i \bmod n$, where s_i are different integers. The attacker receives the (side-channel) indications $\{f_i\}$, where f_i is true if M_i , the i^{th} output of the textbook RSA decryption function, has correct padding. We use m_i to denote outcome of the PKCS#1 decryption of c_i , i.e., the result of unpading M_i .

this padding is very popular, and used by numerous systems and standards, including several variants of the important SSL/TLS protocol (Chapter 7. Bleichenbacher’s attack, and variants of it, apply to many systems and standards. In particular, Manger [223] showed a variant that achieves much better performance, against version 2.0 of PKCS#1, which is based on OAEP. As shown in [283], this (more efficient) attack can also be used against many implementations of PKCS#1 v1.5. We note that the attack exploits a seemingly-minor detail where PKCS#1 version 2.0 differs from the ‘plain’ OAEP design; this detail was fixed in version 2.2 of PKCS#1 [240].

The setup of Bleichenbacher’s attack is illustrated in Figure 6.21. The attacker’s goal is to compute a string M which is the same size as the modulus n , which we denote l bytes, and such that $M = c^d \bmod n$, for a given value c .

Bleichenbacher’s attack works for arbitrary c , i.e., c is not necessarily the result of PKCS#1 v1.5 encryption. This generality is actually quite useful, and exploited in some attacks based on Bleichenbacher’s attack, as we discuss in Chapter 7.

However, for now, we focus on the simpler and common case where c is the result of RSA PKCS#1 v1.5 encryption of some plaintext m^* . Often, the attacker obtains c by eavesdropping to a transmission from a benign sender, which obtained c by encrypting some challenge message m^* using RSA with PKCS#1 v1.5 padding. In this case, if the attacker succeeds to output $M = c^d \bmod n$, then this easily provides the challenge message m^* , since $m^* =$

Unpad(M).

Since c is the result of RSA encryption using PKCS#1 v1.5, then we have $M = \text{Pad}_{v1.5}(m)$. From the definition of the pad in Equation 6.58, we know that a properly-padded message such as M must begin with 0x0002. Hence:

$$2B \leq M < 3B, \text{ where } B = 2^{8(l-2)} \quad (6.60)$$

Where l is the length, in bytes, of n (and hence of M). Note that Equation 6.60 does not reflect any knowledge about the plaintext m , only about the padded-plaintext $M \leftarrow \text{Pad}_{v1.5}(m)$.

Let M_0 denote the set of messages that satisfy Equation 6.60. The attack proceeds by collecting a sequence of ciphertexts $\{c_i = c \cdot s_i \bmod n\}$ which are all PKCS#1 version 1.5 conforming, i.e.:

$$2B \leq M_i \equiv c_i^d \bmod n < 3B \quad (6.61)$$

Now, since $M_i = c_i^d \bmod n$, we have:

$$M_i = c_i^d \bmod n \quad (6.62)$$

$$= (c \cdot s_i^e)^d \bmod n \quad (6.63)$$

$$= c^d \cdot s_i \bmod n \quad (6.64)$$

$$= M \cdot s_i \bmod n \quad (6.65)$$

By combining Equation 6.65 and Equation 6.61, we have that for every i holds $2B \leq M \cdot s_i \bmod n < 3B$. Now, $B < n$, hence, given s_i , we can easily find the integer q_i such that:

$$2B \leq M \cdot s_i + q_i \cdot n < 3B \quad (6.66)$$

By simple arithmetic, and focusing on the case that $q_i > 0$, we have:

$$\frac{2B - q_i \cdot n}{s_i} \leq M \cdot s_i + q_i \cdot n < \frac{3B - q_i \cdot n}{s_i} \quad (6.67)$$

For each value of s_i , we obtain a different interval bounding M ; clearly, quite quickly, we bound M to a sufficiently small interval, that we can find the correct value by exhaustive search.

Now, the description is pretty accurate, but there is one step we skipped; and that step actually requires much of the attacker's work - even though, as we next show, it is conceptually simple. Can you identify the missing step?

The step we refer to is at the very beginning: *how* does the attacker collect the sequence of PKCS#1 version 1.5 conforming ciphertexts $\{c_i = c \cdot s_i^e \bmod n\}$? Obviously, the attacker can select the s_i values randomly and compute c_i ; however, clearly, usually, the result would *not* be PKCS#1 v1.5 conforming.

For example, consider $c' = c \cdot 2^e \bmod n$. Now, this would result in $M' = c'^d \bmod n = c^d \cdot 2 \bmod n = M \cdot 2$. Hence, the two most significant bytes of M' are either 0x0004 or 0x0005, which are both *not* PKCS#1 v1.5 conforming.

This would happen for most choices of the multiplier s_i ; however, a sufficient fraction would result in c_i that satisfies Equation 6.61. This makes the attack more computationally intensive, true - but it remains practical!

6.6 Public key signature schemes

We now discuss the third type of public-key cryptographic schemes: *signature schemes*, introduced in subsection 1.4.1. Signature schemes consist of three efficient algorithms (KG, S, V), illustrated in Figure 1.6:

Key-generation KG : a randomized algorithm, whose input is the key length l , and which outputs the private signing key s and the public validation key v , each of length l bits.

Signing S : a (deterministic or randomized) algorithm, whose inputs are a message m and the signing key s , and whose output is a signature σ .

Validation V : a deterministic algorithm, whose inputs are a message m , signature σ and validation key v , and which outputs an indication whether this is a valid signature for this message or not.

Figure 1.7 illustrates the process of signing a message (by Alice) and validation of the signature (by Bob). We denote Alice's keys by $A.s$ (for the private signing key) and $A.v$ (for the public validation key); note that this figure assumes that Bob knows $A.v$ - we later explain how signatures also facilitate distribution of public keys such as $A.v$.

Signature schemes have two critical properties, which make them a critical enabler to modern cryptographic systems. First, they *facilitate secure remote exchange in the MitM adversary model*; second, they facilitate *non-repudiation*. We begin by briefly discussing these two properties.

Signatures facilitate secure remote exchange of information in the MitM adversary model. Public key cryptosystems and key-exchange protocols, facilitate establishing of private communication and shared keys between two remote parties, using only public information (keys). However, this still leaves the question of authenticity of the public information (keys).

If the adversary is limited in its abilities to interfere with the communication between the parties, then it may be trivial to ensure the authenticity of the information received from the peer. In particular, many works assume that the adversary is passive, i.e., can only eavesdrop on messages; this is also the basic model for the DH key exchange protocol. In this case, it suffices to simply send the public key (or other public value).

Some designs assume that the adversary is inactive or passive during the *initial* exchange, and use this to exchange information such as keys between the two parties. This is called the *trust on first use (TOFU)* adversary model.

In other cases, the attacker may inject fake messages, but cannot eavesdrop on messages sent between the parties; in this case, parties may easily authenticate a message from a peer, by previously sending a challenge to the peer, which the peer includes in the message.

However, all these methods fail against the stronger *Man-in-the-Middle (MitM)* adversary, who can modify and inject messages as well as eavesdrop

on messages. To ensure security against such attacker, we must use strong, cryptographic authentication mechanisms. One option is to use message authentication codes, however, this requires the parties to share a secret key in advance; if that's the case, the parties could use this shared key to establish secure communication directly.

Signature schemes provide a solution to this dilemma. Namely, a party receiving signed information from a remote peer, can validate that information, using only the public signature-validation key of the signer. Furthermore, signatures also allow the party performing the signature-validation, to first validate the public signature-validation key, even when it is delivered by an insecure channel which is subject to a MitM attack, such as email. This solution is called *public key certificates*.

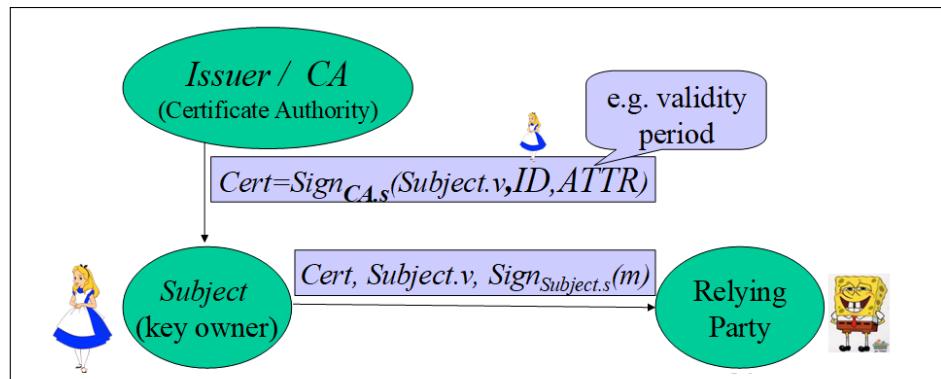


Figure 6.22: Public key certificate issuing and usage processes.

As illustrated in Fig. 6.22, a public key certificate is a signature by an entity called the *issuer* or *certificate authority (CA)*, over the public key of the *subject*, e.g., Alice. In addition to the public key of the subject, *subject.v*, the signed information in the certificate contains attributes such as the validity period, and, usually, an identifier and/or name for the subject (Alice).

Once Alice receives her signed certificate *Cert*, she can deliver it to the *relying party* (e.g., Bob), possibly via insecure channels such as email or the Internet Protocol (IP). This allows the relying party (Bob) to use Alice's public key, i.e., rely on it, e.g., to validate Alice's signature over a message *m*, as shown in Fig. 6.22. Note that this requires Bob to trust this CA and to have its validation key, *CA.v*.

This discussion of certificates is very basic; more details are provided in chapter 8, which discusses *public-key infrastructure (PKI)*, and in Chapter 7, which discusses the important TLS/SSL protocol.

Signatures facilitate non-repudiation. The other unique property of digital signature schemes is that they facilitate *non-repudiation*. Namely, upon receiving a properly signed document, together with a signature by some well-

known authority establishing the public signature-validation key, the recipient is assured that she can convince other parties that she received the document signed properly. This is a very useful property. This property does not hold for message-authentication codes (MAC schemes), where a recipient can validate an incoming message has the correct MAC code, but cannot prove this to another party - in particular, since the recipient is able to compute herself the MAC code for arbitrary messages.

6.6.1 RSA-based signatures

RSA signatures were proposed in the seminal RSA paper [276], and are based on the RSA assumption, with exactly the same key-generation process as for the RSA PKC. The only difference in key generation, is that for signature schemes, the public key is denoted v (as it is used for *validation*), and the private key is denoted s (as it is used for signing).

There are two main variants of RSA signatures: signature with message recovery, and signature with appendix. We begin with signatures with appendix, as in practice, almost all applications of RSA signatures are with appendix; in fact, we present (later) signatures with message recovery mainly since they are often mentioned, and almost as often, a cause for confusion.

RSA signature with appendix. In the (theoretically-possible) case that input messages are very short, and can be encoded as a positive integer which is less than n , we can sign using RSA by applying the RSA exponentiation directly to the message, resulting in the signature σ . In this case, the signature and validation operations are defined as:

$$\begin{aligned} S_s^{RSA}(m) &= (m^s \mod n, m) \\ V_v^{RSA}(\sigma, m) &= \{m \text{ if } m = \sigma^v \mod n, \text{ 'error' otherwise}\} \end{aligned}$$

Above, s is the private signature key, and v is the public validation key. The keys are generated using the RSA key generation process; see subsection 6.5.1

In practice, as discussed in §3.2.6, input messages are of variable length - and rarely shorter than modulus. Hence, real signatures apply the *Hash-then-Sign (HtS)* paradigm, using some cryptographic hash function h , whose range is contained in $[1, \dots, n - 1]$, i.e., allowable input to the RSA function. Applied to the RSA FIL signature as defined above, we have the signature scheme $(S_s^{RSA,h}, V_v^{RSA,h})$, defined as follows:

$$\begin{aligned} S_s^{RSA,h}(m) &= ([h(m)]^s \mod n, m) \\ V_v^{RSA,h}(\sigma, m) &= \{m \text{ if } h(m) = \sigma^v \mod n, \text{ error otherwise}\} \end{aligned}$$

The resulting signature scheme is secure, if h is a CRHF; see §3.2.

This signature scheme is called *signature with appendix* since it requires transmission of both original message and its signature. This is in contrast to a

rarely used variant of RSA signatures which is called *signature with message recovery*, which we explain next. ‘Signature with recovery’ is rarely, if ever, applied in practice; we describe it since there is a lot of reference to it in literature, and in fact, this method causes quite a lot of confusion among practitioners. Hopefully the text below will help to avoid such confusion.

RSA signature with message recovery. RSA signatures with message recovery have the cute property, that they only require transmission of one mod n integer - the signature; the message itself does not need to be sent, as it is *recovered* from the signature. This cute property would result in a small savings of bandwidth, compared to signature with appendix, when both methods are applicable. However, as we explain below, this method is rarely applicable; furthermore, it is cause for frequent confusion.

RSA signatures with message recovery require the use of an invertible *padding* function $R(\cdot)$ which is applied to the messages to be signed. The main goal of R is to ensure sufficient, known *redundancy* (in $R(m)$; this is why we denote it by R). This redundancy, applied to the message before the public key signature operation, should make it unlikely that a random value would appear as a valid signature.

The output of $R(m)$ is used as input to the RSA exponentiation; hence, to ensure recovery, the value of $R(m)$ must be allowed input, i.e., in the range $[1, ;n - 1]$ (where n is the RSA modulus). Note that this implies that length of m has to be even shorter than the length of $R(m)$, since $R(m)$ must contain all of m , as well as the redundancy.

Once R is defined, the signature and validation operations for RSA with Message Recovery (RSAwMR) would be:

$$S_s^{RSAwMR}(m) = [R(m)]^s \pmod{n} \quad (6.68)$$

$$V_v^{RSAwMR}(x) = \{R^{-1}(x^v \pmod{n}) \text{ if defined, else error}\} \quad (6.69)$$

For validation to be meaningful, there should be only a tiny subset of the integers x s.t. $x^v \pmod{n}$ would be in the range of R , i.e., the result of the mapping of some message m . Since there are at most n values of $x^v \pmod{n}$ to begin with, this means that the range of R , i.e., the set of legitimate messages, must be tiny in comparison with n - which means that the message space should be *really* tiny.

In reality, messages being signed are almost always much longer than the tiny message space available for signatures with message recovery. Hence, the use of this method is almost non-existent. In fact, our description of signature schemes (Figure 1.6) assumed that the message is sent along with its signature, i.e., our definition did not even take into consideration schemes like this, which avoid sending the original message entirely.

Note that RSA signatures with message recovery are often a cause of *confusion*, due to their syntactic similarity to RSA encryption. Namely, you may come across people referring to the use of ‘RSA encryption with the private key’ as a method to authenticate or sign messages. What these people really

mean is to the use of RSA signatures with message recovery. We caution to avoid such confusing use of terminology; RSA signatures are usually used with appendix, but even in the rare cases of using RSA signatures with message recovery, *RSA signing is not the same as encryption with the private key!*

6.7 Labs and Additional Exercises

Lab 4 (Breaking textbook and weakly-padded RSA). *In this lab we will break textbook RSA encryption, as well as padded RSA encryption, using specific (weak) padding schemes.*

As for the other labs in this textbook, we will provide Python scripts for generating and grading this lab (`LabGen.py` and `LabGrade.py`). For this lab, the lab-generation script is called `LabGenRSA.py`, and should be provided in the lab-scripts folder. If not yet posted online, professors may contact the author to receive the scripts. The lab-generation script generates random challenges for each student (or team), as well as solutions which will be used by the grading script. We recommend to make the scripts available to the students, as example of how to use the cryptographic functions. It is easy and permitted to modify these scripts to use other languages/libraries or to modify and customize them as desired.

1. To warm up, we perform textbook RSA decryption. In your *lab-input* folder, find files *e1*, *d1*, *n1*, *ma1*, *mb1*, and *cy1*, all generated by the `LabGen.py` script. Use the private decryption key *d1* (and the modulus *n1*) to decrypt *cx1* and *cy1*; save the results in the corresponding files *mx1* and *my1* in the *lab-answers* folder. To allow you to check your program, *one* of these two answers (*mx1* and *my1*) should be identical to one of the two input message files, *ma1* and *mb1*. If you got this one right, most likely you also got the other decryption right.
2. To further warm up, let's also do textbook RSA encryption. Use the public encryption key *e1* (and the modulus *n1*) to encrypt *ma1* and *mb*; save the results in the corresponding files *ca1* and *cb1* in the *lab-answers* folder. Again, *one* of these two answers (*ca1* and *cb1*) should be identical to one of the two input ciphertext files, *cx1* and *cy1*. If you got this one right, most likely you also got the other decryption right.
3. In this item, we *break textbook RSA* encryption. In your *lab-input* folder, find a file *ciphertexts.csv* containing ‘eavesdropped ciphertexts’ (and corresponding identifiers), and file *plaintexts.csv* containing ‘suspected plaintexts’ (and corresponding identifiers), both using the CSV format (check it out). You should be able to identify two plaintexts from *plaintexts.csv* as corresponding to two of the ciphertexts (in *ciphertexts.csv*). The file *pair0-1* in the *lab-input* folder contains one match (as a pair of comma-separated identifiers of plaintext and ciphertext); check that one of the matches you found is the same as the contents of this file. If so, then the other pair you found should also be correct; save it in file *pair0-2* in the *lab-answers* folder, using the same format as of *pair0-1*. Measure also the runtime, and upload it as file *t0*.
4. Now that we see how textbook RSA is insecure, let's try a naive padding. Specifically, let us define the NP1 ('Naive Padding 1') as: $NP1(m) =$

$0x02 \# r \# m$, where m is the (pre-padding) plaintext message and r is a byte whose value is chosen randomly. This is obviously an (overly) simplified version of the *PKCS#1* version 1.5 padding algorithm, $\text{Pad}_{v1.5}(\cdot)$, as defined in Equation 6.58 (subsection 6.5.6).

Reuse the *ciphertexts.csv* and *plaintexts.csv* files. You should again be able to identify two of the plaintexts from *plaintexts.csv* as corresponding to *two* of the ciphertexts (in *ciphertexts.csv*), this time, when applying padding *NP1* to the plaintexts before applying RSA textbook encryption (using e_1 and n_1). The file *pair1-1* in the *lab-input* folder contains one match (as a pair of comma-separated identifiers of plaintext and ciphertext); check that one of the matches you found is the same as the contents of this file. If so, then the other pair you found should also be correct; save it in file *pair1-2* in the *lab-answers* folder, using the same format. Measure also the runtime, and upload it as file *t1*.

5. Repeat the previous item, for random string r containing two, three and four bytes; check your results using the provided *pair2-1*, *pair3-1* and *pair4-1* files, and save your ‘solution pairs’ as files *pair2-2*, *pair3-2* and *pair4-2*. Measure also the runtime; upload it as files *t2*, *t3* and *t4*, and create and submit a graph of the runtime for random string of length 0 to 4. Can you give an approximate function for the runtime? Approximate the runtime if you used this attack for padding of eight random bytes (the minimal number of random bytes required by *PKCS#1* version 1.5)? And for padding of 12 random bytes?
6. Find now in *lab-input* folder the files *c8-1*, *c8-2*, *c12-1* and *c12-2*. These are all encryptions using the *NP1* padding, except using a random string r of length 8 bytes (for *c8-1*, *c8-2*) or 12 bytes (for *c12-1*, *c12-2*). Your goal is to find the corresponding plaintexts.

To assist you in finding the plaintexts, your professors should provide a web server, where you can upload a ciphertext c and receive an indication if this ciphertext is properly padded using *NP1* (and either 8 or 12 random bytes) or not.

To allow you to test your program, you will find the plaintexts *p8-1*, *p12-1* in the *lab-input* folder. Once you find the one or both of the other solutions, upload them in the corresponding files (*p8-2*, *p12-2*) in the *lab-answers* folder.

Upload also the runtimes, as files *t8* and *t12*, and add them to your graph of runtimes from the previous item, comparing them to the runtimes you projected using the attack of the previous item.

7. What is the main difference between your attack and the ‘real’ Bleichenbacher’s attack, and the main security advantage of *PKCS#1* version 1.5 padding over *NP1*?

Exercise 6.12 (Addition/multiplication key exchange is insecure). *Present a sequence diagram similar to Figure 6.5 and Figure 6.6, but using addition or multiplication instead of XOR/exponentiation. Show that the resulting protocol is vulnerable to an eavesdropping attacker.*

Exercise 6.13. *Show that the exponential key exchange (Figure 6.6) is insecure against an eavesdropper, even if the base g used by the protocol is a secret shared between Alice and Bob.*

Exercise 6.14 (Justification for limitations on possible random inputs for key exchange protocols). *Show that in both the Diffie-Hellman key exchange protocol and the Modular Exponentiations key exchange protocol, there will be no gain in security if the parties choose their random inputs (a, b and, for exponential key exchange, also k) from a larger set, say $\{1, \dots, 2p\}$. Show that the same holds for the selection of g .*

Exercise 6.15. *The Diffie-Hellman protocol is a special case of a key exchange protocol, defined by the pair of functions (KG, F) , as introduced in subsection 6.1.3.*

1. *Present the Diffie-Hellman protocol as a key exchange protocol, i.e., define the corresponding (KG, F) functions.*
2. *We presented two assumptions regarding the security of the DH protocol: the Computational-DH (CDH) assumption and the Decisional-DH (DDH) assumption. Show that one of these assumption does not suffice to ensure key-indistinguishability? What about the other one?*

Exercise 6.16. *It is proposed that to protect the DH protocol against an imposter, we add an additional ‘confirmation’ exchange after the protocol terminated with a shared key $k = h(g^{ab} \bmod p)$. In this confirmation, Alice will send to Bob $MAC_k(g^b)$ and Bob will respond with $MAC_k(g^a)$. Show the message-flow of an attack, showing how a MitM (Man-in-the-Middle) attacker can impersonate as Alice (or Bob). The attacker has ‘MitM capabilities’, i.e., it can intercept messages (sent by either Alice or Bob) and inject fake messages (incorrectly identifying itself as Alice or Bob).*

Exercise 6.17. *Suppose that an efficient algorithm to find discrete log is found, so that the DH protocol becomes insecure; however, some public-key cryptosystem $(\mathcal{G}, \mathcal{E}, \mathcal{D})$ is still considered secure, consisting of algorithms for, respectively, key-generation, encryption and decryption.*

1. *Design a key-agreement protocol which is secure against an eavesdropping adversary, assuming that $(\mathcal{G}, \mathcal{E}, \mathcal{D})$ is secure (as a replacement to DH).*
2. *Explain which benefits the use of your protocol may provide, compared with simple use of the cryptosystem $(\mathcal{G}, \mathcal{E}, \mathcal{D})$, to protect the confidentiality of messages sent between Alice and Bob against a powerful MitM adversary. Assume Alice and Bob do have known public keys.*

Exercise 6.18. Assume that there is an efficient (PPT) attacker \mathcal{A} that can find a specific bit in $g^{ab} \bmod p$, given only $g^a \bmod p$ and $g^b \bmod p$. Show that the DDH assumption does not hold for this group, i.e., that there is an efficient (PPT) attacker \mathcal{A} that can distinguish, with significant advantage over random guess, between $g^{ab} \bmod p$ and between g^x for x taken randomly from $[1, \dots, p - 1]$.

Exercise 6.19. It is frequently proposed to use a PRF as a Key Derivation Function (KDF), e.g., to extract a pseudo-random key $k' = \text{PRF}_k(g^{ab} \bmod p)$ from the DH exchanged value $g^{ab} \bmod p$, where k is a uniform random key (known to attacker). In particular, in subsection 6.3.1, a variant of the Auth-DH protocol uses a function f assumed to fulfill both the PRF requirements and the KDF requirements. In this exercise, we explore alternatives.

1. Let f be a secure PRF; note that f may be a KDF or not. Show a function f' which is (1) also a PRF and (2) not a secure KDF.
2. Let g be a secure KDF; note that g may be a PRF or not. Show a function g' which is (1) also a KDF and (2) not a secure PRF.
3. Present a variant of the Auth-DH protocol, as a modification of Figure 6.11, which uses a PRF (instead of MAC) and a KDF (instead of KDF). Explain why this variant is secure, when the key of the PRF and the KDF are chosen independently (using uniform distribution).
4. Let f be a secure PRF and g be a secure KDF. Show functions f' , g' such that f' is a PRF and g' is a KDF, and furthermore the following holds: the protocol in the previous item may be insecure when used with f' and g' , if both of them use the same symmetric master key MK . Note: this is an example of the principle of key separation (Principle 10).

Exercise 6.20 (How not to ensure resilient key exchange). Fig. 6.23 illustrates a slightly different protocol for authenticating the DH protocol, using a changing key k_i (to ensure resilient key exchange). Present a sequence diagram showing that this protocol is not secure.

Exercise 6.21. The protocol in Fig. 6.24 is an (incorrect) attempt at a robust-combiner authenticated DH protocol.

1. Show a sequence diagram for an attack showing that this variant is insecure.
2. Show a simple fix that achieves the goal (robust combiner authenticated DH protocol).

Exercise 6.22. Assume it takes 10 seconds for any message to pass between Alice and Bob.

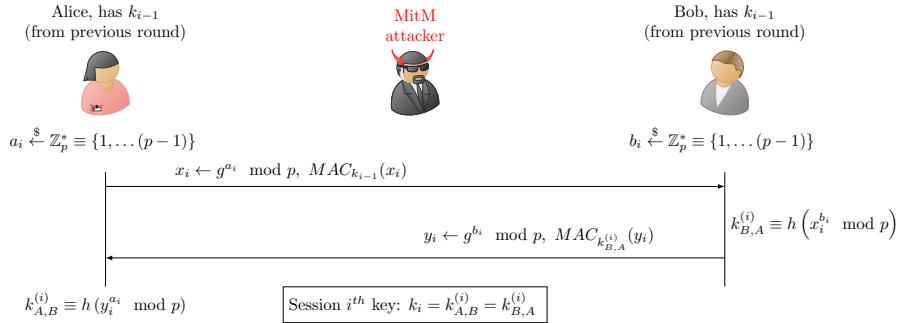
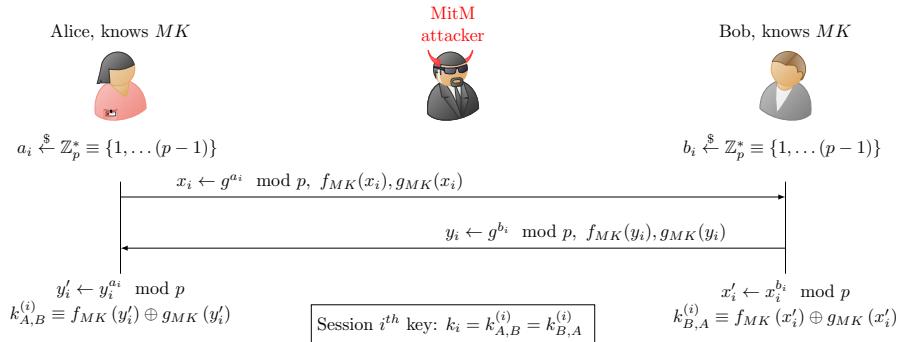
Figure 6.23: How *not* to ensure resilient key exchange: illustration for Ex. 6.20

Figure 6.24: Insecure ‘robust-combiner’ authenticated DH protocol, studied in Exercise 6.21.

1. Assume that both Alice and Bob initiate the ratchet protocol (Fig. 6.12) every 30 seconds. Draw a sequence diagram showing the exchange of messages between time 0 and time 60 seconds; mark the keys used by each of the two parties to authenticate messages sent and to verify messages received.

2. Repeat, if Bob’s clock is 5 seconds late.

Exercise 6.23. In the DH ratchet protocol, as described (Fig. 6.12), the parties derive symmetric keys $k_{i,j}$ and use them to authenticate data (application) messages they exchange between them, as well as the first message of the next handshake.

1. Assume a chosen-message attacker model, i.e., the attacker may define arbitrary data (application) messages to be sent from Alice to Bob and vice versa at any given time, and ‘wins’ if a party accepts a message never sent by its peer (i.e., that message passes validation successfully). Show that, as described, the protocol is insecure in this model.

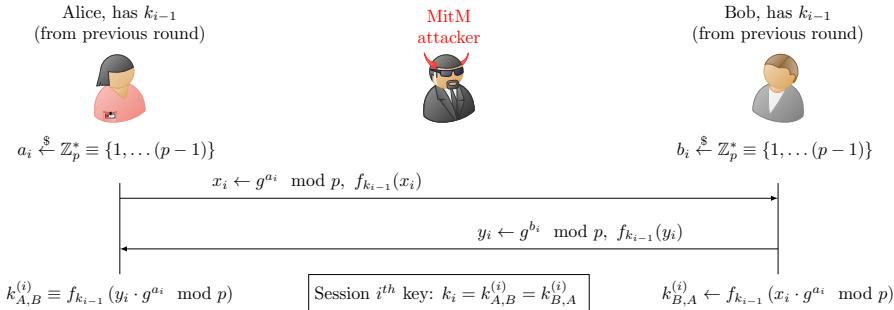


Figure 6.25: Insecure variant of the DH-Ratchet Protocol, for Ex. 6.26.

2. Propose a simple, efficient and secure way to avoid this vulnerability, by only changing how the protocol is used - without changing the protocol itself.

Exercise 6.24. The DH protocol, as well as the ratchet protocol (as described in Fig. 6.12), are designed for communication between only two parties.

1. Extend DH to support key agreement among three parties.
2. Similarly extend the ratchet protocol.

Exercise 6.25 (DH-Ratchet). Figure 6.12 shows the DH-Ratchet protocol, where the key used to authenticate the DH exchange as well as the data messages is changing periodically (as indicated), and where f is a PRF (Pseudo-Random Function). Assume that this protocol is run daily, from day $i = 1$, and where k_0 is a randomly-chosen secret initial master key, shared between Alice and Bob; messages on day i are encrypted and authenticated using session key k_i , by selecting a random string r and sending r and $f(r||f_{k_i}(r) \oplus m)$. An attacker can eavesdrops on the communication between the parties on all days, and on days 3, 6, 9, ... it can also spoof messages (send messages impersonating as either Alice or Bob), and act as Man-in-the-Middle (MitM). On the fifth day ($i = 5$), the attacker is also given the initial master key Mk_0 .

- Explain why sending r and $f(r||f_{k_i}(r) \oplus m)$ ensures authenticity and confidentiality, provided that k_i is secret.
- What are the days whose messages the attacker will be able to decrypt (find out) upon day ten?
- Show a sequence diagram of the attack, and list calculations done by the attacker.

Exercise 6.26 (Insecure variant of DH-Ratchet). Figure 6.25 shows a variant of the DH-Ratchet protocol, using a (secure) pseudorandom function f to derive the session key.

1. Does this protocol ensure forward-secrecy (FS)? If so, explain; if not, present sequence diagram of attack.
2. Repeat, for PFS.
3. Repeat, for Recover-Security (RS).
4. Repeat, for PRS.

Exercise 6.27 (GSM). Design a more secure variant of the GSM handshake protocol, which foils the attack described in Exercise 5.17; the mobile and visited network can identify support of this variant by referring to it as a new cipher, say A5/33. The actual data encryption can use any secure shared-key encryption; the critical improvement is to the negotiation, namely, to prevent attacks as in Exercise 5.17.

The change may involve one or few new handshake messages between mobile and visited network, but no change to the rest of the GSM network, in particular, no change to the home network. Your solution may require the mobile and/or visited network to use additional cryptographic mechanisms, including public key mechanisms, but only during handshake. Hint: your solution should set the key to be used by A5/3, using cryptographic mechanism(s) we learned.

Exercise 6.28. We saw that El-Gamal encryption (Equation 6.36) may be re-randomized, using the recipient's public key, and mentioned that this may be extended into an encryption scheme which is universally re-randomizable, i.e. where re-randomization does not require the recipient's public key. Design such encryption scheme. Hint: begin with El-Gamal encryption, and use as part of the ciphertext, the result of encrypting the number 1. Or see [146].

Exercise 6.29. A public-key cryptosystem is IND-rCCA secure, if it passes the IND-CPA test, when the attacker is restricted to avoid any ciphertext queries whose output is the challenge message m^* [75]. Show that:

1. The El-Gamal PKC is not IND-rCCA secure.
2. Textbook RSA is not IND-rCCA secure.

Exercise 6.30. The RSA algorithm calls for selecting e and then computing d to be its inverse ($\mod \phi(n)$). Explain how the key owner can efficiently compute d , and why an attacker cannot do the same.

Exercise 6.31. The RSA key generation algorithm requires the selection of two large primes p, q . Would it be secure to save time by using $p = q$? Or first choose p , then let q be the next-largest prime?

Exercise 6.32 (Tiny-message attack on textbook RSA). We discussed that RSA should always be used with appropriate padding, and that 'textbook RSA' (no padding) is insecure, in particular, is not randomized so definitely does not ensure indistinguishability.

1. Show that textbook RSA may be completely decipherable, if the message length is less than $|n|/e$. (This is mostly relevant for $e = 3$.)
2. Show that textbook RSA may be completely decipherable, if there is only a limited set of possible messages.
3. Show that textbook RSA may be completely decipherable, if the message length is less than $|n|/e$, except for a limited set of additional (longer) possible messages.

Exercise 6.33. Consider the use of textbook RSA for encryption (no padding). Show that it is insecure against a chosen-ciphertext attack.

Exercise 6.34. Consider a variation of RSA which uses the same modulus N for multiple users, where each user, say Alice, is given its key-pair $(A.e, A.d)$ by a trusted authority (which knows the factoring of N and hence $\phi(N)$). Show that one user, say Mal, given his keys $(M.e, M.d)$ and the public key of other users say $A.e$, can compute $A.d$. Note: recall that each user's private key is the inverse of the public key ($\mod \phi(n)$, e.g., $M.e = M.d^{-1} \mod \phi(n)$).

Exercise 6.35. Public-key algorithms often use term ‘public key’ to refer to only one component of the public key. For example, with RSA, people often refer to e as the public key, although the actual RSA public key consists of the pair (e, n) , i.e., also includes the modulus n .

Consider an application which receives an RSA signature (e_A, n) , where e_A is the same as in the public key (e_A, n_A) of user Alice, but $n \neq n_A$; however, the application still concludes that this is a valid signature by Alice. Show how this allows an attacker to trick the recipient into believing - incorrectly - that an incoming message (sent by the attacker) was signed by Alice.

Note: similar situation exists with other public key algorithms, e.g., elliptic curves, where the public key consists of a specification of a curve and of a particular ‘public point’ on the curve, but often people refer only to the point as if it is the (entire) public key. In particular, this led to the ‘Curveball’ vulnerability in the Windows certificate-validation mechanism [296], which was due to validation of only the ‘public point’ and use of the curve selected by the attacker.

Exercise 6.36. You are given textbook-RSA ciphertext $c = 281$, with public key $e = 3$ and modulus $n = 3111$. Compute the private key d and the message $m = c^d \mod n$.

Hint: it is probably best to begin by computing the factorization of n . \square

Exercise 6.37. Consider the use of textbook RSA for encryption as well as for signing (using hash-then-sign), with the same public key e used for encryption and for signature-verification, and the same private key d used for decryption and for signing. Show this is insecure against chosen-ciphertext attacks, i.e., allows either forged signatures or decryption.

Exercise 6.38. *The following design is proposed to send email while preserving sender-authentication and confidentiality, using known public encryption and verification keys for all users. Namely, assume all users know the public encryption and verification keys of all other users. Assume also that all users agree on public key encryption and signature algorithms, denoted E and S respectively.*

When one user, say Alice, wants to send message m to another user, say Bob, it computes and sends: $c = E_{B.e}(m + \text{'Alice'} + S_{A.s}(m))$, where $B.e$ is Bob's public encryption key, $A.s$ is Alice's private signature key, and 'Alice' is Alice's (unique, well-known) name, allowing Bob to identify her as the sender. When Bob receives this ciphertext c , it first decrypts it, which implies it was sent to him. To validate that the message was sent by Alice, he looks up Alice's public verification key $A.v$, and verifies the signature.

1. *Explain how a malicious user Mal can cause Bob to believe it received a message m from Alice, although Alice never sent that message to Bob. (Alice may have sent a different message, or sent that message to somebody else.)*
2. *Propose a simple, efficient and secure fix.*

Exercise 6.39 (Combining public key signatures and encryption). *Many applications require both confidentiality, using recipient's public encryption key, say $B.e$, and non-repudiation (signature), using sender's verification key, say $A.v$. Namely, to send a message to Bob, Alice uses both her private signature key $A.s$ and Bob's public encryption key $B.e$; and to receive a message from Alice, Bob uses his private decryption key $B.d$ and Alice's public verification key $A.v$.*

1. *It is proposed that Alice will select a random key k and send to Bob the triplet: $(c^K, c^M, \sigma) = (E_{B.e}(k), k \oplus m, \text{Sign}_{A.s}(\text{'Bob'} \# k \oplus m))$. Show this design is insecure, i.e., a MitM attacker may either learn the message m or cause Bob to receive a message 'from Alice' - that Alice never sent.*
2. *Propose a simple, efficient and secure fix. Define the sending and receiving process precisely.*
3. *Extend your solution to allow prevention of replay (receiving multiple times a message sent only once).*

Note: signcryption schemes combine the public key signature and encryption operations, possibly with greater efficiency than applying separately the encryption and signing operation.

Chapter 7

The TLS/SSL protocols for web-security and beyond

In this chapter, we discuss the *Transport-Layer Security (TLS)* protocol, which is the main protocol used to secure connections over the Internet - and, in particular, web-communication. We believe that TLS is the most studied applied cryptographic protocol; it is widely deployed in many applications and has huge impact on the security of the Internet. Its extensive study has resulted in many attacks and subsequent countermeasures, defenses, improvements and several versions.

History, versions, and this chapter. Let us begin with a few words on the history of TLS. The TLS standards are defined by the Internet Engineering Task Force (IETF), as an evolution of the *Secure Socket Layer (SSL)* protocols developed by the Netscape corporation. In fact, version 3 of SSL (SSLv3) is closely related to versions 1.0 to 1.2 of TLS, and less similar to version 2 of SSL. Therefore, the history of TLS really begins with the SSL protocols.

The beginning of SSL was around 1994, with the beginning of the commercial use of the World Wide Web (WWW). Possibly the first company to focus on the commercial potential of the web was the Netscape corporation, established in 1994. At the time, a major concern was the ability to perform online-purchases securely. Credit cards were quickly recognized as an appropriate payment method, since they were already used widely for phone purchases and mail orders; such remotely-authorized transactions were referred to as *card not present*, to identify the risk due to reliance on card details without visual confirmation of the physical card and handwritten signature, which were required for the more common (at the time) *card present* transactions. However, the transmission of credit card information over the Internet was considered less secure, than the somewhat-protected transmission in a phone call or by mail. In both phone and physical mail, there is some level of authentication of the merchant, since the customer initiated the phone call or addresses the physical mail; but Internet communication may be viewed by different providers and,

depending on technology, even other users. A secure solution was considered essential for the commercial use of the Web.

Netscape came out with the first protocol to protect credit card transactions over the Web - the SSL protocol. SSL initial goal was, basically, to provide security for credit card transactions, which will be comparable to the security of credit card transactions performed remotely, over phone or mail (referred to as ‘card not present’ transactions). SSL-protected web transactions became, basically, a new way to perform ‘card not present’ transactions. The focus of SSL on this goal gave it important advantages over alternatives protocols for credit-card payments over the web, developed about concurrently: the SET protocol (by Microsoft, Visa and later also Mastercard) [6] and the iKP protocol developed by IBM [28, 29]. Both SET and iKP had a more ambitious goal (compared to SSL): to provide security comparable to ‘card present’ transactions, by having the client’s device digitally-sign each purchase, as a secure alternative to the handwritten signature on a credit card slip. As a result, they were significantly more complex to understand, implement and deploy, compared to SSL.

Three main advantages helped SSL to quickly become a success. First, SSL was quickly implemented and deployed by Netscape, whose browser was, by far, the most popular at the time, with a large lead over all other browsers combined. Second, while SET and iKP provided better security for credit card transactions, they were limited to this credit-card application; in contrast, SSL could be used for other applications requiring secure client to server communication, not only for credit card purchases. Finally, SSL is *simple* - simple in its concept, simple to implement, simple to integrate in applications, most notably, in a browser, and, most significantly, simple for adoption. Specifically, SET and iKP required adoption by credit card processors as well as merchants and customers, with private keys and certificates for each party; while SSL required only adoption by merchants and customers, and only merchants needed private keys and certificates. Indeed, for all parties (the merchant, customer and credit-card processor), once the customer uses an SSL-enabled browser and the merchant offers an SSL-enabled website, they basically operate as in other card-not-present scenarios; no additional change to their systems or processes is required.

The importance of simplicity and ease of deployment and use for applied security mechanisms cannot be overstated, and we will return to it when we introduce, later in this chapter, the *Keep it Simple and Secure (KISS) principle* (Principle 14). For example, while SSL supports both server-authentication and client-authentication, it is usually deployed with only server authentication, requiring only servers (merchants) to obtain public key certificates; even today, only few clients obtain certificates (for client authentication).

The very first versions of SSL were not published officially, until June 1995, when Netscape published SSL version 2 (SSLv2) [168]. Later in 1995, Microsoft changed strategy; they published and implemented the *Private Communication Technology (PCT) Protocol* [41], which is similar to SSLv2.

SSLv2 had significant design vulnerabilities. In November 1996, Netscape

published the much-improved, and quite different, SSLv3 [131] (later published as [130]). The specification published was quite complete, allowing independent interoperable implementations.

Also in 1996, the IETF established a working group to develop an agreed standard protocol to replace the proprietary-developed SSL and PCT protocols. To avoid arguments on which of the two names should be used, a new name was chosen: the *TLS* (*Transport Layer Security*) protocol. However, TLS 1.0 [98], the first standard produced by the TLS working group, was closely based on SSLv3.

Unfortunately, although SSLv3 and TLS 1.0 addressed some of SSLv2's vulnerability, they still had serious vulnerabilities, as well as non-security limitations. In April 2006, the TLS working group of the IETF defined TLS 1.1 [99] to fix the issues discovered in TLS 1.0. However, additional vulnerabilities and concerns were discovered, motivating another release: TLS 1.2 [100] (published in August 2008). These three TLS versions (1.0 to 1.2) were all quite similar to SSLv3, only fixing clearly-exploitable vulnerabilities and adding features.

After vulnerabilities were discovered even in TLS 1.2, the working group decided to do a major redesign. This took about 10 years; the IETF published TLS 1.3 [272] in August 2018, and this is the currently used version of TLS. In contrast to previous designs, the TLS 1.3 designers gave preference to mechanisms with proven security properties; ideally, we would like the complete TLS protocol to be provably secure. While a complete proof of security was not yet published for TLS 1.3, there are encouraging and important partial results. As a result of this stronger emphasis on security, as well as due to significant changes to improve performance (mainly, reduce latency), TLS 1.3 is a major deviation from the previous versions.

Our discussion, in the rest of this chapter, follows these three major versions of SSL/TLS. We first describe SSLv2, then the SSLv3 and TLS 1.0 to 1.2 design, and finally the TLS 1.3 design. One reason to describe the older versions is to learn about protocol vulnerabilities and attacks. Another reason is that these attacks are often still relevant, for two reasons. First, many clients and servers still support outdated versions. Second, several *downgrade* attacks (subsection 5.6.3) break implementations of newer versions of SSL/TLS, by exploiting support for older, vulnerable versions.

TLS: most used, most versatile, most studied - and most attacked. The SSL/TLS protocol is arguably the most ‘successful’ security protocol - it is definitely very widely used. One reason for this wide use is that SSL/TLS is widely applicable; it is used in more diverse scenarios and environments than any other security protocol. Many extensions and changes have been proposed over the years, allowing the use of SSL/TLS in new scenarios and satisfying new requirements, as well as improving security.

Some of these extensions and changes were adopted as an inherent part of a new revision of the protocol, and many others can be deployed using the built-in extensions mechanism (subsection 7.4.3), which became a standard part of TLS

beginning with version TLS 1.1. Indeed, we believe TLS is probably the applied cryptography protocol which was most widely studied and analyzed, with many vulnerabilities exposed and fixed; this gives us considerable confidence in the security of the (later versions of) TLS.

From a security point of view, this popularity is a double-edged sword. On the one hand, this wide popularity motivates extensive efforts by the ‘white-hat’ security community, including researchers from academia and industry, to identify vulnerabilities and improve the security of the protocols and their implementations. This published research resulted in significant improvements to the security of SSL/TLS; many attacks and corresponding countermeasures were published by researchers, and new versions of the protocols were gradually more and more secure, culminating with TLS 1.3, which has significant design changes whose goal is to improve security.

On the other hand, this wide popularity also implies that ‘black-hat crackers’ have a strong motivation to find vulnerabilities in the SSL/TLS protocols and in their popular implementations. In fact, the desire to ‘break’ secure connections, may even motivate powerful organizations, e.g., the NSA, to invest extensive efforts in ‘injecting’ intentional, hidden vulnerabilities (*cryptographic backdoors*) into the specifications and implementations of SSL/TLS and of other popular cryptographic systems, libraries, protocols and standards.

One example of what may be a cryptographic backdoor, is the *Dual-EC Deterministic Random Bit Generator* (DRBG), found vulnerable in [82]. The Dual-EC DRBG was included in NIST, ANSI and ISO/IEC standards, and implemented in the widely-used BSAFE cryptographic toolkit from RSA (also used by implementations of SSL/TLS). There are claims that the NSA created and promoted the Dual-EC DRBG as a cryptographic backdoor, and allegedly even paid RSA to make it the default pseudorandom generator in BSAFE. One evidence for NSA’s involvement came from the NSA memos exposed by Edward Snowden in 2013, which also indicated that NSA spends \$250 million per year to insert backdoors into software, hardware and standards; see [46, 323].

Some of this purported effort to insert trapdoors into standards, seems to have been directed at TLS. In particular, consider [275], a proposal for a TLS extension by Eric Rescorla (as a consultant to the US government) and Margaret Salter (an NSA employee). The purported goal of this extension was to increase the number of random bits exchanged during the TLS handshake. However, these additional random bits seem to significantly improve the efficiency of the Dual-EC DRBG attack. This may indicate that this was another attempt to insert a cryptographic trapdoor - in this case, to the TLS specifications; see [46].

This widespread use of SSL/TLS also has important implications for learning and teaching SSL/TLS. Obviously, the importance of SSL/TLS motivates studying it; furthermore, the evolution of SSL/TLS, and the different attacks and countermeasures, are a valuable, interesting lesson, which can help to identify and avoid vulnerabilities in different protocols and systems. On the other hand, this also means that there is an excessive wealth of important and interesting information - indeed, entire books were dedicated to cover SSL/TLS, e.g., [254, 271], and even they do not cover all aspects and attacks. We have

tried to maintain a reasonable balance; however, there were many hard choices and surely there is much to improve. As in other aspects, your feedback would be appreciated.

7.1 Introduction to TLS and SSL

7.1.1 TLS/SSL: High-level Overview

The TLS and SSL protocols were originally designed to secure the communication between a web-browser and a web-server, and, while they are now widely deployed for additional applications, web-security remains their main application. We present a highly-simplified overview of this typical use-case in Figure 7.1.

In Figure 7.1, we show how Alice, a web user, surfs to the SSL/TLS protected website *https://b.com*; we focus on the simpler variants of SSL/TLS, which are based on RSA encryption, and denote *b.com*'s public RSA encryption key by *B.e*. Notice that the URL of SSL/TLS protected websites begins with the protocol name *https*, rather than the protocol name *http*, used for unprotected web sites. The process consists of the following steps:

Step 0 (in advance): Web server obtains certificate. Before the server of *b.com* can provide SSL/TLS service, it needs to obtain a *certificate* for its public key *B.e*, signed by a *certificate authority (CA)* trusted by the client (in this case, browser). For that purpose, the server sends to the CA, in flow 0.*a*, its domain name, *b.com*, its public encryption key *B.e*, and optionally other identifiers (*IDs*). The CA should validate that the server indeed ‘owns’ domain *b.com*, and is associated with any additionally provided identifiers (the optional *IDs*). If validation passes, the CA signs the certificate using its private signing key *CA.s*, and sends it to the server (flow 0.*b*). The certificate contains the public key *B.e*, the domain *b.com*, the optional *IDs* and other ‘administrative’ information, such as validity period. For more details about certificates see Chapter 8; in particular, the validation process is discussed in subsection 8.2.8.

Step 1: client requests website *https://b.com*. The user (Alice) enters the desired *Universal Resource Locator (URL)*, *https://b.com*. The URL consists of the protocol (*https*) and the domain name of the desired web-server (*b.com*); in addition, the path may contain identification of a specific *path* and *object* in the server. In this example, Alice does not specify any specific path or object; and the browser considers this a request for the default object *index.html*. The choice of the *https* protocol, instructs the browser to open a *secure connection*, i.e., send the HTTP requests over an SSL/TLS session, rather than directly over an unprotected TCP connection. The request may be specified in one of three ways: (1) by the user ‘typing’ the URL into the address bar of the browser, i.e., ‘manually’, (2) ‘semi-automatically’, by the user clicking on an *hyperlink* or bookmark which specified this URL, or (3) by an instruction from the webpage currently displayed by the browser.

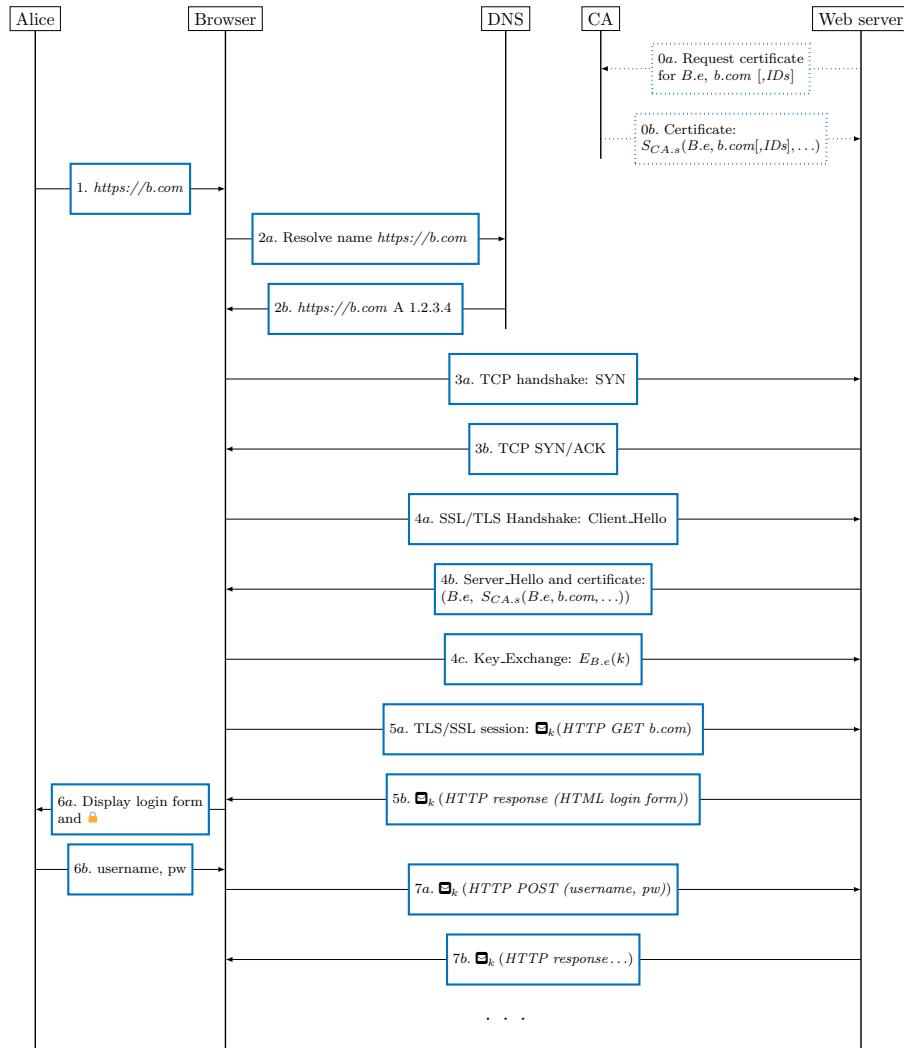


Figure 7.1: Simplified overview of the operation of SSL/TLS, to secure the connection between the browser and the web-server, using RSA for key exchange.

Step 2: resolving domain name into IP address. To communicate with the *b.com* web server, the browser needs the IP address of the server. The *Domain Name System (DNS)* provides resolution (mapping) from domain names to IP addresses. We simplify this process into a request from the browser to the DNS (flow 2a), and a response from the DNS to the browser specifying the IP address (flow 2b). The step is skipped if the IP address is already known, typically, cached from a previous connection.

Step 3: TCP handshake. The SSL/TLS protocol runs over the *TCP* (*Trans-*

mission Control Protocol) protocol, which provides important services such as reliability and congestion/flow control. The first two flows of a TCP connections are called the *TCP handshake*, and contain only control signals, no data. The first flow is referred to as *TCP SYN* (flow 3a), and the second flo is referred to as *TCP SYN/ACK* (flow 3b).

Step 4: SSL/TLS handshake. The SSL/TLS protocol also begins with a *handshake*, i.e., few control flows, which establishes the secure connection. Different versions of SSL/TLS support different handshakes, which we describe in following sections; we simplify the handshake based on RSA encryption in Figure 7.1. All SSL/TLS begin with the *Client_Hello* message (flow 4a). The server responds with *Server_Hello* and the certificate (flow 4b). This provides the browser with the server's public encryption key ($B.e$); the browser selects a random key, here denoted simply as k , and shares it with the server by encrypting it using $B.e$ and sending the encryption ($E_{B.e}(k)$) to the web server (flow 4c, the *Key_Exchange* message).

Step 5: the SSL/TLS session (record protocol), initial webpage. At this point, the browser and server can communicate securely, with their messages protected using the SSL/TLS *record protocol* and the key they shared (which we denoted k). We denote the protection of the record protocol by the envelope symbol, with the key as subscript, i.e., $\boxtimes_k(\cdot)$. The browser first sends an HTTP GET request, requesting the *index.html* webpage (flow 5a); the server responds by sending back the page, written in HTML (Hypertext Markup Language), e.g., a login form (flow 5b).

Step 6: page displayed to user. Flow 6a represents the browser displaying the webpage to the user (Alice), together with a few *security indicators* such as a padlock . Flow 6b represents the user entering user name and password.

Step 7: additional HTTP requests and response. Flows 7a and 7b are examples of additional HTTP requests and responses, protected by the SSL/TLS record protocol. In flow 7a, the browser sends the username and password; the interaction typically continues with the server HTTP response (flow 7b) and additional requests and responses (not shown).

7.1.2 TLS/SSL: security goals

TLS and SSL are designed to ensure security between two computers, usually referred to as a *client* and a *server*, in spite of attacks by a MitM (Man-in-the-Middle) attacker. The goals include:

Key exchange: securely setup a secret shared key, preventing exposure of this key to a MitM attacker.

Server authentication: authenticate the identity of the server, i.e., assure the client that it is communicating with the right server.

Client authentication: authenticate the identity of the client. Client authentication is optional; in fact, TLS/SSL is usually used without client authentication, allowing an anonymous, unidentified client to connect to the server. When client authentication is desired, it is usually performed by sending a *secret credential* within the TLS/SSL secure connection, such as a password or cookie.

Connection Integrity: Ensure that the communication received by one party, is exactly identical to the communication sent by the peer (in spite of a MitM attacker). This includes preventing message re-ordering and truncating attacks. Of course, an attack - or even benign failure - could disrupt communication, leaving some information sent but never received by the peer, but TLS/SSL would detect such events.

Connection confidentiality: Ensure that a MitM attacker cannot learn anything about the information sent between the two parties, except for the ‘traffic pattern’ - amount of information sent/received information.

Perfect forward secrecy (PFS): From version 3 of SSL, the SSL/TLS handshakes support the (optional) use of authenticated DH key agreement, which ensures perfect forward secrecy (PFS), as discussed in subsection 6.3.1.

Crypto-agility: We say that a cryptographic protocol, such as SSL/TLS, provides *cryptographic agility* or *crypto-agility*, if it allows the parties to select the specific cryptographic algorithms they use for a given function (e.g., block cipher, hash function or signatures). We introduced and discussed the importance of crypto-agility in subsection 5.6.2; in particular, crypto-agility is essential, when a vulnerability is found or suspected in a particular algorithm. SSL/TLS supports crypto-agility; it allows the cryptographic algorithms to be negotiated in each session, through *cipher suite negotiation*.

All versions of SSL/TLS were designed to meet these goals, with the exception of SSLv2 where Perfect Forward Secrecy (PFS) was not a goal. Of course, the goals may not be actually met, due to different vulnerabilities; we present some of the important vulnerabilities in this chapter, and in particular, see Table 7.2.

Robust cryptography. Crypto-agility helps us to recover *after* a possible cryptographic vulnerability is *identified*; until then, the system could be vulnerable. In contrast, *cryptographic robustness* requires the protocol to ensure its security properties, *even while* some of its cryptographic modules or assumptions are weak. In particular, the protocol should maintain security even if its sources of randomness are imperfect, and when one of its cryptographic algorithms ('building blocks') is vulnerable, i.e., deploy *robust combiner* design. Cryptographic robustness was not defined as an explicit goal in SSL/TLS, however, there have been several elements of robust cryptographic designs since SSLv3.

7.1.3 SSL/TLS: Engineering goals

In addition to the security goals, the success of TLS/SSL is largely due to its focus - from the very first versions - on the generic ‘engineering goals’, applicable to any system, of *efficiency*, *ease of deployment and use*, and *flexibility*. By addressing these goals, TLS/SSL is widely used and applicable in a very wide range of applications and scenarios. Let us briefly discuss these three *engineering goals*.

Efficiency - and session resumption. Efficiency is always a desirable goal. In the case of TLS/SSL, there are two main efficiency considerations: *computational overhead* and *latency*. In terms of *computational overhead*, the main consideration is minimizing the computationally-intensive public-key operations. To minimize public-key operations, once the handshake establishes a shared key (using public key operations), the parties may reuse this key to establish future connections without requiring additional public-key operations. We refer to the set of connections based on the same public-key exchange as a *session*, and a handshake that reuses the pre-exchanged shared key as a *session-resumption* handshake.

In terms of minimizing *latency*, the main consideration is to minimize the number of round trip exchanges. End-to-end delays are typically on the order of tens to hundreds milliseconds, which is usually much higher than the transmission delays, esp. for the limited amount of information sent in TLS/SSL exchange. Reducing the number of round trips became even more important as transmission speeds increased; this is reflected by the fact that until TLS 1.3, all designs had a fixed number of two round-trips to complete the handshake, only then allowing the client to send a protected message (already in the third exchange). In contrast, a TLS 1.3 handshake requires only a single round-trip (before sending a protected message), and even allows the clients to send a request already in the first exchange (with some limitations and somewhat reduced security properties, see later).

A more minor efficiency consideration is minimization of bandwidth; this is mainly significant in scenarios where bandwidth is limited, such as very noisy wireless connections.

Extensibility and versatility. Extensibility is always important - definitely for a widely deployed security protocol such as SSL/TLS, which is used in diverse scenarios and environments. Indeed, part of the success of SSL/TLS derives from its extensibility and versatility; the protocol supports many optional mechanisms, e.g., client authentication, and flexibility such as crypto-agility (Principle 11). Furthermore, from TLS version 1.1 (and even earlier for some implementations), the TLS protocol supports a built-in extension mechanism, providing even greater flexibility (subsection 7.4.3).

Ease of deployment and use. Finally, the success and wide-use of the SSL/TLS protocols are largely due to their ease of deployment and usage. As

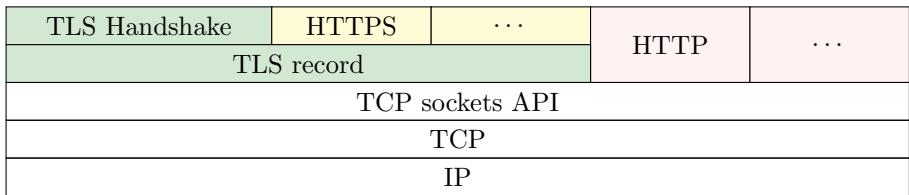


Figure 7.2: Placement of TLS/SSL in the TCP/IP protocol stack. The TLS/SSL record protocol (first box in top line, in green) establishes keys for, and also uses, the TLS record layer protocol (first box in second line, also in green). The HTTPS protocol, and other application protocols that use the TLS/SSL record protocol, are in two middle boxes of the top line (in yellow). Application protocols that do *not* use TLS/SSL for security, including the HyperText Transfer Protocol (HTTP), are in the two last boxes in the two top lines (in pink). These protocols, as well as TLS/SSL itself, all use the TCP protocol, via the sockets library layer. TCP ensures reliable communication, on top of the (unreliable) Internet Protocol (IP).

shown in Figure 7.2, the TLS/SSL protocol is typically implemented ‘on top’ of the popular *TCP sockets API*, and then used by applications, directly or via the *HTTPS* or other protocols. This architecture makes it easy to install and use SSL/TLS, without requiring changes to the operating-system and kernel. This is in contrast to some of the other communication-security mechanisms, in particular the IPsec protocol [105, 129], which, like TLS, is also an IETF standard. The ease of deployment and use of SSL/TLS is probably the reason that TLS has become an almost-universal security substrate for many systems, even where other protocols may have advantages. For example, IPsec is probably better for use for *Virtual Private Networks (VPNs)*, yet SSL/TLS VPNs are more widely deployed.

7.1.4 TLS/SSL and the TCP/IP Protocol Stack

See Figure 7.2 for the placement of the TLS protocols with respect to the TCP/IP protocol stack, and Figure 7.3 for a typical connection.

[This subsection is yet to be written; this material is well covered in many textbooks on networking, e.g., [202].]

7.2 The SSL/TLS Record Protocol

In this section we begin our in-depth discussion of the SSL/TLS protocols. Specifically, we focus on the *record protocol* component of SSL/TLS; this protocol protects the communication, using symmetric cryptography, i.e., encryption, authentication (MAC) and/or authenticate-encryption. The symmetric key used by the record protocol, must be previously setup, securely, by the *handshake protocol*, which we discuss in the following sections.

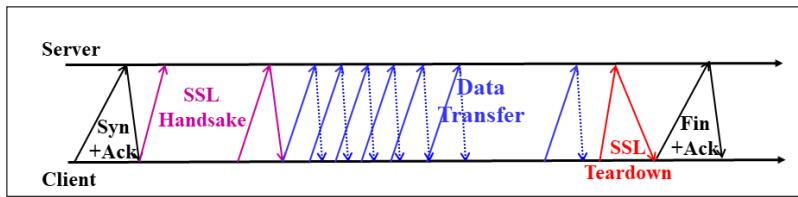


Figure 7.3: Phases of TLS/SSL connection. The black flows (Syn+Ack and later Fin+Ack) are the TCP connection setup and tear-down exchanges, required to ensure reliability. The fuchsia flows represent the TLS/SSL handshake; notice there are often more than the three shown. The blue flows represent the data transfer, protected using TLS/SSL record layer; and the red flows represent the TLS/SSL connection tear-down exchange.

We begin the in-depth discussion of SSL/TLS with the record protocol, rather than with the more ‘interesting’ handshake protocol, for two reasons. First, we think that the record protocol is simpler, and can be understood independently of the handshake protocol. Second, by presenting attacks on vulnerable record protocol options, we motivate some of the mechanisms later introduced by the handshake protocol.

We discussed the basic principles underlying record protocols already earlier, in Chapter 4 and Chapter 5. In this section, we focus on the SSL/TLS record protocol, the most widely-deployed record protocol, with some unique aspects and instructive vulnerabilities.

We focus on the common case, where the SSL/TLS record protocol is applied ‘on-top’ of an underlying reliable communication protocol – typically, TCP¹. Hence, without an attack, messages sent are received reliably, without losses, duplications or re-ordering; any deviation must indicate an attack and justifies closing the connection.

Both TCP and the SSL/TLS record protocol treat the data from the application as one long stream of bytes, regardless of the sequence of (usually multiple) calls in which the protocol receives the data from the application. Namely, the application in the receiver, should parse the stream of bytes which the record protocol outputs, into the different application-level units (typically called messages).

The record protocol involves authentication, encryption and few other functions applied to the data. We already discussed such combinations in Section 4.7; in this section, we focus on the TLS-specific aspects. In subsection 7.2.1, we discuss the TLS *Authenticate-then-Encrypt (AtE)* record protocol, used in SSL and in versions of TLS until TLS 1.2. TLS 1.2 also supports the AtE record protocol, but also supports the alternative *AEAD* record protocol, which we discuss in subsection 7.2.7 (see also discussion of AEAD schemes in subsection 4.7.1). TLS 1.3 supports *only* the AEAD record protocol.

¹We do not cover DTLS [273], a variant of TLS, designed to work over the UDP protocol, i.e., over an unreliable datagram service.

In subsection 7.2.3-7.2.6, we present attacks exploiting vulnerabilities in the AtE record protocol. This motivates the adoption of the AEAD record protocol (in TLS 1.3 and, optionally, in TLS 1.2).

Cipher suites. Different versions and implementations of SSL/TLS may support different cryptographic algorithms. The list of cryptographic algorithms in used by both record protocol and handshake protocol, at a specific connection, is called the *cipher suite*. For the AtE record protocol, this defines an encryption algorithm, a MAC algorithm, and (optionally) a compression algorithm. For the AEAD record protocol, the record protocol has less options: it is defined by a single AEAD algorithm.

7.2.1 The Authenticate-then-Encrypt (AtE) Record Protocol

We begin our discussion of the SSL/TLS record protocol, by focusing on the Authenticate-then-Encrypt (AtE) design, used by SSLv3 and TLS, until until version 1.2 (which allows AEAD as an alternative) and 1.3 (which only allows AEAD).

Figure 7.4 illustrates the sequence of processing-steps applied by the sender, running the SSL/TLS AtE record protocol. These steps are applied to the input that the sender receives - from the application, or from the TLS alert or handshake protocols. Let us discuss each of the steps in the order they are applied by the sender of the data:

Fragment: break the TCP stream into *fragments*; namely, a single (long) ‘message’, sent in one ‘send’ event by the application, may be parsed by the record protocol into multiple fragments, as shown in the top lines of Figure 7.4. Note that the record protocol may also aggregate multiple (short) ‘messages’, sent in consecutive ‘send’ event, into one fragment (this is less common and not shown in the figure). Each fragment consists of up to 16KB. One motivation for fragmenting is to allow *pipeline* operation, reducing the latency. For example, the sender may process the first fragment, then send it while, in parallel, processing the second fragment. Another motivation for fragmenting is to allow recipients to allocate a fixed-sized buffer for incoming fragments, and avoid the risk of buffer overflow bugs and attacks. TLS recipients should discard an incoming fragment larger than 16KB.

Compress: apply lossless compression to each fragment. Compression may reduce the processing overhead and the communication. As discussed in Section 4.7, ciphertext cannot be compressed. Therefore, compression should be done before encryption - or not at all. Note that the *length* of compressed data depends on the amount of redundancy in the plaintext, and encryption usually does not hide the *length* of the (compressed) plaintext; hence, there is a risk of exposure of the (approximate) amount of redundancy in the plaintext, when applying compress-then-encrypt.

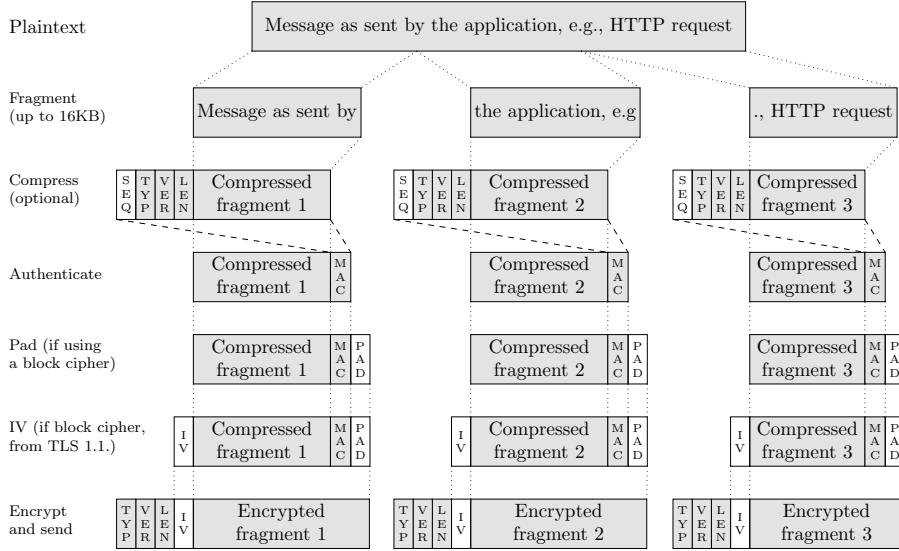


Figure 7.4: The Authenticate-then-Encrypt (AtE) design of the SSL/TLS record protocol. Unfilled fields (pad, IV) are only used for block ciphers; the IV field is added only from TLS 1.1. The MAC is computed over the sequence number (SEQ), type (TYP), version (VER), length (LEN) and compressed fragment, as in Equation 7.1. The type, version and length fields are sent, as plaintext, together with the corresponding encrypted fragment. The record protocol used (only) the AtE design until TLS 1.1; TLS 1.2 supports the use of either the AtE design or the AEAD design (subsection 7.2.7).

Indeed, the fact that SSL/TLS applies compression before encryption was exploited in the *CRIME*, *BREACH* and *TIME compression attacks* [23, 232, 277, 293]. These attacks motivated *disabling of TLS compression*, and currently TLS compression is rarely used (and not even supported in TLS 1.3). However, compression attacks may still be possible, in the (common) use of application-level compression; this is exploited in the *BREACH* attack [139]. We discuss these attacks in subsection 7.2.6.

Authenticate: The AtE record protocol authenticates the plaintext by applying a MAC function, before encryption. The input to the MAC function consists of the concatenation of a *Sequence number (SEQ)* field, indicator the sequence number of the record, the *type*, *version* and *length* fields and the *Compressed Fragment* itself; the type, version and length fields are defined as:

Type: one byte indicating the type of data in this record. The most common type is ‘application data’, which is encoded by 0x17; other types are used for the handshake protocol, alert protocol (error indicators) and for a special *Change Cipher Specification (CCS)*

message, indicating change of to new set of cryptographic keys (and, optionally, algorithms).

Version: an identifier of the version of SSL/TLS.

Length: the number of bytes in the (optionally compressed) fragment.

Namely, the MAC of a message sent by the server is calculated by:

$$MAC = MAC_{k_S^{MAC}} \left(\begin{array}{l} SeqNum + Type + Version + \\ + Length + Compressed_Fragment \end{array} \right) \quad (7.1)$$

A similar equation - using k_C^{MAC} instead of k_S^{MAC} - is used for the MAC of messages sent by the client.

MAC keys. The sender computes the MAC uses a shared key; we use k_C^{MAC} to denote the key for traffic from client to server, and k_S^{MAC} to denote the key used for traffic from server to client². The recipient validates that the value in the received MAC field, is the same as the result of the MAC function applied with the corresponding key. Both k_S^{MAC} and k_C^{MAC} are generated by the handshake protocol.

Padding: The input to a block cipher must be exactly one block; however, the length of the output from the authentication, consisting of the compressed fragment and the MAC, would often not be an integral number of blocks. Therefore, when using a block cipher, the SSL/TLS AtE record protocol appends a *padding* string, ensuring that the total length of the input to the encryption is an integral number of blocks, as shown in Figure 7.5. If the length of the authenticated fragment is x bytes, and the block-length is l bytes, then the required number of pad bytes would be $p = l - (x \bmod l)$. SSL restricts the pad to only fill up to one block ($0 < p \leq l$), but TLS allows longer pad, up to 256 bytes, which can be used to hide the exact length of the fragment.

SSL uses X9.17 padding, and TLS uses PKCS#5 padding, to allow removal of the padding after decryption, and to validate the correctness of the pad; see their description in Section 2.9. Basically, in SSLv3, the length is in the last padding byte, and the value of other padding bytes is undefined, while in TLS, *all* padding bytes must contain the number of padding bytes. This seemingly-minor difference is significant, as we show by describing the *Poodle padding attack* in subsection 7.2.3 below.

Prepend IV: Many block-cipher encryption algorithms, e.g., CBC, require an *initialization vector (IV)*, which is usually selected randomly; see Section 2.8. In TLS 1.1 and 1.2, the IV is sent by the record protocol,

²In SSLv2, the same keys are used also for encryption, and hence simply denoted k_C and k_S , and derived as in Eq. (7.20). Also, note that the TLS specifications refers to k_S^{MAC} as the *server's MAC-write-key*, i.e., a key used by the server to compute the MAC for segments being sent ('written'), as well as the *client's MAC-read-key*; and similarly for k_C^{MAC} . We find our notation simpler.

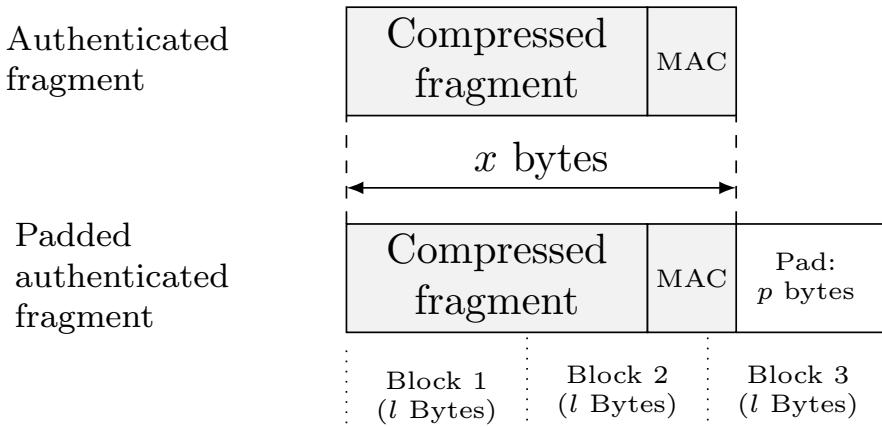


Figure 7.5: Padding in the SSL/TLS AtE record protocol, when using a block cipher with block of l bytes; $l = 8$ for DES and $l = 16$ for AES. Pad contains $p = l - (x \bmod l)$ bytes, where x is the length of the authenticated fragment (compressed fragment plus MAC); e.g., if $l = 16$ and $x = 35$, then $p = 13$, and the padded authenticated fragment fits in three blocks (as in the figure). In TLS, all p pad bytes must contain $p - 1$. In SSL, the last pad byte contains $p - 1$; the other $p - 1$ pad bytes may have any value. Stream-cipher encryption does not require padding.

prepended to the plaintext/ciphertext, as shown in Figure 7.4. SSL and TLS 1.0 try to save the (very limited) resources required to select and send the IV, and do not send the IV. This flawed design was exploited by the devastating BEAST attack [110]; its publication was a main driver for adoption of TLS 1.1. See subsection 7.2.4.

Encrypt: SSL/TLS encrypts the concatenation of the *compressed plaintext fragment*, the *MAC* and, if necessary, the *padding*. Padding is required when using a mode-of-operation of a block cipher; it is not required for stream ciphers.

A basic problem with the record protocols for SSLv3 and TLS versions 1.0 to 1.2, is its use of *Authenticate-then-Encrypt (AtE)* design, rather than the secure *Encrypt-then-Authenticate* design, or the use of a secure *authenticated encryption*. We discussed these alternatives in Section 4.7. The EtA design has been standardized, as a TLS extension (subsection 7.4.3), to improve the security of TLS 1.0 to 1.2 [153], and the *authenticated encryption*, specifically using *AEAD*, is used by the TLS 1.3 record protocol, and optionally in TLS 1.2 (subsection 7.2.7). We discuss attacks exploiting the use of EtA in subsection 7.2.3-7.2.5.

7.2.2 The CPA-Oracle Attack Model

Following Principle 1, we now model the adversary against the SSL/TLS record protocol, which we call the *CPA-Oracle Attack model*. The model is applicable to different applications of SSL/TLS. We focus on the common use of SSL/TLS to secure the communication between web-client and web server. Intuitively, our attack model combines three adversary capabilities:

MitM: the adversary has *Man-in-the-Middle (MitM)* capability when it can intercept packets sent between client and server, modify them and inject forged packets.

Rogue website: the victim client innocently visits a *rogue website* controlled by the adversary, e.g., *666.com*. A rogue website can send automatically-executed hyperlinks to the browser, e.g., a request to embed an image or script from a victim website. Such attacks are referred to as *cross site attacks*, and their goal is usually to abuse the relationship between the client and the victim website. Cross site attacks are of the most common attacks on web security, and there is extensive study of non-cryptographic defenses against them. These defenses rely on prevention of MitM attacks by SSL/TLS (against MitM attackers).

CPA oracle: the adversary can receive an indication if the decrypted plaintext is valid or invalid. In some attacks, the adversary also has the ability to distinguish between a padding error and a MAC-validation error; this ability is key to padding attacks. From SSLv3, the error messages are encrypted, but different attacks, against different versions and implementations, were able to detect the whether a failure is due to invalid padding or to invalid MAC, based on differences in timing. See subsection 7.2.3.

The *CPA-Oracle Attack model* (Figure 7.6) is a simplified model of an attacker with MitM, rogue website and padding oracle capabilities. The goal of the attacker is to expose information about a secret string x , taken from some (known) distribution. Often, x is a *cookie* that the browser automatically includes with every request that it sends to the benign website *B.com*. The rogue website capability allows the attacker to cause the browser to send different requests to the website, always including the cookie as part of the request; the attacker may control much of the request, including the *path* (which comes before the cookie) and the *body/payload* (which comes after the cookie). Furthermore, the attacker often knows the contents of the rest of the request, except the cookie. For simplicity, the CPA-Oracle Attack model lets the attacker choose, with every request, both prefix p_i and suffix s_i , so that the plaintext input to the TLS/SSL record protocol in the i^{th} request is $m_i = p_i \# x \# s_i$. For simplicity, we further assume that the attacker knows the length $|x|$ of the secret cookie.

Error handling. The SSL/TLS record protocol aborts a connection upon receiving any invalid ciphertext c'_i . The attacker can send a new request, i.e., new plaintext prefix p_{i+1} and suffix s_{i+1} . TLS/SSL negotiates a new pseudorandom

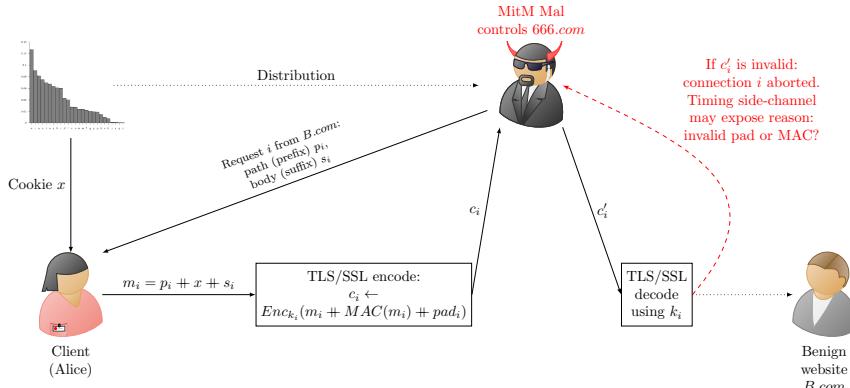


Figure 7.6: The CPA-Oracle Attack model on the TLS/SSL AtE record protocol. The attacker’s goal is to find a string x , such a cookie (or password), sent (encrypted) by a browser to a web-server with every request. The model allows the attacker to control the prefix p_i (the request path), and the suffix s_i (the request body) for every request, such that the plaintext input to SSL/TLS is $p_i \# x \# s_i$. We allow the attacker to intercept and modify the ciphertext, and to receive feedback on the results of SSL/TLS validation of decryption of c'_i . SSL/TLS breaks the connection upon each error; the attacker may cause the browser to send a new request (using same secret x , and possibly changing (p_i, s_i) , but the parties will use in each connection i a separate key k_i . Error messages are encrypted, but some attacks, on some versions/implementations, distinguish between invalid MAC and invalid padding, see subsection 7.2.3.

key k_i for each connection i . The attacker could send multiple requests on the same connection until it is aborted; if requests i and $i+1$ are sent on the same connection (no abort), then $k_i = k_{i+1}$.

The Plaintext-Recovery security goal. Ideally, the use of SSL/TLS *should* prevent an attacker, which can eavesdrop to the communication between Alice and the benign website $B.com$, from learning any information about x . However, the AtE record protocol allows the use of compression before encryption, which makes it impossible to ensure *indistinguishability* between ciphertexts of a highly compressible message and a mostly-random messages; see subsection 4.7.5. Instead, we consider the more modest security goal of *preventing plaintext recovery*. More specifically, the goal is to prevent exposure of a *secret string* x , typically a cookie, which is part of plaintext.

7.2.3 Padding Attacks: Poodle and Lucky13

The use of *Authenticate-then-Encrypt (AtE)* by SSL and versions 1.0, 1.1 and (optionally) 1.2 of TLS, may result in vulnerabilities to *padding oracle* attacks, introduced in Section 2.9. However, Section 2.9 focused only on *encrypted* communication, without authentication or integrity checks on the plaintext. In

contrast, the SSL/TLS AtE record protocol validates MAC on the plaintext, *after* padding is removed. Note that if the pad is formatted correctly but the last byte of the pad contains a wrong value (not the value $p = l - (x \bmod l)$), then an incorrect number of ‘pad bytes’ would be removed; see Figure 7.5. This will result in a failed MAC validation, since the input to the MAC will begin before or after the correct beginning of the MAC field (depending on the value in the last pad byte).

Furthermore, from early on, SSL/TLS designers were aware of the risk of padding attacks, and took steps to prevent them. First, upon detecting an error - invalid pad, invalid MAC or invalid contents - the connection is aborted. Second, all error messages are encrypted, to prevent an attacker from distinguishing between padding errors and MAC errors. Indeed, early padding attacks [312] resulted in only limited information leakage from the encrypted messages.

This was quickly followed by more realistic attacks such as [76], which used a *timing side channel* to distinguish between pad errors and MAC errors, based on differences in the processing time of the two. Effective attacks, based on easy-to-measure timing differences, are known for SSLv3 and TLS 1.0.

These attacks, at least, *should* have motivated the TLS 1.1 and 1.2 designers to change from the (insecure) *AtE* of SSL/TLS, to the secure *EtA* (Encrypt-then-Authenticate) paradigm, or to an *Aead*-based protocol. Abandoning the vulnerable *AtE* follows the conservative design principle (Principle 3); but, unfortunately, this did not happen.

Instead, the TLS 1.1 and 1.2 specifications include additional countermeasures that attempt to prevent distinguishing between pad errors and MAC errors, such as computing the MAC even if the pad is invalid. Such countermeasures are ingenious, but unreliable. Furthermore, surely these steps cannot prevent a possible padding attack that works without distinguishing between MAC and pad failures!

We discuss two of the most important padding oracle attacks on SSL/TLS: the *Lucky13* and *Poodle* padding attacks. *Lucky13* is based on circumventing the countermeasures and distinguishing between MAC and pad failures, while *Poodle* works even if the two errors are indistinguishable - making it much easier to exploit. Both of these attacks are against the use of CBC mode; *Lucky13* addresses TLS, which uses *PKCS#5 padding*, and *Poodle* addresses SSL, which uses X9.23 padding.

Lucky13. We first briefly discuss the *Lucky13 padding attack* [7]. *Lucky13* extends the padding oracle attacks from Section 2.9, specifically the attack against PKCS#5 padding in Exercise 2.24. *Lucky13* uses careful timing side-channel analysis, to *distinguish between pad errors and MAC errors*; i.e., it circumvents the countermeasures against timing side channels in TLS 1.1 and TLS 1.2, which were designed specifically to prevent such distinction. This allows *Lucky13* to then follow a similar approach to the padding oracle attack of [76].

Lucky13 uses carefully-constructed plaintexts, allowing it to attack the secret x one byte at a time, like the method used in Exercise 2.24. The attack cleverly uses the fact that when that in the (rare) cases that the pad is *valid*, the padding bytes are removed. It designs the plaintext carefully to cause a difference in the number of invocations of the compression function used iteratively by the MAC algorithm; see subsection 3.8.2. The details are elegant, and while we will not cover them, the reader is encouraged to look them up in [7].

Poodle. Even more significantly, SSLv3 is also vulnerable to the *Poodle padding attack*, which does *not* require distinction between padding and MAC failures. Furthermore, many implementations of TLS are vulnerable to the *Poodle downgrade attack*, which we discuss in Section 7.5. By downgrading TLS to SSL, the *Poodle downgrade attack* allows the Poodle padding attack to succeed against many implementations of TLS 1.0 to 1.2. This is a major motivation for adoption of TLS 1.3. We therefore describe the Poodle padding attack.

The *Poodle* padding attack was introduced in [238]; it is based on observations made years earlier, in [236]. Specifically, the Poodle padding attack is based on three observations:

1. SSL uses *X9.23 padding*, where the only requirement for valid padding, is that the *last byte* contains a number smaller than the block length l . There is no requirement on the values of the other pad bytes (in contrast to *PKCS#5 padding*, see Section 2.9).
2. The CPA-Oracle Attack model allows the attacker to detect both invalid-padding and invalid-MAC errors.
3. The CPA-Oracle Attack model allows the attacker to prepend chosen prefix p to the secret (cookie) x , to append chosen suffix s to x .

By adding or removing bytes from the prefix p and suffix s , the attacker ensures that the entire plaintext, before padding, contains an integral number of blocks: $|m| \equiv 0 \pmod{l}$. As a result, the pad length will also be a whole block, with the last byte containing $(l - 1)$. For convenience, let us focus on 8-byte blocks, as with DES; then the last plaintext block, denoted m_n , consists of eight bytes containing $0x07$, i.e., $(\forall j : 1 \leq j \leq 8) m_n[j] = 0x07$.

The Poodle attacks takes advantage of a similar observation as we used to solve Exercise 2.24, namely, that a random plaintext block would have valid PKCS#5 padding if, and almost always if, its last byte contains $0x00$. The difference is that SSL uses *X9.23 padding*, and also applied authentication (MAC) before padding and encrypting; in particular, this will mean that now we need the last byte of the plaintext to contain $0x07$, in order to remove an entire padding block and leave the MAC intact. The attack proceeds in three steps.

First Poodle step: collect. In this step, the attacker ‘collects’ 256 SSL record protocol packets, which we denote $r^{0x00}, \dots, r^{0xFF}$. For convenience, all records should consist of exactly n blocks, e.g., $r^i = r_1^i \dots r_n^i$. The records should be correctly encoded and, in particular, decrypt into valid-padded plaintext; we further require that the pad will fill the entire last block of the plaintext. Since SSL , for 8-byte blocks:

$$(\forall i \in \{0x00, \dots, 0xFF\}) D_k(r_n^i)[8] \oplus r_{n-1}^i[8] = 0x07 \quad (7.2)$$

We further require that the value of the last byte of the before-last block of each record would be identical to the index of the record. Namely:

$$r_{n-1}^{0x00}[8] = 0x00, \dots, r_{n-1}^{0xFF}[8] = 0xFF \quad (7.3)$$

This means we need to generate candidate records repeatedly, until we collect all 256 records. This is not a lot of overhead, and should not require much more than 256 requests to the encryption oracle, i.e., hyperlinks sent to the browser to cause it to send a request to the victim server. We will use this r^i collection in the following steps of the attack.

Second Poodle step: find last byte of x . In this step, the attacker finds the last byte of the cookie/secret x ; for example, assume the cookie x is 8 bytes, so we find $x[8]$. The attacker ensures, by adjusting the length of the prefix p , so that $x[8]$ is the last byte of some plaintext block. Since we use CBC, there are two consecutive ciphertext blocks, which we denote by c^- and c^+ , such that:

$$x[8] = c^-[8] \oplus D_k(c^+)[8] \quad (7.4)$$

The attacker now constructs chosen ciphertexts $c^{0x00}, \dots, c^{0xFF}$, by using c^+ to replace the last block of each of the r^i records. Namely:

$$(\forall i \in \{0x00, \dots, 0xFF\}) c^i = r_1^i \# \dots \# r_{n-1}^i \# c^+ \quad (7.5)$$

The attacker invokes the CPA-Oracle Attack oracle on each of these chosen ciphertexts. The padding is valid only if the value of the last decrypted plaintext byte would be between $0x00$ and $0x07$. If the padding is valid, the MAC is checked; and it is valid only if the padding consists of the entire last block, i.e., only if the last byte of the plaintext (and padding) contains $0x07$. Since we use CBC mode, this occurs when:

$$0x07 = c_{n-1}^i[8] \oplus D_k(c_n^i)[8] \quad (7.6)$$

By substituting $c_{n-1}^i = r_{n-1}^i$ and $c_n^i = c^+$ (both from Equation 7.5), we have:

$$0x07 = r_{n-1}^i[8] \oplus D_k(c^+)[8] \quad (7.7)$$

Substitute now $r_{n-1}^i[8] = i$ (Equation 7.3) and $D_k(c^+)[8] = x[8] \oplus c^-[8]$ (Equation 7.4), we have:

$$0x07 = i \oplus (x[8] \oplus c^-[8]) \quad (7.8)$$

Equation 7.8 holds when $i = 0x07 \oplus x[8] \oplus c^-[8]$, and therefore, one (exactly) of the chosen ciphertexts will have valid padding *and* valid MAC. Furthermore, when we identify that ciphertext c^i has valid padding and MAC, we can also find $x[8]$, the last byte of the secret/cookie, as $x[8] = i \oplus 0x07 \oplus c^-[8]$.

Last Poodle step: finally, the attacker repeats the second step, with a minor change, to find the other bytes of the cookie/secret x . This can be done easily for the cookie (or any other secret automatically sent by the browser), as follows. The adversary changes the prefix p to make a different byte of x be the last byte in some plaintext block. Then the attacker can proceed exactly as in step 2 to find this other byte of x .

7.2.4 The BEAST Attack: Exploiting CBC with Predictable-IV

The basic design of the SSL/TLS record layer is defined for an arbitrary encryption algorithm; however, the specifications define a limited number of standard cipher suites. Furthermore, *all* of the EtA cipher suites use CBC mode encryption (subsection 2.8.5). Several of the attacks on the record protocol, e.g., Poodle and other padding oracle attacks, are based on properties of CBC mode. In this section, we briefly discuss the BEAST³ attack, which is also focused on the use of CBC mode, but addresses a very different vulnerability, existing in SSL and TLS 1.0 (but not in later versions). This vulnerability is the use of *predictable Initialization Vector (IV)*.

As presented in subsection 2.8.5, CBC mode requires the use of a random IV for each message. The value of the IV does *not need to be secret*, and in most implementations - including TLS 1.1 and 1.2 - the IV is sent ‘in the clear’, visible to an eavesdropper. However, the SSL design - adopted also by TLS 1.0 - used the handshake protocol to derive the IV for the first fragment in a connection, similarly to the derivation of the shared keys used by the record protocol. This means that both sender and recipient have a shared, pseudorandom IV for the first fragment; therefore, the protocol *does not send the IV* along with the rest of the ciphertext. We conjecture, that the designers felt that this is a better design, probably since it appears that keeping the IV secret may, somehow, be beneficial against some future attack against CBC mode (with a particular block cipher). There is also the minor benefit of reducing the amount of bytes sent.

Using a pseudorandom IV, without sending it, is fine. So, there is no problem with this ‘implicit IV’ method, for the *first* fragments in a connection. However, what about other fragments, sent over the same connection? Also for these (non-first) fragments, SSL and TLS 1.0 do not send an IV. Instead, for any non-first fragment in the connection, the SSL and TLS 1.0 design uses the last ciphertext block of the previous fragment sent over the connection, as the IV

³BEAST stands for Browser Exploit Against SSL/TLS.

for the new fragment. Namely, their IV is the value of the previous ciphertext block sent (from most-recently-sent fragment).

Since the ciphertext is produced by a block cipher, this may seem secure. However, this is another example of the risk of trusting intuition and not carefully validating the security of a design, and not relying on the exact cryptographic properties of the underlying mechanisms. Specifically, the security of CBC relied on the assumption that the IV is *random*, which implies *unpredictable*; once the IV is fixed (as the value of the last-sent ciphertext block), *it is completely predictable and not random any more!*

Let us see the cryptographic details of BEAST, under the CPA-Oracle Attack model. The model allows the attacker to choose each prefix-suffix pair (p_i, s_i) , potentially as a function of the previous ciphertext c_{i-1} . We later briefly discuss the challenges in actually deploying BEAST in practice, since, obviously, the CPA-Oracle Attack model is only a simplification of reality.

BEAST: cryptographic aspects. Suppose we use 8-bytes blocks (as with DES). The attack exposes the secret/cookie x byte by byte; let us first show how we expose the first (most significant) byte, $x[1]$. The attacker provides first chosen-plaintext prefix p^* (and suffix s^*), chosen to ensure that a specific block of the resulting plaintext $m^* = p^* + x + s^*$ would contain the last *seven* bytes of p^* , followed by the *first* byte $x[1]$ of the secret/cookie. It is convenient to have p^* contain fifteen bytes (two blocks minus one byte). The exact contents are not important, but for our discussion, a convenient choice is: $p^* = 123456789ABCDEF$. As a result, we have:

$$m^*[1 : 16] = p^* + x[1] = 123456789ABCDEF \# x[1] \quad (7.9)$$

Let c^* denote the resulting encryption of m^* . Let $c^*(1) = c^*[1 : 8]$, i.e., the first block (eight bytes) of c^* , and $c^*(2) = c^*[9 : 16]$ denote the second block; similarly, $m^*(2) = m^*[9 : 16] = 9ABCDEF \# x[1]$. Since we use CBC encryption, we have:

$$c^*(2) = E_k(c^*(1) \oplus m^*(2)) = E_k(c^*(1) \oplus (9ABCDEF \# x[1])) \quad (7.10)$$

For $i = 0, \dots, 255$, the attacker next obtains IV_i , the IV that would be used to encrypt the next fragment. In the SSL (and TLS 1.0) design, the IV is the last-sent ciphertext block (the end of the previous ciphertext fragment). The attacker asks for encryption of plaintext p'_i , computed as:

$$p'_i = (m^*[9 : 15] \# i) \oplus c^*(1) \oplus IV_i = (9ABCDEF \# i) \oplus c^*(1) \oplus IV_i \quad (7.11)$$

Let c'_i be the CBC encryption of p'_i with the known IV value IV_i ; hence, $c'_i = E_k((9ABCDEF \# i) \oplus c^*(1))$. We try the 256 different values for i until we find one of them, denoted i^* , such that:

$$c^*(2) = c'_{i^*} = E_k((9ABCDEF \# i^*) \oplus c^*(1)) \quad (7.12)$$

Since E_k is a permutation, equal outputs of E_k imply equal inputs to E_k . Hence, from Equation 7.10, we find $x[1]$ by the following deductions:

$$\begin{aligned} c^*(1) \oplus (9ABCDE \# x[1]) &= (9ABCDE \# i^*) \oplus c^*(1) \\ (9ABCDE \# x[1]) &= (9ABCDE \# i^*) \\ x[1] &= i^* \end{aligned} \quad (7.13)$$

Finding other bytes. Let us explain how we find $x[2]$, by utilizing the fact that we already know $x[1]$; the method extends to the other bytes too. The attack simply requires choosing a *fourteen bytes* prefix \hat{p}^* , e.g., $\hat{p}^* = 123456789ABCDE$. As a result, we have:

$$\hat{m}^*[1 : 16] = \hat{p}^* \# x[1 : 2] = 123456789ABCDE \# x[1 : 2] \quad (7.14)$$

Since $x[1]$ is already known, we are in a similar situation to earlier, of finding the last (unknown) byte in the second block, which is now $\hat{m}^*[16] = x[2]$. Let $\hat{c}^* = E_k(\hat{m}^*)$, $\hat{c}^*(1) = \hat{c}^*[1 : 8]$, $\hat{c}^*(2) = \hat{c}^*[9 : 16]$ and $\hat{m}^*(2) = \hat{m}^*[9 : 16]$. Since we use CBC, we have:

$$\hat{c}^*(2) = E_k(\hat{c}^*(1) \oplus \hat{m}^*(2)) = E_k(\hat{c}^*(1) \oplus (9ABCDE \# x[1 : 2])) \quad (7.15)$$

The attacker now performs a similar test to the one before to find the value of $x[2]$. Namely, for $i = 0, \dots, 255$, the attacker obtains IV_i , the IV that would be used to encrypt the next fragment, and then asks for encryption of \hat{p}'_i :

$$\hat{p}'_i = (9ABCDE \# x[1] \# i) \oplus \hat{c}^*(1) \oplus IV_i \quad (7.16)$$

The attacker eavesdrops to obtain \hat{c}'_i , the CBC encryption of \hat{p}'_i with IV IV_i :

$$\hat{c}'_i = E_k(\hat{p}'_i \oplus IV_i) = E_k((9ABCDE \# x[1] \# i) \oplus \hat{c}^*(1)) \quad (7.17)$$

Similarly to before, the attacker finds a value $\hat{i}^* \in \{0, \dots, 255\}$ for which:

$$\hat{c}^*(2) = \hat{c}'_{\hat{i}^*} \quad (7.18)$$

Since E_k is a permutation, equal outputs of E_k imply equal inputs to E_k . Substitute the inputs to E_k from Equation 7.15 and Equation 7.17 and we have:

$$\begin{aligned} \hat{c}^*(1) \oplus (9ABCDE \# x[1 : 2]) &= (9ABCDE \# x[1] \# \hat{i}^*) \oplus \hat{c}^*(1) \\ (9ABCDE \# x[1 : 2]) &= (9ABCDE \# x[1] \# \hat{i}^*) \\ x[2] &= \hat{i}^* \end{aligned} \quad (7.19)$$

In this way, we find $x[2]$ (as \hat{i}^*); other bytes follow similarly. \square

BEAST: system aspects. The cryptographic aspects of BEAST were published already in 2004 by Bard [19], and observed for SSH and IPsec years earlier [32, 278]. Hence, following the *conservative design* principle (Principle 3), the SSL/TLS designers should have avoided the use of observable IV, i.e., the value of the last-sent ciphertext block, preventing this vulnerability - as was finally done from TLS 1.1.

Unfortunately, as in many similar scenarios, the designers ignored these well-known warnings, and used anyway the last-sent ciphertext block as the IV for the next fragment. The reason is that deploying Bard's attack [19] seemed too challenging. In particular, the attack requires the attacker to control the *very first block* of the new fragment; but in the classical use of SSL/TLS to secure HTTP communication between browser and website, every HTTP request begins with a fixed header. So it seems that the attacker cannot control the first block - and the attack is prevented. Bard's paper [19] showed this problem may be overcome, but the solution wasn't very practical.

As a result, the vulnerability persisted in TLS 1.0. It was eventually addressed, in TLS 1.1; however, deployment of TLS 1.1 was limited for years.

One of the main drivers for the adoption of TLS 1.1 was the publication [110], by Duong and Rizzo, of several practical 'implementation tricks' allowing the BEAST attack to deploy the basic cryptanalytical ideas of Bard, addressing the system-challenges that made Bard's attack [19] so challenging. Let us briefly mention the two most important implementation tricks, which are relevant for other attacks too. First, they observed that TLS attacks can target the HTTP *cookie* field, which is often used as a (secret) authenticator sent automatically by the browser whenever sending a request to a specific website. Second, they observed that the use of the *WebSocket* mechanism [122] made the attack usable against TLS as used by browsers, i.e., when running the HTTP protocol over TLS (denoted as HTTPS). Further details are beyond our scope.

7.2.5 Exploiting RC4 Biases to Recover Plaintext

BEAST, Lucky13 and POODLE are all critical attacks against the use of CBC encryption as specified by TLS (1.0 to 1.2). Some countermeasures were proposed to these attacks, with the most popular one being to simply use stream-cipher encryption, using RC4, instead of using a block-cipher such as DES or AES in CBC mode.

However, RC4 was known, for years, to have some vulnerabilities, such as the bias of the second byte observed and exploited in [224]; see subsection 2.5.6. Therefore, following the conservative design principle (Principle 3), it should have been *avoided*, and definitely not used as a 'more secure' alternative to CBC-mode.

This choice to use RC4 as a supposedly more secure alternative to CBC, was another example of underestimating the risk due to what appeared as 'impractical vulnerabilities' - this time, of the RC4 stream cipher (or pseudo-random generator). Indeed, in the common use of SSL/TLS to protect HTTP communication, the beginning of the information sent by the client is normally

the HTTP header - which begins with well-known bytes. Therefore, we cannot exploit the significant bias of the second RC4 byte [224].

However, in [9], it was shown that RC4 has additional biases, which may allow exposure of confidential, sensitive communication such as the cookie. This included two types of biases:

Single byte biases: biases were detected for some output bytes of RC4, not just the second byte - although it has the largest bias. In [9], additional biases were found in the first 256 bytes of RC4. By careful analysis of the results of a large number of encryptions of the same secret x , the attack can recover the secret with significant probability, which depends on the number of encryptions and on the positions of the secret (earlier positions usually had more bias). For example, after 2^{25} encryptions, the first 50 bytes were recovered with probability of more than 50% per byte.

Double-byte biases: RC4 also has biases of *pairs* of bytes in different (adjacent) positions, as reported already in 2000 [124]. By carefully analyzing such pairs, using the known biases, the attacker can recover the plaintext *from arbitrary positions*.

The combination of the attacks on CBC and on RC4, was an important motivation to development and adoption of other cipher suites, and of improved record protocols, mainly, the AEAD record protocols (subsection 7.2.7).

7.2.6 Exploiting Compress-then-Encrypt: The CRIME, TIME and BREACH Attacks

We conclude our discussion of attacks against the SSL/TLS record protocol, by discussing attacks which focus on the *compression* of the plaintext before encryption. As we discussed in subsection 4.7.5, when we apply compression to plaintext and then encrypt the ciphertext, there is a risk of exposure of partial information about the plaintext. In particular, an attacker would be able to distinguish between the compressed-then-encrypted ciphertexts of two equal-length plaintexts p_1, p_2 , if p_1 has high redundancy (compresses to a much shorter string) while p_2 has low redundancy (compression does not reduce its length).

However, the potential exposure due to plaintext compression, may not appear to be a serious threat - and, unfortunately, this threat, presented already in 2002 [188], was mostly ignored for many years. In particular, as shown in Figure 7.4, the AtE SSL/TLS record protocol includes an (optional) compression process; and applications using SSL/TLS often apply their own processing to the messages, before sending them via SSL/TLS.

As you will find in Exercise 7.6, under the CPA-Oracle Attack model, it can be quite easy to deploy this attack and expose part or all of the cookie. Deploying it in practice involves several challenges, such as the following. First, the attacker may not be able to completely control the entire plaintext (except for the secret). Second, when using a block cipher, small changes in the length

of the plaintext may not reflect corresponding changes in the length of the ciphertext. Third, the deployed compression schemes are considerably more complex than Exercise 7.6; and, as the exercise shows, details of the compression scheme are very relevant to the feasibility of the attack.

The CRIME attack [277], presented by Duong and Rizzo, demonstrated how these and other challenges can be overcome, allowing efficient exposure of *cookie* or other secrets repeatedly-sent in HTTP requests or responses. Specifically, the demonstration focused on exposure of cookies by utilizing TLS compression. We will not describe the attack here, since the principle is quite simple (as will be evident from Exercise 7.6), however, the attack necessarily involves details of the compression mechanisms in use, which are beyond our scope. The details are not that complex, and interested readers are encouraged to read about them; a good description is provided in [23].

In their presentation, Duong and Rizzo also discussed potential variants of the CRIME attack that can extract secrets sent in HTTP responses, such as *CSRF tokens*⁴, as well as the potential abuse of other compression mechanisms, such as the widely-deployed HTTP compression.

In spite of this, and in contrary to the conservative design principle (Principle 3), the main response to CRIME was disabling of TLS compression. Ignoring Principle 3 was obviously a CRIME, and the punishment followed, in the form of two effective, convincing attacks that exploited HTTP compression mechanisms: TIME [23] and BREACH [139].

The BREACH attack showed how HTTP compression allows the application of CRIME to expose secrets in HTTP responses, most notably the above-mentioned CSRF tokens. Like CRIME, the disclosure is very effective.

TIME, like BREACH, also assumes only HTTP compression (not SSL/TLS compression). But TIME also extends CRIME in a more profound way: it shows how to expose secrets in HTTP requests, such as cookies, *without requiring eavesdrop capabilities*. Namely, the attacker does not have the entire capabilities in the CPA-Oracle Attack model; it only controls a website visited by the user. This is the *cross-site attack model*, which is the most commonly deployed model in studies of non-cryptographic web security. Luckily, TIME seems significantly harder to deploy, namely, it may require an extensive amount of queries and time.

Preventing Compress-then-Encrypt Exposure. There are several possible countermeasures to the Compress-then-Encrypt exposure. We mention three of them.

First, the most certain way to avoid exposure of confidentiality due to the use of Compress-then-Encrypt, is simple: to avoid compression. While the use of compression for *data* can be critical for performance, it may be possible to avoid compression of the sensitive information. For example, cookies are sent

⁴A CSRF token is a pseudorandom identifier sent by a website to a browser, allowing the browser to submit operations on the user's account. CSRF tokens are the common defense against the *Cross Site Request Forgery (CSRF)* attack.

in the HTTP headers, which are usually much shorter than the payload; it may be acceptable to compress only the payload. Or, avoid HTTP compression completely, in spite of the performance hit!

A second possible countermeasure is to perform special encoding to sensitive data, that will prevent compress-then-encrypt exposure of (only) that data - while allowing compression of the rest of the data. For example, we can apply a ‘randomizing transform’ R to sensitive data s , such as cookies and CSRF tokens, before applying compression. One simple randomizing transform would XOR the sensitive data s , with a random or pseudorandom string r , which is appended to the data separately. A standard transform may even be embedded into TLS, requiring the application only to mark the sensitive data s . This could be a nice programming project for the interested reader, and may be a useful extension to TLS.

One unavoidable challenge, however, of this approach, if the need to *identify* the sensitive data s . Some kinds of sensitive data may be amenable to automated identification (e.g., cookies), but other types may require the programmer to annotate the data. This is a serious disadvantage, as the countermeasure is prone to be done incorrectly or to not to be done at all.

The third and final countermeasure we mention is to add *random padding* to the compressed data, hiding the exact amount compressed. This countermeasure is very intuitive, but could often fail, e.g., by averaging-out the randomness using multiple measurements.

7.2.7 The TLS AEAD-based record protocol (TLS 1.3)

Most of the vulnerabilities identified for the SSL/TLS AtE record protocol, can be avoided by adopting one of the two other designs discussed in Section 4.7: a *Encrypt-then-Authenticate (EtA)* design, which uses first an encryption scheme and then an authentication scheme, or the authenticated encryption with associated data (*AEAD*) schemes, which combine encryption (for confidentiality) and authentication.

RFC 7366 [153], *Encrypt-then-MAC*, applies the ‘classical’ *Encrypt-then-Authenticate (EtA)* paradigm. The specifications use TLS extensions to signal the use of EtA rather AtE. It could be deployed in versions 1.0 to 1.2 of TLS (for version 1.0, provided that extensions are supported).

TLS 1.3 supports *only* the use of an authenticated encryption with associated data (*AEAD*) scheme, which ensures both confidentiality *and* authenticity; see subsection 4.7.1. AEAD schemes accept two types of data: *plaintext*, which is encrypted and authenticated, and *additional data*, which is only authenticated, not encrypted.

The design of the AEAD record protocol is illustrated in Figure 7.7. Without going into all of the ‘small print fields’ used by the protocol, which we discuss below, this design is simpler than that of the AtE record protocol (Figure 7.4). First, instead of a separate encryption scheme and authentication scheme, we use just the AEAD scheme; we also need only one key to achieve both confidentiality and authenticity. Second, the AEAD function provides all functions of a ‘mode

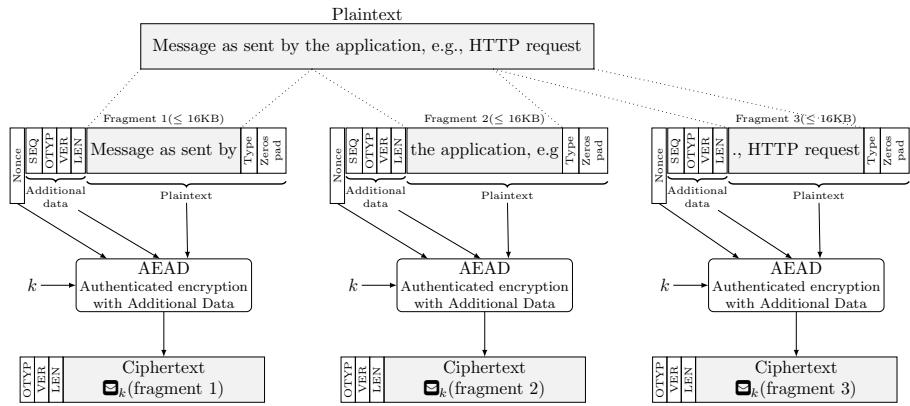


Figure 7.7: The AEAD Record Protocol (used always in TLS 1.3, and optionally in TLS 1.2). The ciphertext produced by the *Authenticated Encryption with Additional Data (AEAD)* function, provides encryption of the plaintext input as well as authentication of both the plaintext and the additional data, using a nonce which should be unique in each invocation. The design contains several ‘small print fields’, e.g., SEQ and OTYP, which are explained in the text, and can be mostly ignored for intuitive understanding.

of operation’ and more; it does not require a separate padding operation or a random initialization vector, although it does require a unique nonce for security [64]. This simplifies its use, and in particular, avoids the need for variants (for stream cipher and for block cipher). Finally, the AEAD record protocol does not support a TLS compression function, to foil Compress-then-Encrypt vulnerabilities as deployed by the CRIME attack (subsection 7.2.6).

The simplicity of the TLS AEAD-based record protocol, follows the KISS principle (Principle 14), and helps avoid implementation vulnerability. This simplicity also facilitated automated verification of the security of the TLS 1.3 record layer specifications and of appropriately-developed implementation [96].

The seven small print fields. Unfortunately, the TLS 1.3 record protocol also comes with seven fields whose meaning and use can be a bit obscure. Readers may mostly ignore these ‘seven small print fields’, but let us explain them anyway, and reserve judgement on whether including all of these small print fields in the design was really necessary or desirable. We present these ‘small print’ fields by their positioning in Figure 7.7, from left to right.

Nonce: The AEAD function requires provision of a unique *nonce* field in every invocation; the same nonce value should be used for the encrypt-and-authenticate operation and to the corresponding decrypt-and-verify operation. The nonce can be seen as a substitute to the random or unique IV required by most modes of operation, or to the counter used by CTR mode. The nonce input is 32 bits, and as the value should only be used

once in the TLS connection, it imposes a (very high) limit on the number of fragments in a connection.

SEQ: The 32-bit sequence number is the *sequence number* of the fragment. By authenticating this data with each fragment, the protocol can detect any tampering with the order of fragments sent in the connection, e.g., reordering or duplicating some fragments. Any detected manipulation results in immediate disconnection, since unintentional errors would be prevented by the lower-layer TCP layer which ensures reliable, ordered connections. Both sender and recipient maintain count of the fragments sent and received, therefore, there is no need to actually send SEQ.

OTYP: This field is referred to in the TLS 1.3 specifications [272] as the *opaque type*; it is included for backward compatibility with earlier versions of TLS, and should always contain the fixed type of 23, indicating application data - even if, in reality, the fragment contains a different type of record, such as of the alert or handshake protocols. The ‘real type’ of the record is included in the plaintext, with the fragment data, and therefore its value is hidden from an eavesdropper.

VER: The *legacy version* field. This field is included for backward compatibility with earlier versions of TLS, and should always contain the value 0x0303. The protocol learns the ‘real’ version of TLS from the handshake protocol, since, in TLS 1.3, the record layer is invoked only after negotiation by the handshake protocol, which authenticate the protocol version.

LEN: The *length* of the ciphertext fragment, i.e., of the *output* of the AEAD. The length is provided as part of the additional data input to the AEAD, therefore, it should be computed *before* applying the OAEP, taking into account the length of the plaintext input and the expansion performed by the OAEP.

Type: This is the ‘real’ type of the fragment. TLS 1.3 and earlier versions, define only four valid types: application data, handshake, alert, and CCS (Change Cipher Specification), each assigned to a non-zero byte value.

Zeros pad: This is an optional field, which can have arbitrary number of bytes whose value is 0x00 (zero). A random or otherwise selected number of zero pad bytes may be used to hide the size of the fragment from an attacker which can observe the TLS ciphertexts. The motivation to hide the length of the fragment is mainly due to the Compress-then-Encrypt vulnerabilities and attacks (CRIME, TIME and BREACH), see subsection 7.2.6.

7.3 The SSLv2 Handshake Protocol

In this section we discuss the *SSLv2 (SSL version 2)* handshake protocol, its features - and some of its main vulnerabilities. SSL version 2 is the earliest

published version of the SSL protocol [168], and its handshake protocol is interesting - beyond its historical importance. One motivation to study it, is that SSLv2 already introduces much of the basic concepts and designs used in later versions - and, since it is a bit simpler, it is a good way for us to introduce these basic TLS/SSL concepts and designs. Another motivation is that the SSLv2 handshake has some serious vulnerabilities; understanding these vulnerabilities is instructive, to develop the ability to detect flaws in cryptographic protocols, and to understand and motivate the design of later versions of the SSL/TLS handshake protocol. Finally, surprisingly, there are still quite a lot of implementations that *support* SSLv2, although they also support (and prefer) later versions, which may make them vulnerable to *downgrade attacks*; see Section 7.5.

The SSLv2 handshake is a non-trivial cryptographic protocol, with support for multiple options and mechanisms - mostly supported also by all later versions (of SSL and TLS), often with extensions and improvements, and removal of insecure mechanisms. We describe the protocol in the following three subsections. In §7.3.1 we present the ‘basic’ handshake, namely, the handshake when there is no existing session (already established shared key), and the protocol uses public-key operations to share a key. In contrast, in §7.3.3 we present the *session resumption* handshake, allowing to re-use the shared key exchanged in a previous handshake between the same client and server, to open a new connection without additional public key operations. In §7.5.1 we discuss how SSLv2 handles cipher suite negotiation, and explain how an attacker may exploit the (insecure) SSLv2 cipher suite negotiation mechanism, to launch the simple yet effective *cipher suite downgrade attack*. Finally, in §7.3.4 we discuss how SSLv2 supports the (optional) client-authentication feature.

Terms and notations. SSLv2, as described in the original publications, e.g., in [168], uses several terms and notations which were modified in later versions. For consistency, we use the terms used by the later versions, also when describing SSLv2; these terms are often also more intuitive. For example, we use the terms *client random* r_C and *server random* r_S , as in SSL3 and TLS. However, the SSLv2 documentation refers to these fields as *challenge* and *connection-ID*, respectively.

7.3.1 SSLv2: the ‘basic’ handshake

In this subsection we discuss the ‘basic’ SSLv2 handshake, illustrated in Fig. 7.8, which is a simplification of the SSLv2 handshake protocol. This simplified version does not include cipher suite negotiation, session resumption and client authentication. We discuss these additional aspects of SSLv2 in the following subsections.

The Hello messages. The SSLv2 handshake begins with the client sending a `CLIENTHELLO` message to the server, specifying the client’s protocol version and

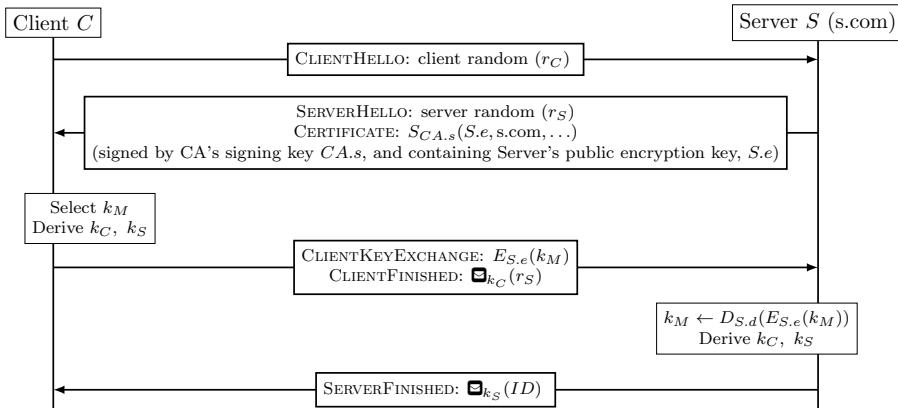


Figure 7.8: ‘Basic’ SSLv2 handshake: new session, no client authentication, and ignoring cipher suite negotiation.. The client and server selects random strings (r_C , r_S respectively) and exchange them in the ClientHello and ServerHello messages. The servers sends its certificate. The client verifies that the certificate is valid and properly signed (by a trusted CA), and that the domain in the certificate matches the desired server domain. If so, the client selects randomly a shared *master key* k_M , encrypts it using RSA encryption with the server’s public key $S.e$, and sends to the server. The client to server key k_C and the server to client key k_S are derived from the master key k_M and the randomizers r_C and r_S , using the MD5 cryptographic hash, as in Equation 7.20. Both Finished messages are protected by the SSL record protocol, which we denote by \Box_{k_C} (client to server communication) or \Box_{k_S} (server to client); the ClientFinished contains the server’s random r_S , preventing replay, and the ServerFinished contains an identifier ID , allowing efficient *session resumption* (Figure 7.9).

the *client random*, r_c , a random bit-string used to randomize the key derivation. The server responds with the SERVERHELLO message, which contains the server random, r_S , and the server’s public-key certificate. The certificate contains the server’s public key, the server’s RSA public encryption key $S.e$, the domain-name of the server, e.g., S.COM, and additional fields. The certificate is signed by an authorized *Certificate Authority (CA)*, using the CA’s private signing key CAs .

The client verifies the certificate. This includes several checks: does the client trust the signing CA? Is the certificate properly signed? Is the domain that the client tries to connect to, the same as the domain in the certificate (or one of the domains in the certificate)? Has the certificate expired or revoked? There are a few more checks; we discuss certificates and their validation in Chapter 8.

The CLIENTHELLO and SERVERHELLO messages retain these basic functions and fields in later versions of SSL and TLS; the most significant changes are in TLS 1.3.

7.3.2 SSLv2 Key-derivation

The SSLv2 handshake protocol establishes a shared *master key*, which we denote k_M . The master key is selected by the client, and sent encrypted, using RSA, to the server, in the Client key exchange message. Namely, the client sends $E_{S,e}(k_M)$.

The public key encryption of k_M is the most computationally-intensive operation by the client; therefore, it is desirable for the protocol to be secure even if the client reuses the same master key k_M and its encryption $E_{S,e}(k_M)$ in multiple connections, assuming that the master key was not exposed. To ensure this, we use the *client random* and *server random* fields from the Hello messages, r_C and r_S , respectively. Namely, we combine r_C and r_S with the master key, and use the combination to derive session-specific cryptographic keys for the session. The derived cryptographic keys are used to protect communication in the connection, and include keys for encryption/decryption as well as for authentication and verification of authentication (MAC).

In SSLv2, the parties derive and use *only two* keys from k_M and the random nonces r_C, r_S : the *client-to-server key* k_C and the *server-to-client key* k_S . The client uses k_C to encrypt messages it sends and compute the MAC to attach to them for authentication, and k_S to decrypt messages it receives from the server and to compute the MAC on the ciphertext, and compare it to the received MAC value, for authentication. These are derived as follows:

$$\begin{aligned} k_C &= MD5(k_M + "1" + r_C + r_S) \\ k_S &= MD5(k_M + "0" + r_C + r_S) \end{aligned} \tag{7.20}$$

Why is this separation between k_C , used to protect messages from client to server, and k_S , used to protect messages from server to client? One reason is that, with a stream cipher, using the same key in both directions would result in insecure re-use of the same key-pad for encryption of two different messages. Another motivation, relevant to block ciphers, is to improve security, following the *key separation principle* (Principle 10). In particular, many websites are public, and send exactly the same information to all users; however, we may want to protect the confidentiality of the contents, e.g., queries, sent by the users. By separating between k_C and k_S , the attacker cannot use the large amount of known plaintext sent from server to client, to cryptanalyze the ciphertext sent from client to server.

On the other hand, the use of a the same secret key for two different cryptographic functions violates the same *key separation* principle. This was fixed in later versions of SSL/TLS, which derive separate keys for encryption and for authentication - with the exception of TLS 1.3, which only needs one key since it uses a single AEAD scheme to ensure both authentication and encryption.

The use of both random numbers r_C and r_S is required, to ensure that a different key is used in different connections. This has three motivations. First, it is necessary to prevent *replay* of messages - from either client or server; see

exercise 7.9. Second, it reduces the total amount of known and chosen plaintext that can be available for cryptanalysis of a key, and the amount of plaintext that is exposed by successful cryptanalysis. Finally, it avoids the possibility that known or chosen plaintext from one connection, e.g., from the public login page of a website, may help attack against data sent in another connection, which may not have the same amount of known or chosen plaintext.

Note, however, that the SSLv2 key derivation does not *fully* follow the key separation principle, since it uses the *same key* for confidentiality (encryption) and for message-authenticity (MAC). This can cause vulnerability even if both encryption and MAC are independently secure; see Exercise 4.3. Indeed, later versions of SSL/TLS use separate keys for the encryption and for MAC, e.g., k_C^E and k_C^{MAC} .

7.3.3 SSLv2: *ID-based Session Resumption*

The main overhead of the SSL/TLS protocol is due to the computationally-intensive public key operations. Often, there are multiple connections between the same (client, server) pair, over a short period of time; in such cases, the server and client may re-use the master key exchanged previously, thereby avoiding additional public key operations. To facilitate re-use of the master key, the server includes an identifier *ID* at the end of the handshake; to re-use the same master key in another connection, the client sends this *ID* with its client-hello, and if the server has the corresponding key, the session is *resumed* efficiently, avoiding additional public key operations. We illustrate this *ID-based session resumption* process in Figure 7.9.

The impact of session-resumption can be quite dramatic. The savings are mostly on the computation (CPU) time; instead of computing public-key encryption of the master key k_M (for client) and decryption (for server) for every TCP connection, we now need only require these operations for the first TCP connection in a session. The ratio of the computation time with and without session resumption is typically on the orders of 100 for typical usage, such as for protecting web communication using the *https* protocol, i.e., running *http* over *SSL/TLS*.

Session resumption in SSLv2 is always based on the use of the *ID*. This *ID-based session resumption* mechanism has a significant drawback: it requires the server to be *stateful*, specifically, to maintain state for each session (for the session-*ID* and the master key). In the typical case where the same web-server is running over multiple machines, this requires that this storage be shared between all of these servers, or to ensure that a client will contact the same machine each time - a difficult requirement that sometimes is infeasible. These drawbacks motivate the adoption of alternative methods for session resumption, most notably, the TLS *session-token resumption mechanism*, that we discuss later.

Note that the session resumption protocol is one reason for requiring the use of client and server random numbers; see the following exercise.

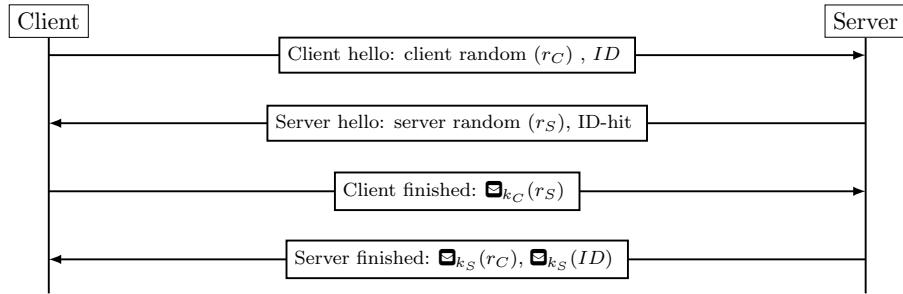


Figure 7.9: SSLv2 handshake, with ID -based session resumption. The client initiates this handshake by including the ID field in the ‘Client hello’ message. The ID field was received from in the ‘Server finish’ message in a previous connection, and cached, with the corresponding session key k_M , by the client. If the server does not have the (ID, k_M) pair in cache, then the handshake completes without resumption, as in Fig.7.8. Otherwise, when (ID, k_M) is in the server’s cache, then the parties can reuse k_M , i.e., ‘resumes the session’, by deriving new shared keys from k_M , using Eq. (7.20). This avoids the public key operations, encryption by client and decryption by server of master key k_M , as well as the overhead of transmitting the certificate $S_{CA,s}(S.e, s.com)$. The server indicates such cache-hit by sending the ID-hit flag in its ‘Server hello’ response, and continuing with the resumption handshake as shown here.

Exercise 7.1. Consider implementations of the SSLv2 protocol, where the (1) client random or (2) server random fields are omitted (or always sent as a fixed string). Show a message sequence diagram for two corresponding attacks, one allowing replay of messages to the client, and one allowing replay of messages to the server.

Hint: perform replay of messages from one connection to a different connection (both using the same master key, i.e., same session). \square

7.3.4 SSLv2: Client Authentication

All versions of SSL and TLS, including SSLv2, support an (optional) *client authentication* mechanism, where the client proves its identity by sending a certificate for a public signature-validation key, and then signs content sent by the server. Client certificates should identify a client approved by the server, and be signed (issued) by a certificate authority (CA) trusted by the server, just like server certificates should be signed by a CA trusted by the client.

In SSLv2, the information signed by the client consists of a signature using the client’s private key, over several fields, including a challenge sent by the server with the request for client authentication, the server’s certificate, and the shared connection keys. Furthermore, this signature should be sent encrypted (using the appropriate connection key). It may not be immediate to see why all

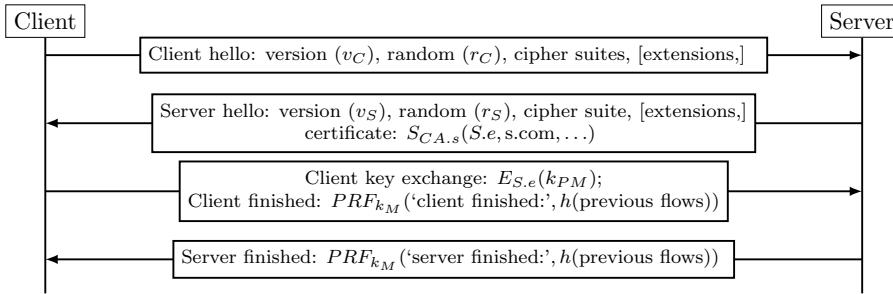


Figure 7.10: The ‘basic’ RSA-based handshake, for SSLv3 and TLS 1.0, 1.1 and 1.2. The master key k_M is computed, as in Eq. (7.21), from the pre-master key k_{PM} , which is sent in the *client key exchange* message (third flow). Notice that the *client key exchange* message simply contains encryption of k_{PM} , i.e.: $E_{S.e}(k_{PM})$. From TLS 1.1, the specifications supports (optional) extensions, as illustrated; see subsection 7.4.3.

of these elements are used, but as we see in Exercise 7.10, removal of some of them may result in a vulnerability.

In the next section we discuss the handshake protocol from SSLv3 to TLSv1.2; the client-authentication design of these versions is simpler and more amenable to security analysis.

7.4 The Handshake Protocol: from SSLv3 to TLSv1.2

We will now discuss the evolution of the SSL/TLS handshake protocol after version 2, from version 3 of SSL [130], to versions 1.0, 1.1 and 1.2 of TLS [98, 100, 313]. These four handshake protocols are quite similar - we will mention the few major differences. Later, in §7.6, we present version 1.3 of TLS, which is the latest - and involves more significant differences, compared to the more incremental changes of these earlier versions.

The handshake protocol, especially before TLS 1.3, has multiple mechanisms and options; we will not cover *all* of them. One important mechanism we will not cover is for handshake *renegotiation*. Renegotiation allows clients and servers to change negotiated aspects of the session. One important use for renegotiation is when a server decides to ask for client authentication, after the session began without client authentication (i.e., with an anonymous client). Renegotiation was a rather complex mechanism, and subject to few attacks, most notably [48, 274].

Figure 7.10 illustrates the ‘basic’ variant of the handshake protocol, of the SSLv3 protocol and the TLS protocol (versions 1.0 to 1.2). Like SSLv2, this ‘basic’ variant uses RSA encryption to send encrypted key from client to server.

In the following subsections, we discuss the main improvements introduced in these later versions of SSL/TLS, including:

Improved key derivation and k_{PM} (§7.4.1): the key derivation process was significantly overhauled between SSLv2 and the later versions, beginning with SSLv3. In particular, the client-key-exchange message of the basic exchange includes the *premaster key* k_{PM} , from which the protocol derives the master key k_M . As before, the master key k_M is used to derive the keys for the record-protocol, used to encrypt and authenticate data on the connection. However, from SSLv3, the protocol correctly separates between the encryption keys and the authentication keys (in contrast to SSLv2).

Improved negotiation and handshake integrity (§7.5): from SSLv3, the finish message authenticates all the data of all previous flows of the handshake; this prevents the SSLv2 downgrade attack (Figure 7.18). TLS, and to lesser degree SSLv3 too, also improve other aspects of the negotiation, in particular, support for extensions, negotiation of the protocol version, and negotiation of additional mechanisms, including key-distribution and compression.

DH key exchange and PFS (§7.4.2): From SSLv3, the SSL/TLS protocols supports DH key exchange, as an alternative or complementary mechanism to the use of RSA-based key exchange (the only method in SSLv2). The main advantage is support for *Perfect forward secrecy (PFS)*.

Session-Ticket Resumption (§7.4.4): an important TLS extension allows *Session-Ticket Resumption*, a new mechanism for session resumption. Session-ticket resumption allows the server to avoid keeping state for each session, which is often an important improvement over the *ID-based session resumption mechanism* supported already in SSLv2 (but which requires servers to maintain state for each session).

Two of these changes - improved key derivation and improved handshake integrity - have impact already on the ‘basic’ handshake. To see this impact, compare Figure 7.10 (for SSLv3 to TLS 1.2) to Figure 7.8 (the corresponding ‘basic’ handshake of SSLv2). We therefore begin our discussion with these two changes.

7.4.1 SSLv3 to TLSv1.2: improved derivation of keys

Deriving master key from premaster key. From SSLv3, the handshake protocol exchanges a *pre-master key* k_{PM} , instead of the master key k_M exchanged in SSLv2. The parties derive the master key k_M from the pre-master key k_{PM} , using a PRF, as in Eq. (7.21):

$$k_M = \text{PRF}_{k_{PM}}(\text{"master secret"} \# r_C \# r_S) \quad (7.21)$$

The main motivation for this additional step is that the value exchanged between the parties may not be a perfectly-uniform secret binary string, as required for a cryptographic key. When exchanging the shared key using the

'basic', RSA-based handshake, this may happen when the client does not have a sufficiently good source of randomization, or if the client simply resends the same encrypted premaster key as computed and used in a previous connection to the same server - not a recommended way to use the protocol, of course, but possibly attractive for some very weak clients.

When exchanging the shared key using the DH protocol, there is a different motivation for using this additional derivation step, from premaster key to master key. Namely, the standard DH groups are all based on the use of a safe prime; as we explain in §6.2.3, this implies that we rely on the Computational DH assumption (CDH), and that the attacker may be able to learn at least one bit of information about the exchanged key. By deriving the master key from the premaster key, we hope to ensure that the entire master key would be pseudorandom.

Deriving connection keys. Another important improvement of the handshake protocols of SSLv3 to TLS1.2, compared to the SSLv2 handshake, is in the derivation of the connection keys which are used for encryption and authentication by the record protocol. This aspect is not apparent from looking at the flows (Fig. 7.10).

Specifically, recall that in SSLv2, we derived from the master-key k_M two keys, k_S for protecting traffic sent by the server S , and k_C for protecting traffic sent by the client C , as in Eq. (7.20). In SSLv3 and TLS, we use k_M to derive, for traffic sent by the client C and server S , *three* keys/values each, for a total of *six* keys/values: two authentication (MAC) keys, (k_C^A, k_S^A) , two encryption keys, (k_C^E, k_S^E) , and two initialization vectors, (IV_C, IV_S) , used for initialization of the 'modes of operation' (Section 2.8). In each pair of keys, we use the one with subscript C for traffic from client to server, and the one with subscript S for traffic from server to client.

To derive these six keys/values, we generate from k_M a long string which is referred to as *key block*, which we then partition into the six keys/values. The exact details of the derivation differ between these different versions of the handshake protocol, and arguably, *none* of the derivations is fully justified by standard cryptographic definitions and reductions. We present the following simplification, leaving the exact details for exercises; the interested reader can find the full details in the corresponding RFC specifications.

Our simplification is defined using a generic pseudorandom function PRF , whose input is an arbitrary-length string, and whose output is a 'sufficiently long' pseudo-random binary string called *key-block*, as follows:

$$\text{key-block} = PRF_{k_M}(\text{'key expansion'} + r_C + r_S) \quad (7.22)$$

The key-block is then partitioned into the *six* keys/values, as illustrated in Table 7.1.

Table 7.1: Derivation of connection keys and IVs, in SSLv3 to TLS1.2

$\text{key-block} = \text{PRF}_{k_M}(\text{'key expansion'} + r_C + r_S)$					
k_C^A	k_S^A	k_C^E	k_S^E	IV_C	IV_S

7.4.2 SSLv3 to TLSv1.2: DH-based key exchange

From SSLv3, the TLS/SSL handshake supports DH key exchange. Three types of DH key exchange are supported, *ephemeral (signed)*, *static (certified)* and *anonymous (unauthenticated)*. The ephemeral method is the most popular TLS handshake method, due to its significant security benefits. The static (certified) method is rarely, if ever, used, and does not offer increased security. However, the two methods are actually quite similar. The anonymous method is rarely used, since it does not provide server authentication; we focus on the other two methods.

In both methods, the parties derive a shared key k_{PM} , referred to as the *pre-master key*, following the DH protocol. Specifically, TLS/SSL use a modular group, with an agreed upon safe prime p and generator g . The parties exchange their ‘public keys’, $g^{S.x}$ (for the server) and $g^{C.y}$ (for the client), where each party uses a randomly-generated private key: $S.x$ for the server and $C.y$ for the client. The parties then derive the pre-master key k_{PM} , again as in ‘plain’ DH key exchange, namely:

$$k_{PM} = g^{C.y \cdot S.x} \pmod{p} \quad (7.23)$$

Recall that when using a modular group, the value exchanged by the DH protocol is not pseudorandom; namely, security may rely only on the computational DH assumption (CDH), as we know that the stronger *DDH* (*Decisional DH*, Definition 6.7) assumption does not hold for such groups. This is one motivation for not using k_{PM} directly as a key to cryptographic functions. Instead, we derive from the pre-master key k_{PM} another key, the *master key* k_M , which should be pseudorandom. See §7.4.1, where we discuss the derivation of the master key and of the keys for specific cryptographic functions, such as PRF, MAC or shared-key encryption.

DH Static (certified) handshake. In static (certified) DH key exchange, the server’s DH public key is signed as part of the signing process of a *public key certificate*. Namely, the signing entity is a *certificate authority* which is trusted by the browser, and the certificate contains the domain name (e.g., s.com) and other parameters such as expiration date: $S_{CA.s}((g, p, g^{S.x} \pmod{p}), s.com, \dots)$. See Figure 7.11.

In practice, the use of a certificate implies that the server’s DH public key, $g^{S.x}$, is fixed for long periods, similarly to the typical use of RSA or other public key methods. Hence, the static (certified) DH key exchange is similar in

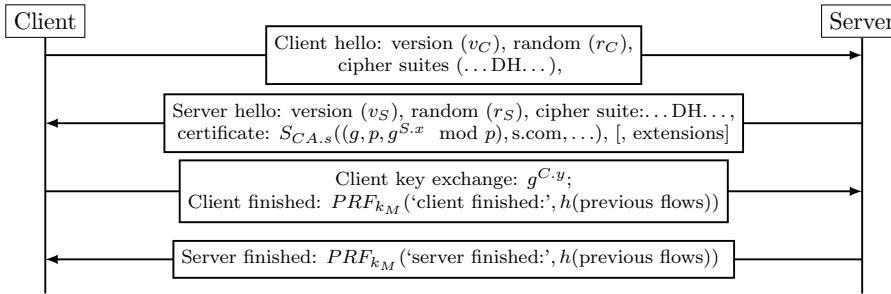


Figure 7.11: SSLv3 to TLSv1.2: the static DH handshake, using static (certified) DH public parameter for the server, $g^{S.x} \bmod p$. Pre-master key k_{PM} is computed as in Eq. (7.23), and master key k_M is computed - from k_{PM} - as in Eq. (7.21).

its properties to the RSA key exchange; the difference is simply that instead of using RSA encryption to exchange the key, and relying on the RSA (and factoring) assumptions, the static (certified) DH key exchange relies on the DH (and discrete-logarithm) assumptions.

DH Ephemeral (DHE) handshake: ensuring Perfect Forward Secrecy (PFS). The *DH Ephemeral (DHE)* key exchange uses a different, randomly-chosen private key for each exchange; for DH, this means that each party selects a new private exponent ($S.x$ for the server, $C.y$ for the client) in each handshake. This is illustrated in Figure 7.12.

The DH exchange is ‘server-authenticated’, i.e., the server *signs* its ‘public’ DH value ($g^S.x \bmod p$), and links it with the particular handshake by including in the signed data also the server and client random numbers (r_S and r_C). To allow the client to validate this signature, the server should send a public key certificate that specifies its public *signature-verification key*, rather than the server’s public decryption key, as used for SSL/TLS handshakes where the client encrypts the pre-master key using the server’s public key. Following the key separation principle, these two public keys should be different, but many servers actually use the same public key (and private key) for both purposes. The lack of key separation was exploited in several attacks against TLS [14, 63, 179, 283].

Once the TLS session terminates, the private exponents are erased - as well as any keys derived from them, including the pre-master key k_{PM} , the master key k_M , the derived key block (Eq. (7.22)) and the keys derived from it (k_S^A , k_S^E , k_C^A , k_C^E). This ensures *perfect forward secrecy (PFS)*, i.e., the i^{th} session between client and server is secure against a powerful MitM attacker, even if the attacker is given, all the keys and other contents of the memory of both client and server before and after the i^{th} session, as long as the keys are given only *after* the i^{th} handshake is completed.

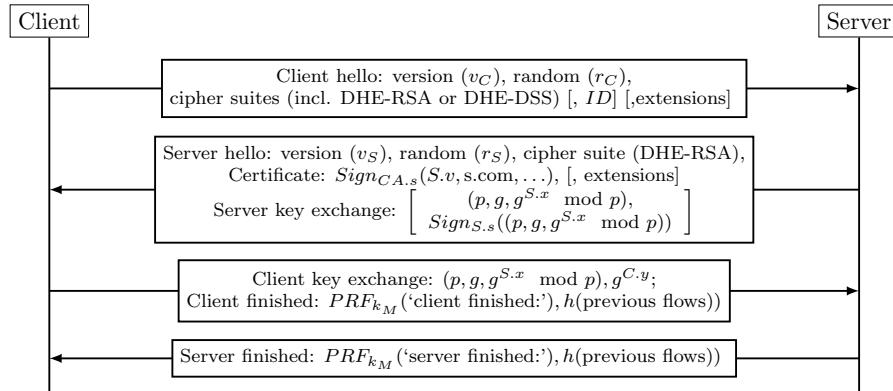


Figure 7.12: SSLv3 to TLSv1.2: the DH Ephemeral (DHE) handshake. The DH exponents $S.x$ and $C.y$ are chosen randomly for this handshake. The server signs its DH public key $g^{S.x} \bmod p$, using RSA (RSA-DHE ciphersuite) or DSS (DSS-DHE ciphersuite). The pre-master key k_{PM} is computed as in Eq. (7.23), and master key k_M is computed as in Eq. (7.21).

Security assumptions of DH key exchange. An obvious, important difference between the RSA key exchange and the DH key exchange methods, is that instead of using RSA encryption to exchange the key, and relying on the RSA (and factoring) assumptions, the static (certified) DH key exchange relies on the computational-DH (and discrete-logarithm) assumptions. Notice, however, that in the typical case where the certificate uses RSA signatures, the security of the handshake still relies *also* on the RSA (and factoring) assumptions. Namely, the DH key exchanges require *both* the computational-DH (and discrete logarithm) assumption, *and* the RSA (and factoring) assumption. In this sense, DH key exchanges, using RSA signatures, requires more assumptions compared to the RSA key exchange. TLS 1.2 (and 1.3) also support ECDSA signatures, which, like the DH key exchange, are based on the discrete logarithm assumption, avoiding the reliance on an addition assumption (RSA and therefore also hardness of factoring), which retaining the advantage of perfect forward secrecy (PFS).

7.4.3 The TLS Extensions mechanism

One of the most important improvements of TLS over SSL, is that TLS support a flexible and secure *extensions mechanism*. This mechanism allows clients to specify additional fields, not defined in the protocol, but supported (and ‘understood’) by some of the servers. Once a server receives an extension that it supports, its behavior may change from the ‘standard protocol’ in arbitrary way (as defined by the extension); however, servers should ignore any unknown extension.

Extensions were ‘unofficially’ supported as early as TLS1.0, where servers

are required to ignore any unknown fields appended beyond the known fields, as defined in [57, 58]. Support for extensions became a (mandatory) part of the TLS specifications from version 1.1. Some standard extensions facilitate important functionality, and some are needed for security; and users may define additional extensions. Let us discuss one important extension here, and another one in the next subsection.

The *Server Name Indication (SNI)* is an example of an important, popular extension. Support of SNI became mandatory from TLS 1.1, and was one of the main factors motivating websites and clients to adopt TLS. Many servers refuse handshakes where the client does not include the SNI extension in Client Hello.

The main use of SNI is to support the common scenario, where the same web server is used to provide web-pages belonging to multiple different domain names, e.g., *a.com* and *b.org*. Each domain name may require a different certificate; the SNI extension allows the client to indicate the desired server domain name early on in the protocol, before the server has to send a certificate to the client - allowing the server to send the desired certificate based on the web-page that is being requested. Before SNI, the common way to a web-server to support multiple web-sites, with different domain names, was by having each site use a dedicated port - an inconvenient and inefficient solution.

However, the SNI extension is valuable even for servers which host only a single domain. This is since SNI allows the server to verify that the domain that the client wants to connect with, is the same as the hosted domain, before the server spends the considerable computational resources to complete the TLS handshake. This avoids spending server resources due to incorrect client requests sent to the server; it also avoids exposing the hosted domain (to an attacker sending Client Hello to find out the domain name).

By requiring the SNI extension, a server can also prevent a potential *Denial of Service (DoS)* attack, exhausting server's resources. Without SNI, a rogue-website could abuse visits by benign users, to attack other sites; this kind of DoS attack is called a *cross-site Denial-of-Service attack*; without SNI, it could be especially effective against TLS 1.3. For details, see [162].

7.4.4 SSLv3 to TLSv1.2: session resumption

Both SSLv3 and TLS, like SSLv2, support the (stateful) *ID*-based session resumption mechanism; however, many TLS servers also support extensions, including the *session-ticket* extension, which is an alternative, 'stateless' method for session resumption. In this subsection we discuss these two methods.

***ID*-based session resumption in SSLv3 and TLS 1.0-1.2** We begin with the (stateful) *ID*-based session resumption mechanism, which did not change much from its implementation in SSLv2.

Figure 7.13 illustrates the handling of *ID*-based session resumption, in the SSLv3 handshake protocol, and in versions 1.0-1.2 of the TLS protocol. In

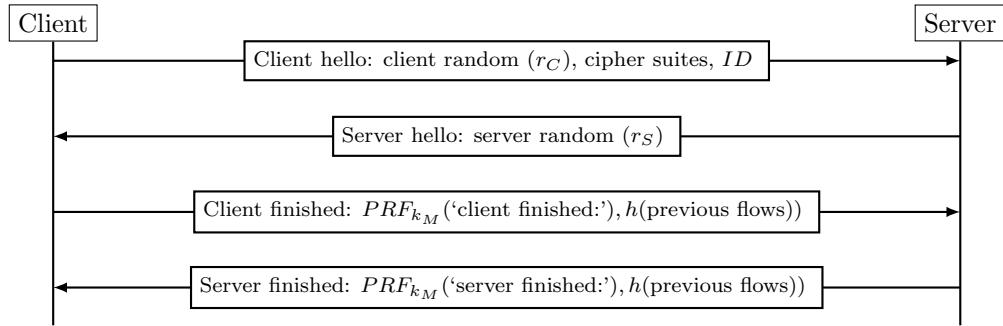


Figure 7.13: SSLv3 to TLS1.2 handshake, with ID -based session resumption.

in the figure, the client-hello message contains the session-ID, denoted simply ID , which was received from server in a previous connection.

Session resumption is possible, when the server still has the corresponding entry (ID, k_M, γ) saved from a previous connection; ID is the *session identifier*, k_M is the session's master key, and γ contains 'related information' such as the cipher suite used in the session.

When the server has the (ID, k_M, γ) entry, it reuses k_M and γ , i.e., 'resumes the session', and derives new shared keys from it (using Eq. (7.20)). This avoids the public key encryption (by client) and decryption (by server) of master key k_M , as well as the transmission of the relevant information, most significantly, the public key certificate.

Note that when either the client or the server, or both, do not have a valid (ID, k_M) pair, then the handshake is essentially the same as for a 'basic' handshake (without resumption), as in Fig. 7.8. The only changes are the inclusion of the ID from client (if it has it), and the inclusion of an ID in the 'server-finish' message, to be (optionally) used for future resumption of additional connections (in the same session).

The session resumption mechanism can have a significant impact on performance; in particular, websites often involve opening of a very large number of TCP connections to the same server, to download different objects. The reduction in CPU time can easily be a ratio of dozens or even hundreds. Therefore, this is a very important mechanism; however, it also has some significant challenges and concerns, as we next discuss.

Session-ID resumption: challenges and concerns. The basic challenge of ID-based session resumption is the need to maintain state, and lookup the state - and key - using the ID . To minimize the storage and lookup time overhead, the cache of saved (ID, k_M) pairs cannot be too large; on the other hand, if the cached is too small, then the resumption mechanism is less effective.

This challenge is made much harder, since web servers are usually replicated - to handle high load and to reduce latency by placing the server closer to the clients, e.g., in a Content Distribution Network (CDN).

Ensuring PFS with ID -based session resumption Another challenge is that the exposure of the master key k_M , exposes the entire communication of every connection to an eavesdropper; namely, the storage of the key may foil the perfect-forward secrecy (PFS) mechanism. To ensure PFS, we must ensure that all copies of the key k_M are discarded, without any copies remaining - a non-trivial challenge.

This challenge is often made even harder due to the way that web-servers implement the (ID, k_M) cache. Specifically, in some popular servers, e.g. Apache, the operator can only define the size of the (ID, k_M) cache. Suppose the goal is to ensure PFS on daily basis, i.e., to change keys daily. Then the cache size must be small enough to ensure that entries will be thrown out after at most a day, yet, if it is too small, there will be many cache misses, i.e., the efficiency-gain of the resumption mechanism will be reduced. Furthermore, even if we use a small cache, a client which continues a session for very long time may never get evicted from the cache, and hence we may not achieve the goal of ensuring PFS on daily basis, if the cache uses the (usual) paradigm of throwing out the least-recently-used element; to ensure entries are thrown after one day at most, it should operate as a queue (first-in-first-out).

Exercise 7.2. Consider a web server which has, on average, one million daily visitors, but the number in some days may be as low as one thousand. What is the required size of the ID -session cache, in terms of number of (ID, k_M) entries, to ensure PFS on daily basis, when entries are removed from the cache only when necessary to make room for new entries? Can you estimate or bound, how many of the connections will be served from cache on a typical day? Assume the ID -session cache operates using a FIFO eviction paradigm.

The Session-Ticket extension and its use for session resumption. The TLS extensions mechanism provides an alternative, stateless session-resumption mechanism. The idea is simple: together with the *finish* message of a successful handshake, the server attaches a *session-ticket extension*. Later, when the client re-connects to the same server, it attaches the previously-received session-ticket extension. See Figure 7.14.

The ticket should allow any of the ‘authorized servers’ (e.g., running the website), to recover the value of the master key k_M of the session with the client - but prevent attackers, eavesdropping on the ticket as sent by the client, from finding k_M . This is achieved by having k_M , and other values sent in the ticket, encrypted using a secret, symmetric *Session Ticket Encryption Key*, which we denote k_{STEK} , known (only) to all authorized servers. Notice that k_{STEK} is not shared with clients or derived by SSL/TLS; the method of generating it and sharing it between the servers is implementation-specific. Since k_{STEK} is a shared key, it is usually simply selected randomly.

Clients cannot encrypt the tickets; hence, they must store both ticket and (plaintext) session’s master key k_M , to allow the client to perform its part of the handshake.

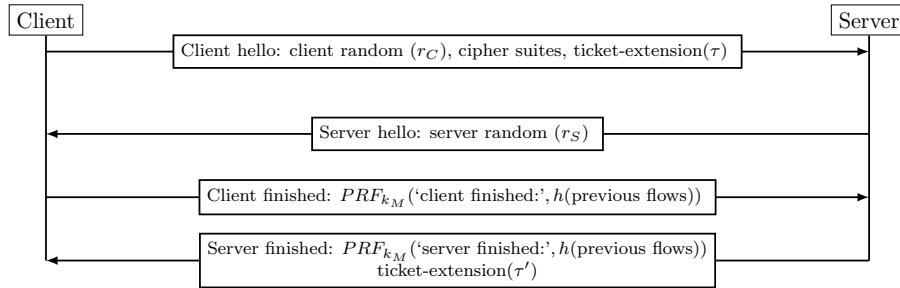


Figure 7.14: Ticket-based session resumption, using a ticket τ which the client sends to the server with *client hello*; the client has received τ from the server in a previous handshake. The server should be able to validate the ticket as one that the server issued previously, and not too long ago, and to retrieve the shared pre-master-key encrypted within the ticket. This is usually done by having the ticket .

The contents of the session ticket are only used by the servers, and are opaque to the clients, i.e., not ‘understood’ or used by the clients; hence, different implementations may use different tickets. RFC5077 [286] recommends a structure which uses Encrypt-then-Authenticate, where the encrypted contents include the protocol version, cipher suite, compression method, master secret key, client identity and a timestamp.

The *timestamp* allows the server to limit the validity period of a ticket (and the keys contained within); if the server receives a ticket which already expired, or is invalid for any other reason, it simply ignores it and proceeds with the ‘regular’ handshake, establishing a new pre-master key (and potentially sending a new ticket to the client).

The limited validity period for the ticket is important, to limit the risk from exposure of the keys in a particular ticket - via cryptanalysis or in other ways, such as abuse of a vulnerability of the browser. Limiting the validity of tickets is clearly also necessary to ensure *Perfect Forward Secrecy (PFS)*; however, this is not *sufficient*. Specifically, to ensure PFS, we should also limit the ability of an attacker to decipher messages from past recorded ciphertexts of long-terminated connections, using an exposed *Session Ticket Encryption Key (STEK)* k_{STEK} . Let us discuss this challenge.

PFS with Session Ticket Encryption Keys. To preserve PFS, e.g., on daily basis, we need to make sure that each Session Ticket Encryption Key k_{STEK} is kept for only the allowed duration - e.g., up to 24 hours ('daily'). In principle, this is easy; we can maintain this key only in memory, and never write it to disk or other non-volatile storage, making it easier to ensure it is not kept beyond the desired period (e.g., daily). This rule may require us to maintain several ticket-keys concurrently, e.g., generate a new key once an hour, allowing it to ‘live’ for up to 24 hours.

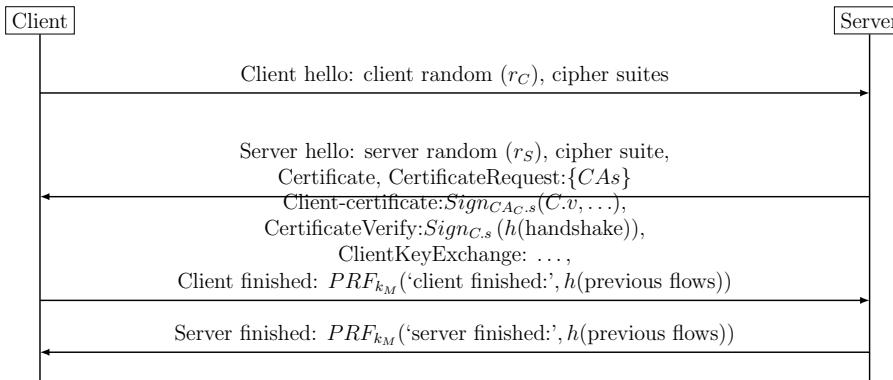


Figure 7.15: Client authentication in SSLv3 to TLS1.2.

In the typical case of replicated servers, the ticket keys k_{STEK} should be distributed securely to all replicates. Changing the key becomes even more important, with it being used in so many machines.

Unfortunately, like for *ID*-based resumption, many popular web-servers implement ticket-based resumption in ways which are problematic for perfect forward secrecy (PFS). These web-server implementations do not provide a mechanism to limit the lifetime of the ticket key, except by restarting the server (to force the server to choose a new ticket key). For some administrators and scenarios, this lack of support for PFS may be a consideration for choosing a server, or for using session-IDs and disabling session-tickets.

7.4.5 SSLv3 to TLSv1.2: Client authentication

The SSL and TLS protocols support, already from SSLv2, a mechanism for authenticating client, as an optional service of the handshake. In this subsection we describe how this optional *client authentication* mechanism works, in SSLv3 and in TLS 1.0 to 1.2.

The SSL/TLS client authentication mechanism is illustrated in Figure 7.15. The mechanism consists of three additions to the ‘basic’ handshake. First, the server signals the need for client authentication, by including the *certificate request* field together with the server-hello message. The certificate-request field identifies the certificate-authorities (issuers) which are accepted by this server; namely, client authentication is possible only if the client has a certificate from one of these entities.

Next, the client attaches, to its *client key exchange* message, two fields. The first is the certificate itself; the second, called *certificate verify*, is a *digital signature* over the handshake messages. The ability to produce this signature, serves as proof of the identity of the client.

This client authentication mechanism is quite simple and efficient; however, it is not widely deployed. In reality, TLS/SSL is typically deployed using only the public key (and certificate) of the server, i.e., only allowing the client to

authenticate the server, but without client authentication. The reason for that is that SSL/TLS client authentication requires clients to use a private key, and to obtain a certificate on the corresponding public key; furthermore, that certificate must be signed by an authority trusted by the server.

This raises two serious challenges. First, clients often use multiple devices, and this requires them to have access to their private keys on these multiple devices, which raises both usability and security concerns. Second, clients must obtain a certificate - and from an authority trusted by the server. As a result, most websites prefer to avoid the use of SSL/TLS client authentication; when user authentication is required, they rely on sending secret credentials such as passwords or cookies, over the SSL/TLS secure connection.

Note also that the client authentication mechanism requires the client to send their certificate ‘in the clear’. This may be a privacy concern, since the certificate may allow identification of the client.

7.5 Negotiations and Downgrade Attacks (SSL to TLS 1.2)

The evolution of SSL/TLS, at least until TLS 1.3, saw an increasingly complex set of different options and choices: different usage modes, different protocol versions, different cipher suites (from SSLv2) and different extensions (from TLS 1.1). To allow this flexibility, the specifications and implementations use different *negotiation* mechanisms. The basic goal of negotiation is for client and server to agree on the same options/choices.

However, there is also a more challenging *security goal*: to prevent *downgrade attacks*, where an attacker causes the parties to use a vulnerable option/choice, although both parties are able, and prefer, to use a secure option/choice. As we have seen for GSM (subsection 5.6.3), downgrade attacks can be simple to understand and deploy, yet efficiently break down security - *and* they could persist for years, as *old versions die slow*. The situation with SSL/TLS is quite similar.

In this section, we discuss the pre-TLS-1.3 negotiation mechanisms, and several effective, and quite simple, downgrade attacks. We exclude discussion of TLS 1.3, since its negotiation mechanisms are quite different, mostly due to insights from these downgrade attacks. Let us begin with SSLv2, which is completely vulnerable to such attacks.

7.5.1 SSLv2 cipher suite negotiation and downgrade attack

In §5.6.2 we presented the *crypto-agility principle* (Principle 11), i.e., allowing flexibility, replacement and upgrade of the cryptographic mechanisms. We also discussed how the GSM support for crypto-agility is vulnerable to *downgrade attack*. How about SSLv2?

Figure 7.16 illustrates how SSLv2 also supports crypto-agility, i.e., the SSLv2 *cipher suite negotiation* mechanism. Figure 7.17 gives an example of

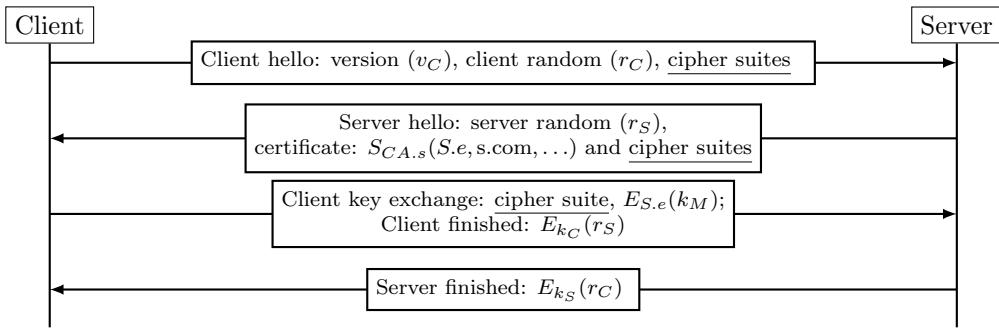


Figure 7.16: SSLv2 handshake, with details of cipher suite negotiation (underlined). The Client hello message indicates the options supported by the client; the Server hello message contains the subset of these, which are also supported by the server. The client chooses one of these, and indicates the choice in the Client key exchange message. Note: the negotiation was modified in later versions.

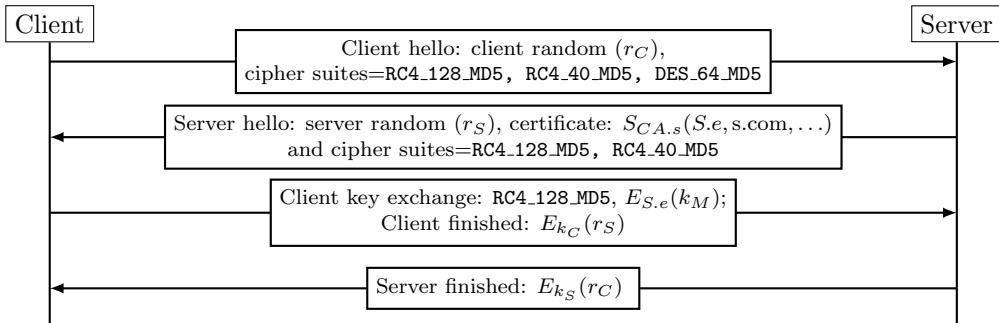


Figure 7.17: Example of SSLv2 cipher suite negotiation. In this example, the client offers three cipher suites, and the server supports two of these. The negotiation was changed from SSLv3.

the negotiation process, when the client supports three cipher suites, all using the MD5 hashing algorithm, but with three different ciphers and key lengths (128-bit keys or 40-bit keys for RC4, and 56-bit keys with DES), and the server supports two ciphers (128-bit RC4 and 40-bit RC4). The 40-bit keys are obviously insecure, but until about 2000, these were the only keys allowed for products sold or distributed outside of the USA, due to USA export controls.

In SSLv2, the finish messages only confirm that the parties share the same server and client keys (K_S and K_C , respectively), but not the integrity of the rest of the hello messages - in particular, there is no authentication of the cipher suites sent by server and client. This allows simple downgrade attacks, removing ‘strong’ ciphers from the list of ciphers supported by client and/or server. Figure 7.18 illustrates how a Man-in-the-Middle (MitM) attacker may perform this downgrade attack on SSLv2; in the example illustrated, the

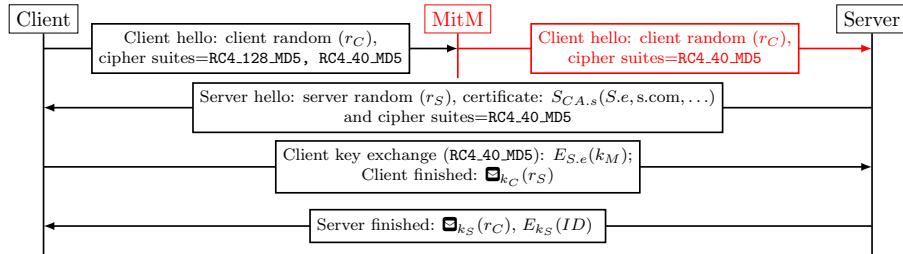


Figure 7.18: Cipher suite downgrade attack on SSLv2. Server and client end up using master key k_M with only 40 secret bits, which the attacker can find by exhaustive search. Attacker does not need to find key during handshake; parties use the 40-bit key for entire connection, attacker may even just record ciphertexts and decrypt later. Note that while SSLv2 is not used anymore, we later discuss sversion downgrade attack that may trick the server and/or client into using SSLv2, exposing them to this (and other) attacks on SSLv2.

attacker removes the ‘regular-version’ 128-bit RC4 encryption from the list of ciphers supported by the client, leaving only the weaker ‘export-version’ 40-bit RC4 encryption. Indeed, the SSLv2 downgrade attack is even simpler and easier to deploy, compared to the GSM downgrade attack (§5.6.3).

7.5.2 Handshake Integrity Against Cipher Suite Downgrade

From SSLv3, the cipher suite negotiation mechanism was improved. The most important change is the adoption of an important, simple defense of the handshake integrity, which prevents the *cipher suite downgrade attack* of Figure 7.18. Two other changes are (1) extended cipher-suites that specify also the key-exchange mechanism (SSLv2 cipher suites specified only the symmetric cryptography), and (2) the server chooses the cipher-suite (in Server hello, i.e., second flow), instead of the client (in third flow).

‘Finished’ handshake integrity foils cipher suite downgrade. Beginning with SSLv3, the handshake protocol includes a simple mechanism for validating the integrity of *all* handshake messages. The client and server authenticate the entire handshake, using the master key derived for that connection. In particular, if the cipher suite downgrade attack of Figure 7.18 would be launched against SSLv3 or TLS, then the server will detect the attack, and disconnect the connection, upon receiving the Client Finished: message.

Specifically, as can be seen, e.g., in Figure 7.10, both client and server send, in their respective *finished* message, a *validation value*, ensuring the integrity of all previously exchanged messages in that handshake. Upon receiving the finished message from the peer (server or client, respectively), the value is checked and if incorrect, the handshake is aborted.

Similarly to the keys-derivation process (§7.4.1), the details slightly differ among the different versions, and we present a slight simplification, consistent

with the one we used in §7.4.1. Both validation values are computed following the ‘hash-then-authenticate’ paradigm, using a hash function h , assumed to be collision resistant, and a pseudorandom function PRF . The validation sent with the Client finished message, which we denote v_C , is computed as:

$$v_C = PRF_{k_M}(\text{'client finished:' } \# h(\text{handshake-messages})) \quad (7.24)$$

Similarly, the validation sent with the Server finished message, which we denote v_S , is computed as:

$$v_S = PRF_{k_M}(\text{'server finished:' } \# h(\text{handshake-messages})) \quad (7.25)$$

Note the similarity to the derivation of the key-block (Equation 7.22).

Security Analysis of Finished Validation. Let us give an intuitive explanation to the security provided by Finished validation mechanism, from SSLv3, focusing on the prevention of cipher suite downgrade attacks.

For simplicity, assume that the client and server both support a secure shared cipher suite X , which they prefer over a vulnerable cipher suite V . Also, focus and on the (typical) handshake, with only server authentication (no client authentication). In this case, the protocol should ensure that if the handshake completes successfully at a client, then it also completed successfully at the server, and the two parties use cipher suite X (or a more preferred secure cipher suite).

Consider an execution in which the client completes the handshake successfully. In such execution, the client must have received a valid Server finished message, i.e., the client’s value of v_S , computed as in Equation 7.25, is the same as the value received from the server, which we denote v'_S .

Basically, the security follows from the use of ‘hash-then-authenticate’. Let us elaborate a bit, mainly to highlight the assumptions involved. The first assumption is that the master key k_M cannot be exposed by an attacker sending a (manipulated or fabricated) Server finished message. Namely, k_M is pseudorandom and known only to the client and to the intended server.

Note that the assumption, and other assumptions identified below, should hold for an *any supported cipher suite*. Also, recall that, from SSLv3, the cipher suite specified both the symmetric-encryption mechanisms and the key exchange mechanisms. Hence, *all* key exchange mechanisms supported by the client must be secure, at least to the extent of preventing exposure of k_M during a successful handshake.

The basic argument for security, is that when the client receives a valid Server finished message, with $v_S = v'_S$, then the server must have previously sent that message, and client and server must have seen identical handshake messages. This argument is based on the assumption that k_M is pseudorandom, and on the use of *hash-then-authenticate* in the computation of both Finished messages.

Hence, our analysis assumes that the PRF function is a secure pseudorandom function, and that the hash function h is a collision-resistant hash function.

The observant reader will notice that the second ‘assumption’ is not really an assumption as much as a *simplification*, since h is a keyless hash and *there are no keyless collision resistant hash functions* (subsection 3.2.2). This is one of multiple reasons that imply that *we cannot hope to provide a full, reduction based proof* for the security of TLS - at least, until version 1.3.

The rest of the security argument follows. A PRF provides message authentication (subsection 4.5.1), when used with a pseudorandom key (in this case, k_M). Hence, when the client receives v'_S , this implies that the server, previously, computed $v'_S = \text{PRF}_{k_M}$ (‘server finished:’ + $h(\text{handshake-messages})$), providing as input (‘server finished:’ + $h(\text{handshake-messages})$), and then sent v_S . Since the client computed the same value ($v_S = v'_S$), it follows that the inputs to the PRF provided by client and server were the same. From the collision-resistance of h , the inputs to the hash were also the same. It follows that both parties have seen identical handshake messages. In particular, they must have seen the same ciphersuite negotiation messages, and hence, cipher suite downgrade attacks are impossible from SSLv3.

Details of the hash function h . The computation of the validation values v_C, v_S involves, in both equations, a cryptographic hash function h , whose definition differs between the different versions. Specifically, in TLS 1.2, the hash function is implemented simply as SHA-256, i.e., $h(m) = \text{SHA-256}(m)$. The TLS 1.0 and 1.1 design is more elaborate, and follows the ‘robust combiner for MAC’ design of §4.6.2; specifically, the hash is computed by concatenating the results of two cryptographic hash functions, MD5 and SHA1, as: $h(m) = \text{MD5}(m) + \text{SHA1}(m)$. SSLv3 also similarly combines MD5 and SHA1, however, in SSLv3 the combination is in the computation of *PRF* itself, and *fails* to ensure a robust combiner; details omitted.

7.5.3 Finished Fails: the Logjam and FREAK cipher suite downgrade attacks

In spite of the above security analysis of Finished message validation, two ciphertext downgrade attacks have circumvented the defense: FREAK and Logjam. Both attacks exploit the fact that, due to USA export regulations until around 2000, SSLv3 and TLS 1.0 support several cipher suites which use short vulnerable keys; these cipher suites were supposed to be used (only) for exported versions of SSL/TLS, i.e., versions distributed outside the USA.

The FREAK⁵ attack [47] exploits the RSA_EXPORT cipher suite, which uses a weak, 512-bit modulus N , which can be factored in few hours by affordable hardware. The attack was effective on popular implementations of TLS, however, it exploited a subtle bug in their implementations, which caused the client to receive the weak (512-bit) key although the client was using the strong (non-export) RSA cipher suite.

Similarly, the Logjam attack [5] exploits an exportable version of the Diffie-Hellman Ephemeral (DHE) key exchange (Figure 7.12), which uses 512-bit

⁵FREAK stands for Factoring RSA Export Keys. With an added ‘A’ for fun, I guess.

groups. While the specific exponents are (correctly) chosen randomly in each run, many implementations use *the same groups*. However, for a given group, it is possible to perform a precomputation step, following which, different discrete logs (with the same modulus) can be computed with acceptable, if still significant, computational costs. If the attack can be carried out in real time, the MitM attacker may be able to find the pre-master key derived by the client, and hence forge a valid Server Finished message, thereby successfully impersonating as the legitimate server. See Figure 7.19.

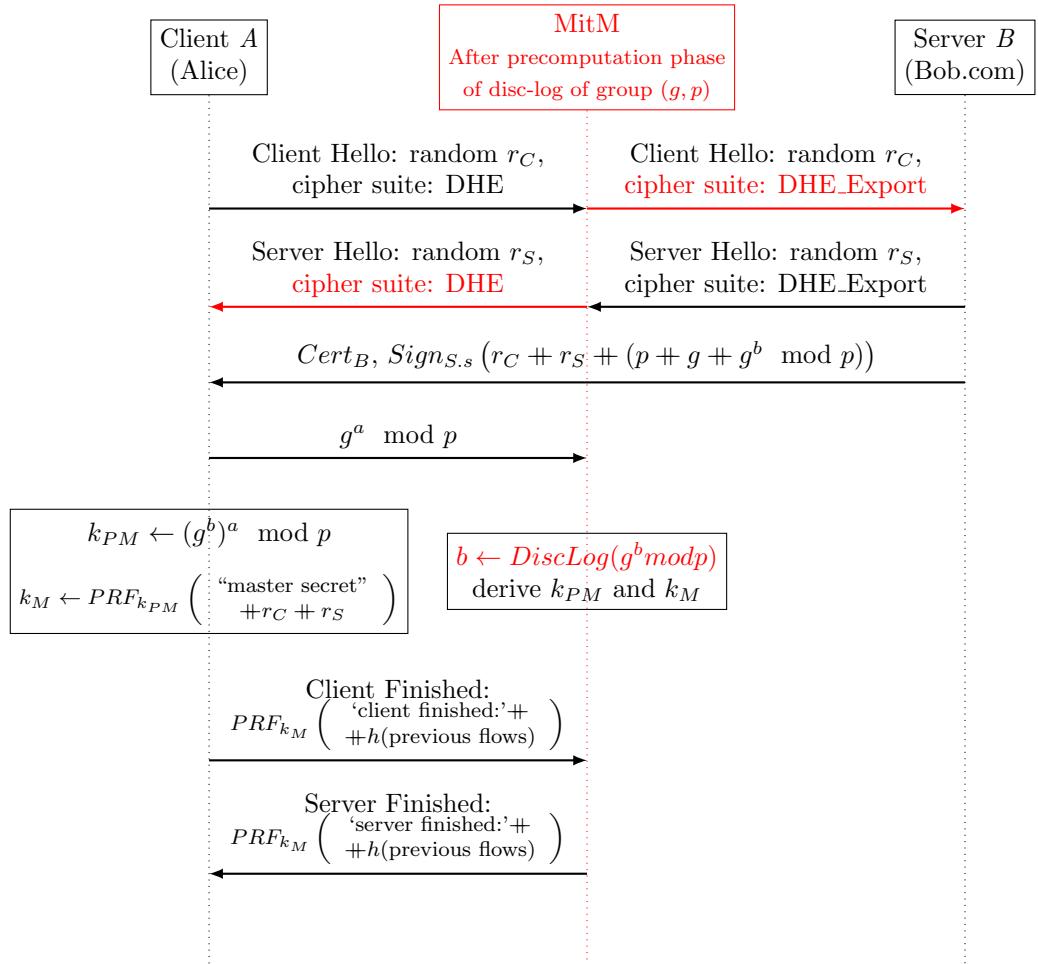


Figure 7.19: The Logjam cipher suite downgrade attack against servers supporting the exportable (weak keys) version of the DH Ephemeral (DHE) key exchange. The attack works for the (surprisingly common) case of known DH group (p, g) , allowing the attacker to precompute the most computationally challenging part of the discrete log computation and use it for the different exponents and handshakes. The MitM attack begins once this precomputation is done. The attacker forces TLS clients to use export-strength Diffie-Hellman Ephemeral (DHE) key exchange (the DHE_Export cipher suite). The attacker modifies Client Hello to request DHE_Export from the server, and modifies Server Hello to appear as if the server uses regular DHE. The client does not detect that the Diffie-Hellman values (p, g, g^b) correspond to export-version of DH, and continues the handshake, sending $g^a \bmod p$ and then Client Finished. The attacker now uses the precomputed values to compute $b \leftarrow DiscLog(g^b \bmod p)$, allowing it to find the master key k_M and complete the handshake.

What can we learn from these attacks? One important lesson is yet-another-example of the risks of due to the use of insufficiently-secure cryptography; *vulnerabilities die hard*, and can often be abused even years after most systems adopted more secure solutions. Another lesson is the risk of relying only on intuitive reasoning, as presented above, for the security of systems. Intuition is useful to identify some attacks and to create initial designs, but cryptographic vulnerabilities can be subtle, and a precise, in-depth security analysis is critical.

7.5.4 Backward compatibility and protocol version negotiation

SSL was an immediate success; it was widely deployed soon after it was released (as SSLv2). Hence, when introducing SSLv3, designers had to seriously consider *backward compatibility*, namely, allowing a client/server running a SSLv3, to interact with a server/client, respectively, running SSLv2. Note that already SSLv2 includes a *version number* in the client hello and server hello messages, although it did not include a protocol version negotiation mechanism. With the definition of (different versions of) TLS, in parallel to further proliferation of web devices, the need for backward compatibility only grew stronger; basically, a new version has almost no chance of adoption without backward compatibility with earlier versions.

TLS protocol version negotiation. The Client Hello messages of TLS 1.0-1.2 and SSLv3 are very similar, and the Client Hello and Server Hello messages include protocol-version identification. This allows a simple and efficient version negotiation mechanism for implementations of TLS 1.0-1.2, allowing downgrade to the least-updated among the versions of the client and the server, from SSLv3 to TLS 1.2.

This *TLS version negotiation mechanism* works as follows. When a TLS server receives a Client Hello indicating an older version, it simply continues the negotiation using this older version of the protocol. Similarly, when a TLS server receives a Client Hello indicating a protocol version newer than the one supported by the server, then the server continues the negotiation using its own (older) version of the protocol.

This version negotiation mechanism allows TLS servers running any version of TLS, or running SSLv3, to interact with clients running any version of TLS or SSLv3; only SSLv2 is incompatible. In all cases, the clients detect the lowest-common version which is to be used from the Server Hello message, and *continue* the protocol using that version. Importantly, the *same handshake continues*, and therefore the integrity mechanism of the Finished messages validates that the downgrade was, indeed, selected by the server.

Vulnerability of TLS version negotiation. The TLS version negotiation mechanism seems, intuitively, secure; however, is it really? As in many cases, the intuition here can be misleading. Specifically, [283] showed how the use of

an optimized version of the Bleichenbacher attack succeeds in decrypting the premaster key, in a small fraction of the attempts. By trying to open a sufficient number of sessions, the attack succeeds against many TLS implementations.

Furthermore, while TLS 1.3 has an improved version-negotiation mechanism (subsection 7.6.1), and does not even rely on RSA encryption, this vulnerability may allow the attack of [283] to downgrade from TLS 1.3 to a lower, vulnerable version.

The SSL version negotiation mechanism and SSLv3 version downgrade attack. The above TLS protocol version negotiation mechanism does not support SSLv2, since SSLv2 uses a different client-hello format. The SSLv3 specification [130] specified a different version negotiation mechanism, which allows SSLv3 clients to interact with SSLv2 servers.

The SSL version negotiation mechanism works by the following simple *downgrade dance*: an SSLv3 client first sends the SSLv3 Client Hello message; but if a valid Server Hello is not received, it sends the SSLv2 Client Hello. This allows interoperability of an SSLv3 client with an SSLv2 server. The other case, of SSLv2 client and SSLv3 server, is handled like the TLS mechanism, i.e., the SSLv3 server detects receipt of SSLv2 Client Hello and continues the handshake using SSLv2.

It is easy to see, that the SSL version negotiation (downgrade dance) mechanism is vulnerable to a downgrade attack. Basically, a MitM attacker drops the SSLv3 Client Hello message, thereby causing the client to connect using SSLv2. See the following exercise. The designers were aware of this risk, and the SSLv3 standard notes that this method for backward compatibility will be ‘phased out with all due haste⁶’.

Exercise 7.3 (The SSLv3 version downgrade attack). *Show message sequence diagram for a MitM version downgrade attack, tricking an SSLv3 server and an SSLv3 client who sends SSLv2-format client-hello (for backward compatibility), into completing the handshake using SSLv2 and using a weak (40-bit) cipher.*

Hint: see this attack (referred to as ‘version rollback attack’) in [316]. \square

Kocher’s ad-hoc defense against downgrade to SSLv2. In practice, interactions between most TLS and SSLv3 servers and clients are protected from the protocol downgrade attack of Exercise 7.3, by an ingenious ad-hoc defense, designed by Paul Kocher. These clients signal their support of SSLv3, by encoding a ‘signal’ of that in the *padding* used in the RSA encryption. For details on this and other issues related to MitM downgrade attacks (also referred to as version rollback), see [316] and appendix E.2 of the TLS 1.0 specifications, RFC2246 [98].

⁶SSLv3 was released in 1996, and never modified to remove this method of backward compatibility. This allowed downgrading even of TLS 1.2 to SSLv2, until March 2011, when TLS versions were redefined [308], removing the ability to downgrade to SSL. That’s haste for IETF, apparently.

7.5.5 The TLS Downgrade Dance and the Poodle Version Downgrade Attack

The TLS backward compatibility mechanism (subsection 7.5.4) requires support by the server: the server should respond *correctly* to Client Hello using newer versions, indicating to the client the need to move to the older version. This is not trivial; while Client Hello messages use basically the same design, there are several additions from SSLv3 onwards, including the important TLS extensions mechanism (from TLS 1.1). While the additions have been carefully designed to be ignored by implementations following correctly and precisely the specifications of the previous versions, many lower-version implementations still fail to process them correctly, or for some other reason, do not continue with the protocol using their (lower-version) protocol. Unfortunately, there are many older version servers which fail, in this way, to support TLS version negotiation.

Many clients try to work with such servers anyway, by the following process, which we call the *TLS downgrade dance*: try first to connect using the latest version, but if receiving no response (or error), try with older versions. The reader will notice that this downgrade dance is basically an extension of the SSL downgrade dance. Unfortunately, it is also vulnerable to a downgrade attack. An attacker can simply block connection attempts (or send back a fake error message), causing the client to use an older, vulnerable version of the protocol; see exercise below.

Exercise 7.4 (The Poodle version downgrade attack). *Consider client that supports the downgrade dance described above. Namely, the client first tries to connect using TLS 1.2; if that fails, it tries to connect using TLS 1.1; and if that also fails, it tries to connect using TLS 1.0. Present a message sequence diagram for a MitM attack, which tricks this client into using TLS 1.0, even when the server it tries to connect with supports TLS 1.2.*

Although the TLS downgrade dance does *not* follow the TLS specifications, it is supported by many TLS clients. This provides a very effective downgrade attack from TLS versions 1.0-1.2 to lower versions, including SSLv3, and sometimes even SSLv2. The ability to perform this downgrade attack was first demonstrated in the Poodle attack [238], and therefore we refer to it as the *Poodle version downgrade attack*. Combined with the *Poodle padding attack* (subsection 7.2.3), this allowed Poodle to be successfully exploited against most web servers implementing TLS 1.0-1.2.

One way to avoid the version downgrade attack is to avoid the downgrade dance. However, evidently, this may cause loss of backward compatibility with many servers (e.g., websites), a price that client developers may not be willing to pay. We next present the *SCSV* signaling mechanism, which allows *secure* use of the TLS downgrade dance.

7.5.6 Securing the TLS downgrade dance: the SCSV cipher suite and beyond

Exercise 7.4 shows a potential vulnerability for a common case, where clients use ‘downgrade dance’ to ensure backward compatibility with servers supporting older (lower) versions of the TLS/SSL protocol. How can we mitigate this risk, while still allowing clients to use the TLS downgrade dance, in order to interact with servers running older versions, that do not support the TLS negotiation mechanism?

A standard solution is the *Signaling Cipher Suite Value (SCSV)* cipher suite, specified in RFC 7507 [237]. Clients that support SCSV, first try to connect to the server using their current TLS version - no change from clients not supporting SCSV. The difference is only when this initial connection fails, and the client decides to try the ‘downgrade dance’, to support connections with servers supporting (only) older versions of TLS/SSL.

In these ‘downgrade dance’ handshakes, the client adds a special ‘cipher suite’ to its list of supported *cipher suites*, sent as part of the *ClientHello* message. The special ‘cipher suite’ is called *TLS_FALLBACK_SCSV*, and is encoded by a specific string. Unlike the original (and main) goal of the *cipher suites* field, the SCSV is not an indication of cryptographic algorithms supported by the client. Instead, the existence of SCSV indicates to the server, that this handshake message is sent as part of a downgrade dance by the client, i.e., that the client supports a *higher* version than the one specified in the current handshake. If the server receives such handshake, and supports a higher version of the protocol itself, this would indicate an error or attack, as this client and server should use the higher version. Therefore, in this case, the server responds with an appropriate indication to the client.

This use of the cipher suites fields for signaling the downgrade dance is a ‘hack’ - it is not the intended, typical use of this field. A ‘cleaner’ alternative would be to achieve similar signaling using a dedicated extension mechanism; later in this section, we describe the TLS extension mechanism, which is used for this purpose in TLS 1.3. We believe that the reason that SCSV was defined using this ‘hack’ (encoding of a non-existent cipher suite) rather than using an appropriate TLS extension, was the desire to support downgrade dance to older versions and implementations of TLS/SSL, that do not support TLS extensions.

7.5.7 The SSL-stripping Attack and the HSTS Defense

An even more extreme downgrade attack is to trick the client into using an insecure connection, i.e., not to use SSL/TLS at all, although the server supports secure (TLS/SSL) connections. The attack is designed against the use of SSL/TLS to protect web communication.

Browsers connect to websites using the protocol specified in the URL, typically, either *HTTP* (unprotected) or *HTTPS* (SSL/TLS protected). The URL is often from a previously-received webpage. If that previously-received webpage is unprotected, then the hyperlink may be modified by a MitM attacker

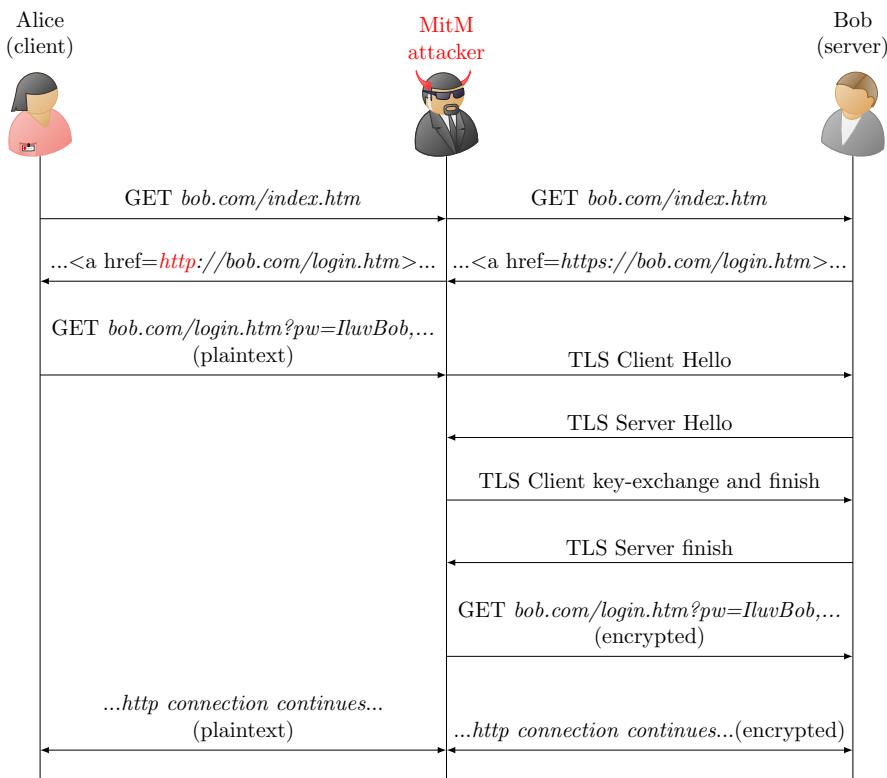


Figure 7.20: The *SSL-stripping* MitM Attack on an SSL/TLS connection. The attack replaces *https* hyperlinks with *http* hyperlinks (second flow). If the users do not notice and enters their password, then the attacker obtains the password. The attacker may continue with the connection, using a secure connection to the server, to obtain more sensitive information and to reduce the likelihood of the user detecting the attack.

- specifically, changing from a URL specifying the protected HTTPS, to a URL specifying the unprotected HTTP. Browsers indicate the protocol used (HTTP or HTTPS) to the user, but many or most users are unlikely to notice a downgrade (from HTTPS to HTTP). This attack is referred to as *SSL-Stripping*, and was first presented by Marlinspike [225].

The SSL-stripping attack is illustrated in Figure 7.20. The attack works by replacing *https* hyperlinks, sent over insecure connections, with the *http* hyperlinks (to the same resource, except the change from *https* to *http*). The user often would not notice that the web-page is delivered over *http*, and send sensitive information such as password.

Of course, the attack only works if the *https* hyperlink is sent over an insecure connection. Therefore, the attack surface is minimized by securing more web-pages, especially search engines.

A good defense against SSL-Stripping and similar attacks is for the browsers to detect or prevent HTTP hyperlinks to a website which always uses (or offers) HTTPS connections. The standard mechanism to ensure that is the *HSTS* (*HTTP Strict Transport Security*) policy, defined in RFC 6797 [169]. The HSTS policy indicates that a particular domain name (of a web server), should be used only via HTTPS (secure) connections, and not via unprotected (HTTP) connections. HSTS is sent as an HTTP header field (*Strict-Transport-Security*), in an HTTP response sent by the web server to the client.

The HSTS policy specifies that the specific domain name, and optionally also subdomains, should always be connected using HTTPS, i.e., a secure connection. Specifically,

1. The browser should only use secure connections to the server; in particular, if the browser receives a request to connect to the server using the (unprotected) HTTP protocol, the browser would, instead, connect to the server using the HTTPS (protected) protocol, i.e., using HTTP over SSL/TLS.
2. The browser should terminate any secure transport connection attempts upon any secure transport errors or warnings, e.g., a warning about the use of invalid certificate.

The HSTS policy is designed to prevent attacks by a MitM attacker, hence, the HSTS policy itself must be protected - and, in particular, the attacker should not be able to ‘drop’ it. For this reason, HSTS policy must be known to the browser *before* it connects to the server. This may be achieved in two ways:

Caching - *max-age*: The HSTS header field has a parameter called *max-age*, which defines a period of time, specified in seconds, during which the browser should ‘remember’ the HSTS directive, i.e., keep it in cache. Any connection within this time, would be protected by HSTS. Namely, suppose that at time T , a browser receives an HTTP response containing the HSTS header with $\text{max-age} = m$ from site *example.com*, over a secure connection (i.e., using TLS). Assume that later, but before time $T+m$, the browser again is directed to request an object from *example.com*; then the browser will open the link to *example.com* only over a secure connection, i.e., using TLS. This motivates the use of a large value for *max-age*; however, notice that if a domain must move back to HTTP for some reason, or there are failures in the secure connection attempts for some reason, e.g., expired certificate, then the site may be unreachable for *max-age* seconds.

Pre-loaded HSTS policy: The browser maintains a list of HSTS domains which are *preloaded*, i.e., do not require a previous visit to the site by this browser. This avoids the risk of a browser accessing an HSTS-using website but without a cached HSTS policy. However, this requires the browser to be preloaded with the HSTS policy - a burden on the site and

on the browser, and some overhead for this communication. An optional parameter of the HSTS header, instructs search engines to add the site to the HSTS preload list of related browsers. This is used by Google to maintain the pre-loaded HSTS list of the Chrome browser.

7.5.8 Three Principles: Secure Extensibility, KISS and Minimize Attack Surface

The different downgrade attack attacks show the importance and the challenge of *secure backward compatible* upgrades. Backward compatibility is essential, to motivate adoption of new versions of protocols. For example, without backward compatibility, a web-server using SSLv2 is unlikely to upgrade to SSLv3, until most clients would upgrade to SSLv3; and clients would not upgrade to SSLv3, until there are many web-servers they can interact with using SSLv3 - a *chicken and egg* problem. We conclude that backward compatibility is essential for successful upgrade. On the other hand, we see the backward compatibility mechanisms may allow vulnerability that can be exploited for downgrade attacks. It is therefore necessary to ensure a *secure extension mechanism allowing backward compatibility*. In particular, every practical security protocol should support a *secure version negotiation* mechanism.

We conclude the principle of *secure extensibility by design*, which requires built-in secure mechanisms for extensions and backward compatibility. This is another important design principle for secure systems and protocols, cryptographic or otherwise.

Principle 13 (Secure extensibility by design). *When designing security systems or protocols, one goal must be to build-in secure mechanisms for extensions, downward compatible versions, and negotiation of options and cryptographic algorithms (cipher suite negotiation).*

However, backward compatibility, like other options and extensions, increases the *attack surface* and make the system *more complex*, both of which imply greater risk of vulnerabilities. Let us discuss these two issues.

First, *flexibility brings complexity, and vulnerabilities lurk in complexity*: the simpler the system, the easier it is to protect. This is the important *KISS principle*⁷.

Principle 14 (The KISS Principle). *Keep It Simple and Secure The simpler a system is, the easier it is to protect; for better security, minimize complexity, options, flexibility and extensibility.*

Now to the *attack surface*, which is based on the intuitively-defined notion of an *attack vector*. An attack vector is an element of the system, which may have a flaw which can be exploited by an attacker to ‘break’ the system; attack vectors may be defined as functions, classes, lines of code, cryptographic functions, API

⁷The KISS principle originates from the US navy, where it meant ‘Keep It Simple, Stupid’. We changed it a bit, to *Keep It Simple and Secure*.

calls, software/hardware modules or protocol variants. The attack surface is a rough measure of the number or extent of the attack vectors in a given system. In a simplified case, let n denote the attack surface, corresponding to n attack vectors, each with probability p_v of a discovered vulnerability. The probability that the overall system will be secure, i.e., not have any discovered vulnerability, is only $(1 - p_v)^n$. Hence, we need to minimize ‘attack surface’, n - as well as to minimize the probability of vulnerability, p_v .

Principle 15 (Minimize the attack surface). *Systems should be designed to minimize their attack surface, i.e., minimize the number of their attack vectors. Roughly, the probability of vulnerability of a system, is exponential in the number of attack vectors.*

DROWN and other Cross-Protocol Attacks (exploiting lack of key separation). Modern protocols, like (newer versions of) TLS, are adopting the extensibility-by-design principle and support secure extensions and backward compatible versions. However, there is yet an important element of extensibility that is often neglected: *version-based key separation* (Principle 10). Namely, suppose the same key - in particular, public-private key pair - is used by both a vulnerable protocol and a secure protocol. Then, it may be possible to expose the key by running the vulnerable protocol, and exploit this to attack the system also when using the secure protocol.

The DROWN attack [14] is an important example of a cross-protocol attack, due to the lack of version-based key separation. A significant number of web-servers were found to support SSLv2, furthermore, *using the same key-pair* as they use for improved-security TLS handshake. This makes these servers vulnerable to an improved Bleichenbacher attack presented in [14], which allows to perform operations using the RSA private key.

7.6 The TLS 1.3 Handshake: Improved Security and Performance

In this section, we (finally) discuss the handshake protocol of TLS 1.3 [272] - the current version of TLS. The TLS 1.3 handshake protocol, like the record protocol Figure 7.7, is a major re-design, providing significant improvements in performance and security compared to the handshake protocol of earlier versions. The main goals of these changes were to *improve security*, to *improve performance*. Another goal was to *simplify* - a goal on its own right, but also reducing the risk of vulnerabilities, following the KISS principle (Principle 14). However, while some significant simplifications were made, we cannot deny that TLS 1.3 introduces its own complexities; arguably, these complexities can be justified by the security benefits they provide to different scenarios. Our description makes some simplifications, in what seems to be more technical details.

Consistency with previous versions was *not* one of the main goals. Indeed, the TLS 1.3 handshake protocol, and especially the key derivation mechanisms,

differ considerably from previous versions, even in the terminology used. For example, the term *premaster key* is not used. This, unfortunately, adds another challenge to the reader familiar with the previous versions.

TLS 1.3 Security Improvements. The security improvements of the TLS 1.3 handshake include:

- TLS 1.3 does not support the (previously widely used) RSA-based key exchange. This avoids many attacks on RSA implementations, mainly variants of *Bleichenbacher's attack*, such as ROBOT [63]. Note, however, that TLS 1.3 still allows the use of RSA signatures for authentication of the handshake flows, including of the public DH values. The signature can be vulnerable to a cross-protocol attack, if (incorrectly) using the same RSA private key for TLS 1.3 CertificateVerify signature, as the key used by insecure protocols such as earlier TLS versions; see subsection 7.6.6.
- TLS 1.3 handshake always uses DH Ephemeral (DHE) key exchange to provide a fresh secret shared key in every exchange, using either finite-fields of elliptic-curves Diffie-Hellman. This ensures *perfect forward secrecy (PFS)*⁸. The use of this single key-exchange mechanism also simplifies the handshake.
- TLS 1.3 disallows the use of other cipher suites with known weaknesses, most notably, these using ‘export-grade’ cryptography. This foils cipher suite downgrade attacks which exploits the use of weak, ‘export-grade’ cryptography, such as the LOGJAM attack [5], exploiting support for 512 bit DH groups, and the FREAK attack [47], exploiting the use of ephemeral 512 bit RSA private keys. See subsection 7.5.3.
- Previous versions of TLS allowed the server to specify an arbitrary Diffie-Hellman group, by sending the modulus p and generator g . This allows different servers to select different groups, which may help to foil discrete-logarithm precomputation attacks such as in the Logjam attack (subsection 7.5.3). However, as in many cases, *flexibility resulted in vulnerability*; many servers used the same groups - and often, using *weak* groups. TLS 1.3 uses *standard* finite groups or elliptic curves; these should correspond to carefully chosen groups and curves, properly defined, e.g., in [137, 207]. This use of specific, studied groups/curves follows the *cryptographic building blocks* principle (Principle 8).
- In the TLS 1.3 Server_Keypath message, the servers sign the *entire* handshake, not just the DHE parameters and the client and server random numbers (r_C and r_S) as in previous versions; compare to Figure 7.12. In particular, when the client receives and validates this signed

⁸However, TLS 1.3 does not ensure *perfect recover security (PRS)*, since it relies on the secrecy of the server’s private key, for signing the exchange; see Exercise 7.16. The reliance on a fixed private key was exploited by the few known attacks against the TLS 1.3 handshake protocol.

`Server.Key_Exchange`, it confirms that the server received correctly the supported-versions, cipher suites and supported-groups information sent in `Client.Hello`. This defends against downgrade attacks (Section 7.5).

- For signatures and hashing, TLS 1.3 forbids the use of the vulnerable RSA PKCS#1 v1.5 and SHA-1 algorithms. Secure alternatives are specified instead: PSS padding for signatures and for hashing, SHA-256, SHA-384 or SHA-512. See PKCS#1 v2.2 (RFC 8017), RFC 5756 and RFC4055 [240, 288, 307].
- TLS 1.3 removed support for handshake *renegotiation*. Renegotiation added complexity, vulnerabilities (e.g., [48, 274]) and attack surface. TLS 1.3 provides an alternative mechanism to support the main use case for renegotiation, which is, invoking client authentication only ‘as-needed’, after the handshake completes.
- TLS 1.3 design prefers cryptographic designs which are amenable to proofs of security, and hence their security seems more well established. This is done *without* requiring an explicit attack exploiting the previous, intuitive or less-established design. The improved, and arguably more complex, key derivation process is a good example (subsection 7.6.5).
- TLS 1.3 specifies the use of pre-shared keys (*PSKs*). This single mechanism replaces multiple mechanisms in previous versions: different PSK-based cipher suites, with and without DH [16, 17, 118] as well as session resumption, both ID-based and *session-ticket* based (subsection 7.4.4). This simplicity helps ensure security (Principle 14).
- TLS 1.3 protects fields and messages as soon as the necessary keys are established. In particular, it protects the extensions sent in the `Server.Hello` and later messages, and it protects the Certificate (sent after `Server.Hello`). One benefit from protecting the certificate is improved privacy for users against an eavesdropper, who could have used the certificate to identify the website used (even if the website cannot be identified from the addressing information, e.g., when using an anonymity-providing proxy). Note that exactly this feature, may cause concern to network administrators who may have relied on the visibility of the certificate to prevent communication with undesired websites.

TLS 1.3 Performance Improvement: reduced latency overhead (less round trips). The *performance improvement* of TLS 1.3 handshake is mostly due to *reduced handshake latency*. Most applications use the *request-response* communication pattern, where the client sends a request to the server who sends back a response. The *handshake latency* is the time since the client application initiated the connection (and transferred the request to the client’s TLS module), and until the server receives the request.

Modern networks use fast transmission rates; hence, the latency is mostly due to propagation and queuing delays, i.e., we can mostly ignore the transmission

time of the packets. This is especially true since the handshake packets, and most requests, are quite short. This means that it doesn't matter much if we send somewhat longer messages or multiple messages consecutively, without waiting for a response. The latency is almost entirely determined by the delay caused by the delay due to *waiting* for a response before sending the next message.

Clearly, request-response interaction requires at least one *round trip*: sending the request to the server and receiving the response. Therefore, the 'base' latency for request-response interaction is one *round trip time (RTT)*; this is required even without security. While network bandwidth has dramatically increased over the years, the round trip times did not reduce that much, and can often reach tenth of a second and more.

TLS 1.3 aims to *minimize the latency overhead*, which basically means, *minimize the number of additional round trips*, required by the handshake *before* the request can be sent.

Previous versions of TLS required *two round trips* (Figs. 7.3 and 7.8); TLS 1.3 requires only *one round trip*. Furthermore, TLS 1.3 supports a *zero round trips* handshake, where *one* client request, containing *some* application data, can be sent as part of the initial flow from the client; see subsection 7.6.4.

Our presentation of the TLS 1.3 Handshake protocol. In the following subsections, we present a simplified overview of the TLS 1.3 Handshake protocol. We cover the most important aspects of the protocol. In subsection 7.6.1, we discuss the TLS 1.3 negotiation and backward compatibility mechanisms. In subsection 7.6.2 we discuss the TLS 1.3 1-RTT ('full') Diffie-Hellman handshake. In subsection 7.6.3 we discuss the Pre-Shared Key (PSK) handshake, used to support both off-band shared keys and session resumption. In subsection 7.6.4 we discuss the *zero-TTL handshake*, which avoids entirely the delay-overhead of earlier versions of TLS and even of the 'full' TLS 1.3 handshake. In subsection 7.6.5, we discuss the key derivation process of the TLS 1.3 handshake protocol. Finally, in subsection 7.6.6 we discuss *cross-protocol attacks*, which are the only known attacks which exploit a vulnerability that exists in the TLS 1.3 specifications.

Overall, we tried to cover the most important aspects of the TLS 1.3 handshake mechanism. However, we had to make some simplifications and omissions. As one important example, see Exercise 7.19, which discusses the risk of *Denial of Service (DoS)* on TLS servers, and two defenses against it, including using the *Cookie* extension. The Cookie extension can also be used to off-load state from the TLS server to the client [272].

7.6.1 TLS 1.3: Negotiation and Backward Compatibility

In Section 7.5, we discussed the cipher suite and version negotiation mechanisms of previous versions of TLS. The redesign of TLS 1.3 includes a significant change in these negotiation mechanisms. However, this redesign was done carefully to ensure backward compatibility with earlier versions of TLS.

Cipher suite negotiation. In previous versions of TLS, as well as SSLv3, the cipher suites defined three separate aspects: (1) the record protocol ciphers, e.g., AES, (2) the key exchange mechanism (RSA, DH or pre-shared key), and (3) the signature algorithm. This resulted in an exponential explosion in the number of cipher suites, with unnecessary complexity (and room for error).

TLS 1.3 separates the cipher suite negotiations into *four* distinct aspects:

Record protocol cipher suite: the AEAD algorithm and the hash function used for key derivation.

DH group and public share: the Diffie-Hellman group (or elliptic curve), and optionally, a *public key share* extension which contains DH public values (key shares).

Signature algorithm: the signature algorithm used for server authentication, i.e., to sign the Server_Finished message.

Pre-Shared Key: identifies pre-shared keys, and specifies whether such keys are to be used to authenticate a Diffie-Hellman key exchange, providing *PFS*, or to be used directly as the shared secret between the parties. See subsection 7.6.3.

TLS 1.3 version negotiation: the Supported_Versions extension. TLS included a version negotiation mechanism from early on; however, as we discussed in subsection 7.6.1, many TLS servers did not implement it correctly, leading clients to adopt the insecure ‘downgrade dance’ and exposing them to the Poodle version downgrade attack.

However, upon early, experimental deployment of TLS 1.3, it was discovered that many servers *still* do not implement this version negotiation mechanism. At first, it appeared that clients should be able to use ‘downgrade dance’ securely, by using the *SCSV* extension (subsection 7.5.6), designed to prevent a MitM attacker from causing unnecessary downgrades. No such luck; it was soon realized that there are also many TLS 1.2 servers that do not support SCSV, which would have allowed a Poodle-like downgrade attack against TLS 1.3.

The TLS 1.3 designers decided that the only secure solution is to *change the version negotiation mechanism*, in a way which is *backward compatible with TLS 1.2 servers*. Specifically:

1. TLS 1.3 uses a Client_Hello message which is compatible with TLS 1.2, *including the version number*. Namely, TLS 1.3 Client_Hello messages include the identifier of TLS 1.2, rather than that of TLS 1.3. TLS servers running version 1.2 or an earlier version, should handle this correctly (if they implement version negotiation correctly). TLS 1.3 servers will manage, since they use the *Supported_Versions* extension, which provides a *new version negotiation mechanism*.
2. TLS 1.3 clients list the versions they support, in order of preference, in the *Supported_Versions* extension. *Supported_Versions* is a new mandatory

extension, which should be supported by all new TLS servers and clients (running version 1.3 or future versions). If this extension is absent, the server should continue the handshake using TLS 1.2. If the extension is present, the server uses the ‘best’ version supported both by the client and by itself, and indicates it in the Supported_Versions extension sent back (with Server_Hello).

Let us pray that this new mechanism will be implemented correctly by TLS 1.3 servers, avoiding a similar predicament upon upgrading to TLS 1.4!

Backward compatible Client_Hello. As we explained, the Client_Hello message of TLS 1.3 must be backwards compatible with TLS 1.2. In particular, the Client_Hello version field will indicate version 1.2, not 1.3; the ‘Supported Versions’ extension will indicate which versions are supported by the client (e.g., version 1.3 and some older versions).

To retain backward compatibility of the Client_Hello with TLS 1.2, several fields contain ‘legacy’ values, used only by legacy (TLS 1.2 and lower) servers receiving the message, and ignored by TLS 1.3 (or newer) servers. Let us discuss each of these ‘legacy’ fields:

Version: As explained above, TLS 1.3 clients indicate here the version of TLS 1.2.

Session_ID: This field is used for any previously-cached Session_ID for that server (subsection 7.4.4).

Compression: This field is used in earlier versions to identify the compression method. In TLS 1.3, should contain the indication for the ‘null’ compression method.

7.6.2 TLS 1.3 Full (1-RTT) DH Handshake

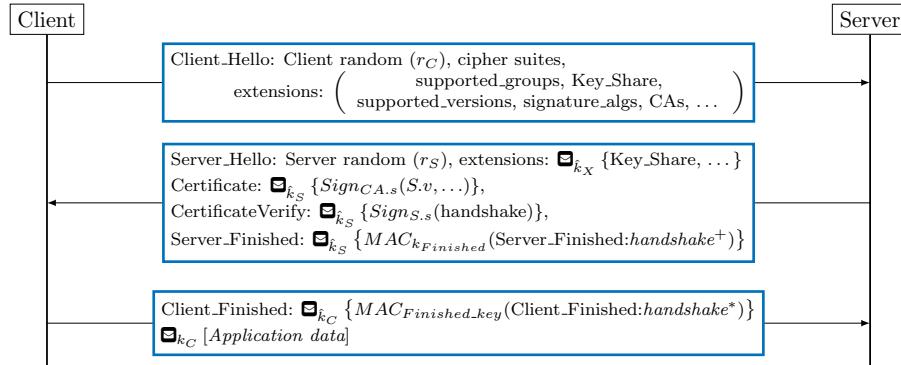


Figure 7.21: TLS 1.3 1-RTT full Diffie-Hellman handshake; see Figure 7.22 to see the version with support for Pre-Shared Key (PSK). The CertificateVerify message contains a signature over the entire *handshake* until it: Client_Hello, Server_Hello and the Certificate. Server_Finished contains a MAC over $handshake^+$, i.e., the entire *handshake*, plus the CertificateVerify message itself. Client_Finished contains MAC over the $handshake^*$, i.e., entire *handshake*, plus CertificateVerify and Server_Finished. We use $\square_{k_X} \{ \dots \}$ to denote AEAP protection using handshake-key of party $X \in \{C, S\}$, and $\square_{k_X} [\dots]$ to denote AEAP protection using application-key of party $X \in \{C, S\}$.

We next present the TLS 1.3 1-RTT Diffie-Hellman Handshake, illustrated in Figure 7.21. This is the typical initial TLS 1.3 handshake, always used by a client and server on their first connection, and optionally used in subsequent connections.

In contrast to the ‘basic handshake’ of the previous sections (and versions), the TLS 1.3 full handshake *always* uses the DH protocol for key exchange; therefore, it *always* ensures PFS.

The handshake ensures server authentication using the server’s signature over the initial handshake messages, sent in a message called **CertificateVerify**. This signature authenticates the server’s DH component ($g_i^{b_i} \bmod p_i$). By including the initial handshake messages in the signed content, the signature also protects against downgrade attacks.

The TLS 1.3 full (1-RTT) handshake allows the client to send the request after a *single* round-trip; that’s why it is called a *1-RTT* handshake. Namely, the server (single) flow contains both the Server_Hello message (with the server’s DH exponent, extensions, certificate and signature), *and* the Server_Finished message, which ensures the integrity of the exchange.

In order to allow the server to send the *finished*: message in its (single) flow, it has to receive all the necessary keying information from the client earlier - i.e., already in the Client_Hello message. Since TLS 1.3 always uses DH key exchange, this means that the client must send sufficient information for the

server to determine the group (for finite field DH) or curve (for Elliptic Curves DH) to be used; furthermore, the client needs to even provide its DH key-share (e.g., $g^a \bmod p$). We can provide all this - efficiently - in the Client_Hello, as a result of the TLS 1.3 use of a limited number of *standard DH finite groups and elliptic curves*.

The client indicates the DH groups (including elliptic curves) that it supports, in the **Supported_Groups** extension; and its DH key-share, in the **Key_Share** extension. Both extensions *must* be sent as part of the TLS 1.3 Client_Hello message. There is a cost here in overhead of computing and sending these values - but the savings in RTT are usually much more important. The client may provide key_share only for some of the groups; if there is no key_share for the group preferred by the server, the server can send a special message, **HelloRetryRequest**, and the client will send a ‘corrected’ Client_Hello. (In this case, the handshake will require two RTTs.)

Three other mandatory extensions in the TLS 1.3 Client_Hello identify the **supported_versions** (protecting version negotiation), the **signature_algorithms** supported and the **certificateAuthorities** (CAs) trusted by the client.

The Server_Hello message is quite similar to this of previous versions, except that it also provides the server’s key_share for the DH exchange, in an aptly named extension. It is followed by the server’s certificate and the TLS 1.3 CertificateVerify message, which authenticates the server *and* ensures the handshake integrity, by including a signature over the entire handshake (to this point). Finally, the server sends its (authenticated) Server_Finished message, again much like in previous versions (just earlier!).

The final flow of the handshake contains the Client_Finished message, authenticating the entire handshake (until this point), much like in earlier versions.

Both client and server Finished messages may be followed immediately by application data sent by the respective parties, protected using the shared key. Notice that since the TLS 1.3 record protocol uses AEAD to protect the data, it uses only one key in each direction (client to server, $k_{C \rightarrow S}$, and server to client, $k_{S \rightarrow C}$); the figure includes only a message from the client. We denote the AEAD protection provided by the record protocol by $\boxtimes_{k_{C \rightarrow S}}(Application\ data)$.

7.6.3 TLS 1.3 Full (1-RTT) Pre-Shared Key (PSK) Handshake

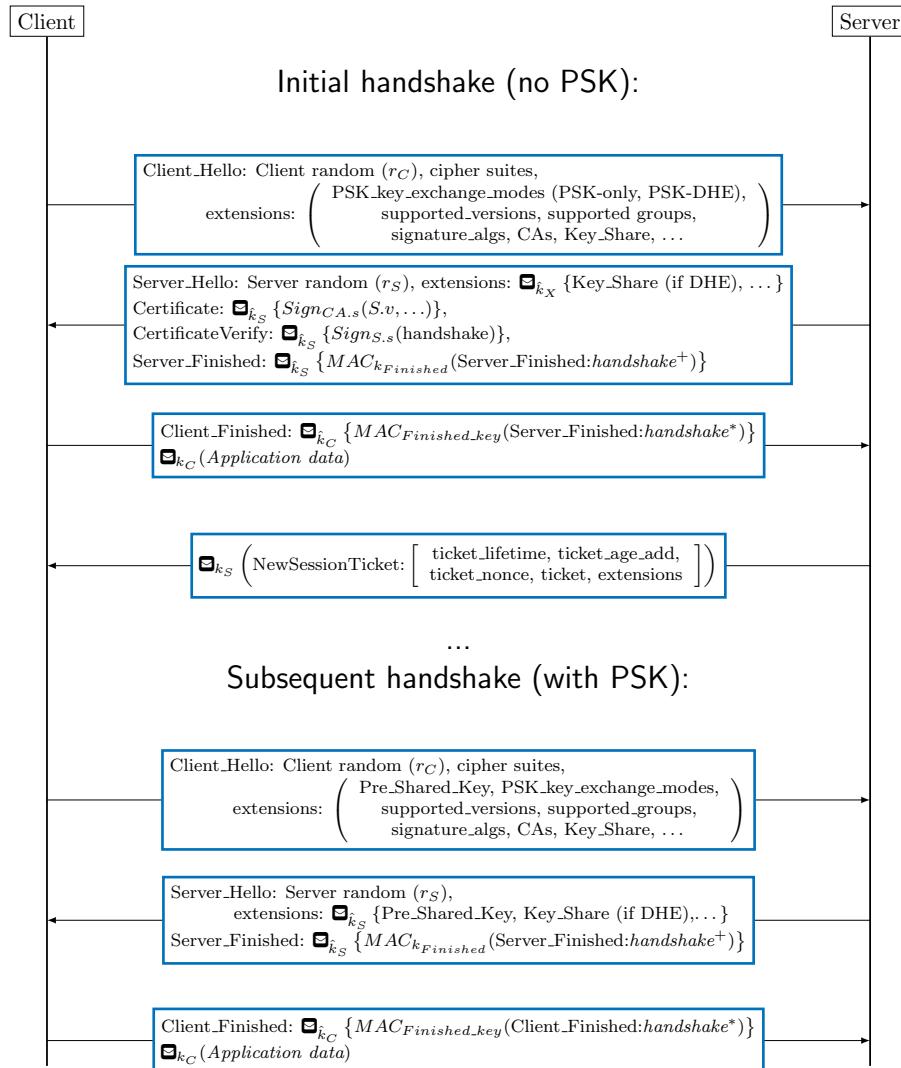


Figure 7.22: TLS 1.3 Full (1-RTT) Pre-Shared Key (PSK) handshake. We show the typical use of a PSK, for session resumption, although a PSK can also be shared off-band. We show two handshake: an initial handshake where the PSK is established, and a subsequent handshake which uses the PSK. To establish a PSK, the Client_Hello includes the PSK_key_exchange_modes extension, indicating if the PSK can be used to authenticate a DHE (providing PFS) and/or to provide shared-key only handshake (without PFS). See text for details.

In Figure 7.22, we present the *TLS 1.3 Full (1-RTT) Pre-Shared Key (PSK) Handshake*. The TLS 1.3 pre-shared key mechanism can be used for *session resumption*, using a key shared in an earlier session, or with out-of-band pre-shared keys, as previously supported by several special cipher suites [16, 17, 118].

For both session resumption and out-of-band pre-shared keys, TLS 1.3 supports two *Pre-Shared Key modes*, which offer different costs and benefits:

The PSK Key Exchange (PSK_KE) mode: the pre-shared key is used to secure the session without the use of Diffie-Hellman key exchange, or any other public key operation. The advantage is reduced computational costs and associated delay and energy costs; as shown in Table 6.1, symmetric cryptography requires a tiny fraction of the computational resources required by public-key cryptography. There is also a reduction in the amount of data sent, which is meaningful in some unusual situations; usually, this reduction is insignificant. The disadvantage of the a PSK Key Exchange (PSK_KE) mode, is that it does not ensure *perfect forward secrecy (PFS)*.

The PSK and DHE (PSK_DHE_KE) mode: the pre-shared key is used to authenticate the Diffie-Hellman key exchange, instead of using the server's certificate and signature (with the CertificateVerify message). This mode has three uses. The first is simply to reduce the overhead of the transmission and verification of the certificate and the signature, while retaining the added security provided by PFS. The second is to use is for scenarios where the server does not have a certificate and cannot perform the certificate-based verification (CertificateVerify message), and the security of the DH key exchange is based on the shared-key authentication. The third usage is when the server does send both certificate and signature; in this case, the goal is the added security provided by the shared key, e.g., in case of exposure of the server's private key.

In Figure 7.22, we focus on the session resumption case, by presenting a sequence of two handshakes. The first is an *initial handshake*, which is essentially a full (1-REE) DH handshake, which further *establishes one or more Pre-Shared Keys*, using the PSK_key_exchange_modes extension and the NewSessionTicket message. It is followed by the second, a *subsequent handshake*, which *uses* the Pre-Shared Key from the previously sent ticket, to *resume the session*, by performing a pre-shared key handshake.

The PSK_key_exchange_modes extension specifies which Pre-Shared Key mode(s) the client wants to use: the PSK Key Exchange (PSK_KE) mode and/or the PSK Key Exchange (PSK_KE) mode. This extension is relevant for the use of the PSK in the current handshake (if it uses a pre-shared key), and for any new pre-shared keys which the server may share using the NewSessionTicket message.

The NewSessionTicket message provides the client with one or more 'tickets'. The ticket(s) sent after a successful handshake, refer to a pre-shared key, derived from a dedicated shared secret called the *resumption-master-secret*.

The `resumption_master_secret` is one of the secrets and keys derived for the session; the derivation uses a keyed function, which we denote h^{Expand} . As a simplification, consider `resumption_master_secret` to be a secret key, and h^{Expand} to be a pseudorandom function (PRF); for more precise details, see subsection 7.6.5.

The PSK associated with the ticket is derived from the `ticket_nonce`, which is one of the fields sent in the `NewSessionTicket`, and the `resumption_master_secret`, as follows:

$$\text{PSK}(\text{ticket_nonce}) = h_{\text{resumption_master_secret}}^{\text{Expand}}(\text{"resumption"}, \text{ticket_nonce}) \quad (7.26)$$

The Client_Hello message of the subsequent handshake, using the PSK, includes two PSK-related extensions: the `PSK_key_exchange_modes` extension, discussed above, and the `Pre_Shared_Key extension`. The `Pre_Shared_Key` extension identifies one or more pre-shared keys known to the client, allowing the server to use any of these that the server may have cached to establish the new connection. For session resumption, the PSK identifier is the `ticket`, provided in a previous `NewSessionTicket` message. The server also uses the `Pre_Shared_Key` extension to signal that it uses a specific pre-shared key (from the list provided by the client).

In Figure 7.22, the subsequent handshake contains the `Key_Share` extension. This extension includes the server's DH key-share, and therefore, is used if, and only if, using the DH key exchange (with finite field or elliptic curve). We conclude, therefore, that the client's `PSK_key_exchange_modes` extension allowed the use of PSK_KE, which was then chosen by the server.

7.6.4 TLS 1.3 Zero-RTT Pre-Shared Key (PSK) Handshake

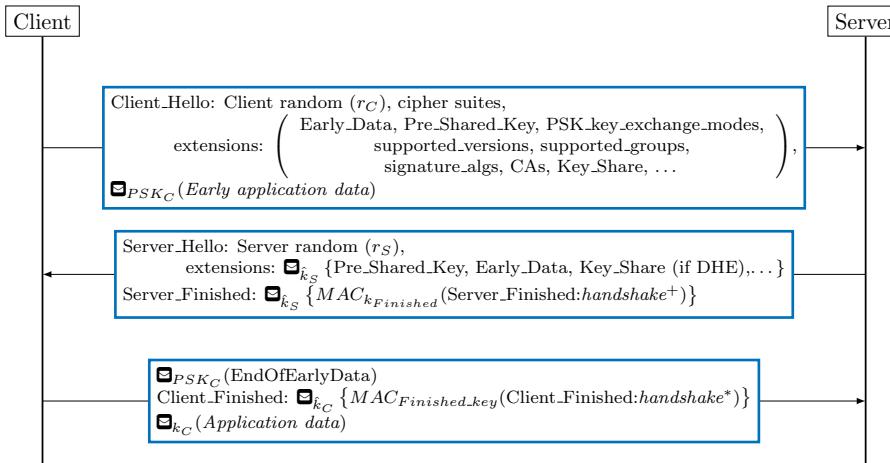


Figure 7.23: TLS 1.3 Zero-RTT Pre-Shared Key (PSK) Handshake. The client provides some ‘early application data’ to the server immediately after the Client_Hello message, without waiting for the server’s response.

In Figure 7.23 we present the *TLS 1.3 Zero-RTT Pre-Shared Key (PSK) Handshake*. This is a special form of a pre-shared key handshake, in which we use the PSK to *secure some data sent in the Client_Hello message*. Specifically, this data is contained in the dedicated **Early_Data** extension, which is sent in the first flow, sent from the client to the server.

Since Early_Data is sent as part of the very first flow of the connection, the delay until it arrives includes only the time since the client sends this message, and until the server receives it. Namely, there is no need to wait for any round-trip of handshake messages, before the client can send this ‘early’ application data to the server. In other words, the Early_Data is sent without waiting for any round-trips to complete, i.e., with *zero RTT latency*. The only delay is, therefore, the unavoidable time for the data itself to transfer from client to server.

The Early_Data does not benefit from all the protections offered by TLS. In particular, it is only protected by the pre-shared key, therefore it does not benefit from perfect forward secrecy (PFS). Furthermore, an attacker can replay the Client_Hello message; this may cause the server to re-process the Early_Data, unless appropriate countermeasures prevent this (see below). Therefore, Early_Data should only be used for client requests which do not require PFS, and either where re-processing is allowed, or with appropriate countermeasures to prevent re-processing. A typical example for a legitimate, common use of Early_Data (and zero-RTT handshake), where both the lack of PFS and the possibility for re-processing are not a concern, is when the client

sends a query to the server, authenticated by a password or cookie sent with the request, and receives back a (protected) response.

Two alternative countermeasures against re-processing of Early Data are possible. The first possible countermeasure is to allow each ticket to be used only once; this requires the server to maintain track of already-used tickets, similarly to the use of session-ID for session resumption in earlier versions. An alternative countermeasure is to limit the lifetime of the ticket, using the *ticket_lifetime* field of the NetSessionTicket message.

7.6.5 TLS 1.3 Key Derivation

TLS 1.3 makes extensive use of key derivation, using a pair of keyed functions: a *key expansion* function h^{Expand} and a *key extraction* function $h^{Extract}$. Both functions are defined in [201], based on the *HMAC* construction and a given hash function h . The hash function h is defined as part of the TLS 1.3 cipher suite.

The key expansion function h^{Expand} is similar to a PRF; like a PRF, it should receive a pseudorandom key, and outputs a pseudorandom string. The length of its output pseudorandom string is specified as one of its inputs. It receives one more input, which is basically the input information to the PRF. The TLS specifications defines different inputs for each derivation.

The key extraction function $h^{Extract}$ is similar to a keyed *Key Derivation Function (KDF)*. Namely, the output of $h_x(y)$ is a (short) pseudorandom string, provided the *either* (a) x is pseudorandom (secret) and y is a unique value, *or* (b) x is a ‘salt’ (randomly chosen but known to attacker), and y is a high-entropy string, i.e., intuitively, contains ‘sufficient secret random bits’.

The TLS 1.3 handshake protocol applies $h^{Extract}$ and h^{Expand} to derive multiple pseudorandom, independent keys for different purposes. Simplifying:

1. When using a pre-shared key PSK , we use it (or actually, a key derived from it) as a key to h^{Expand} , to generate several pseudorandom ‘early secrets/keys’. We use one key to protect the Early_Data (if sent), and another key, which we denote k_1 , as the key for the next derivation step.
2. We next derive $k_{handshake} = h_{k_1}^{Extract}(\text{shared_secret})$, where shared_secret is the partially-secret output of the Diffie-Hellman key exchange. We use $k_{handshake}$ as the key to h^{Expand} , to generate several pseudorandom keys, including a key to protect client-to-server handshake messages, a key to protect server-to-client handshake messages, and a key k_2 which we use for the next derivation step.
3. We next derive $k_{Master} = h_{k_2}^{Extract}(0)$. We use k_{Master} as the key to h^{Expand} , to generate several pseudorandom keys, including a key to protect client-to-server application messages, a key to protect server-to-client application messages, and a key $\text{resumption_master_secret}$ which we use to derive pre-shared keys, as defined in Equation 7.26.

7.6.6 Cross-Protocol Attacks on TLS 1.3

We complete our discussion of TLS 1.3 by briefly discussing *cross-protocol* attacks, since these types of attacks *can* be effective standard-complying implementations of TLS 1.3; arguably, the standard could have specified certain countermeasures which would have prevented this, as we discuss below.

Let us first explain what are cross-protocol attacks; the concept is not limited to TLS. Cross-protocol attacks are attacks which exploit a vulnerability in one ('weak') protocol P_W , to attack another, 'strong' protocol, P_S . The attacks exploit *two* flaws: the vulnerability of P_W , and the fact that P_S and P_W use the same private/secret key, in violation of the *key separation principle* (Principle 10).

In particular, while TLS 1.3 does not use RSA encryption, it does use signatures - which are often RSA signatures - for CertificateVerify. In many implementations, the same private key is used for these TLS 1.3 RSA signatures, and for vulnerable RSA decryption, typically, using version 1.5 or 2.0 of PKCS#1. In particular, many TLS 1.3 implementations were found to use the same private key (to sign CertificateVerify) as used for RSA decryption by older SSL/TLS implementations (of the same organization). This allows use of the Bleichenbacher attack and variants of it, e.g., Manger's attack, to allow the attacker to 'perform the private-key RSA operation', potentially allowing the attacker to sign the CertificateVerify message. Two variants of this attack were published. The first attack was published in [179]; however, it required quite extensive computational abilities from the attacker. The DROWN attack, published a year later, is a much more efficient and practical attack, but it required the key to be shared with an implementation of SSLv2; surprisingly, the researchers found that even in 2016, there was a very significant number of web servers which supported SSLv2 (and reused the same private key for other protocols, e.g., TLS 1.3 signing).

These cross-protocol attacks are not necessarily a major concern for the use of TLS 1.3, for two reasons. The first reason is a technical challenge: the attacker must complete the attack, and in particular abuse the lower-version TLS version to sign the CertificateVerify message, before the client aborts the connection (due to not receiving CertificateVerify in time). This challenge makes deployment of the attack challenging - but it may still be possible in some scenarios.

The second reason for the defense to be of limited concern, is that the attack only works when the TLS 1.3 implementation *uses the same private key* (for signatures) as used by lower versions of TLS (for encryption). Such reuse of the same key for two different purposes (decryption and signing) and by two different versions of TLS, is a *double violation* of the key separation principle (Principle 10). Implementations of TLS 1.3 should avoid this, and use a different private (signing) key than the private (decryption) key used by lower versions - in fact, such key separation should have been done even before this attack was published! Furthermore, typically, the same private key is used for two different purposes and protocol, only when using also the same

certificate. Specifically, TLS 1.3 could require that the certificate contains a *key-usage* extension, that explicitly *forbids* its use for decryption. While this would not absolutely prevent cross-protocol attacks, it would probably make them extremely unlikely, as certificates of keys currently used (for encryption) by older versions of SSL/TLS are unlikely to contain a *key-usage* extension forbiddings the use of the private key for decryption. For details about the key-usage and other certificate extensions, see subsection 8.2.7.

7.7 SSL/TLS: Final Words and Further Reading

The SSL/TLS protocols are the most widely used, and definitely the most studied, applied cryptographic protocol. Extensive efforts by many cryptographers and security experts were invested in validating and improving the security of SSL/TLS. It is instructive to observe that these efforts resulted in the discovery of a large number of serious vulnerabilities. We discussed several of these, focusing on specification vulnerabilities. We summarize the most important SSL/TLS specifications vulnerabilities in Table 7.2. We did not cover the many *implementation vulnerabilities*, although some of them, in particular the Heartbleed bug [77, 328], had comparable and even greater impact.

Name	Year	References	Versions	Ciphers	Type
BEAST	2011	subsection 7.2.4, [110]	SSL, TLS 1.0	CBC	Cryptanalysis
CRIME	2012	subsection 7.2.6, [277]	TLS 1.0-1.1	n/a	Compression
Lucky13	2013	[7]	SSL, TLS 1.0-1.2	CBC	Padding, timing
RC4-biases	2013	subsection 7.2.5, [9]	SSL, TLS 1.0-1.2	RC4	Cryptanalysis
BREACH	2013	subsection 7.2.6, [139]	TLS 1.0-1.2	n/a	Compression
TIME	2013	subsection 7.2.6, [23]	TLS 1.0-1.2	n/a	Compression
Poodle-padding	2014	subsection 7.2.3, [238]	SSLv3	CBC	Padding
Poodle-downgrade	2014	subsection 7.5.5, [238]	TLS 1.0-1.2	n/a	Version downgrade
FREAK	2015	subsection 7.5.3, [47]	TLS 1.0-1.2	RSA	Cipher suite downgrade
Cross-Bleichenbacher	2015	subsection 7.6.6, [179]	TLS 1.3	RSA	Cross-protocol and Bleichenbacher
DROWN	2016	subsection 7.6.6, [14]	TLS 1.0-1.3	RSA	Cross protocol and Bleichenbacher
Logjam	2018	subsection 7.5.3, [5]	TLS 1.0-1.2	DHE	Cipher suite downgrade
ROBOT	2018	[63]	TLS 1.0-1.2	RSA	Bleichenbacher
Bleichenbacher's CAT	2019	subsection 7.5.4, [283]	TLS 1.0-1.3	RSA	Bleichenbacher and downgrade

Table 7.2: Some important TLS/SSL attacks due to specifications vulnerabilities, 2011-2019.

We can learn some important lessons from this history of attacks and improvements, including:

A crack today, a break tomorrow: many devastating attacks, e.g., BEAST and Poodle, can be traced to vulnerabilities reported years earlier, but ignored since they appeared impractical. Or as correctly stated in [7], *attacks only gets better*.

Vulnerabilities are resilient and return: even after a vulnerability has been discovered and countermeasures adopted, attackers are often able to continue taking advantage of the vulnerability, in different ways. First, attackers are often able to adjust the attack and defeat the countermeasures. Second, attacker are often able to *downgrade* the system to use an

outdated, vulnerable version, possibly using downgrade attacks. Third, attackers are sometimes able to use cross-protocol attacks, where they exploit a vulnerable system to circumvent the (strong) defenses of *another* system which uses the same keys. It is much better to design systems securely from early on; of course, an advice easier given than followed!

Separate keys: Cross-protocol attacks such as DROWN [14], as well as BEAST [110] which abuses the continued use of the same key for different messages, remind us of the important principle of *key separation* (Principle 10). The use of TLS 1.3, or of any secure protocol, will not help, if we reuse the same secret key which can be found from its usage in an insecure protocol!

Test, test, test: finally, while we focused on specification flaws, many attacks on SSL/TLS, e.g., Heartbleed, exploit *implementation flaws*. Testing for security is difficult, however, vital for the security of the system, since vulnerabilities will not be detected by normal use of the system. Of course, testing is harder for larger and more complex systems; which brings us to the next and final item...

KISS! Finally, we see again and again the importance of the KISS principle (Principle 14): keep specifications, design and code small and simple, minimize complexity and attack surface and avoid unnecessary options and flexibility. The KISS principle is important against both specification and implementation vulnerabilities. The design of TLS 1.3 began with an extensive effort to follow this principle and eliminate unnecessary options. However, with features creeping in, there are reasons to be concerned that vulnerabilities may be found, in implementations and/or in the specifications themselves.

7.8 Additional Exercises

Exercise 7.5 (The SSL/TLS record protocol fragmentation and compression).

1.

The SSL/TLS record protocol uses fragments of size up to 16KB. Explain potential disadvantage of using much longer fragments (or no fragments).

2. *Explain potential disadvantage of using much shorter fragments.*
3. *Explain why fragmentation is applied before compression (for the AtE protocol).*
4. *Suppose compression is not used. Why would we apply fragmentation before authentication and encryption?*
5. *The SSL/TLS record protocols apply compression, then authentication. Is it possible to reverse the order, i.e., apply authentication and then compression? Can you identify advantages to either order?*

Exercise 7.6 (Vulnerability of Compress-then-Encrypt). *One of the important goals of SSL/TLS is to hide the value of cookies sent by a browser, as part of an HTTP-over-SSL/TLS connection (marked by the https protocol in the beginning of the URL). Cookies are strings that are sent automatically by the browser to a website; they are often used to authenticate the user. Consider a cross-site attacker, as in subsection 7.2.6, i.e., the attacker controls a rogue website visited by the victim user; further, assume that the attacker can eavesdrop on the (protected) communication, following the CPA-Oracle Attack model. This allows the attacker to control the contents of the request sent by the browser, except for the Cookie HTTP header, which is added by the browser and which consists of the string ‘Cookie:’ followed by the value of the (unknown) cookie.*

1. *Assume that the length of the cookie is known, that it contains only alphanumeric characters, and the following compression scheme. Check what is the longest string α which appears at least twice in the uncompressed data; replace the occurrences of α by a special character, say ‘!’, and concatenate to the data the same special character (e.g. ‘!’), followed by α . Present an efficient attack which exposes the first character of the cookie.*
2. *Extend the attack, to find the entire cookie.*
3. *What is the maximal number of requests required by the attack, for a cookie of l characters?*
4. *What is the expected number of requests required by the attack, for cookie of l alphabetic characters selected randomly (with uniform distribution)?*

Note: compression is used by all versions of SSL/TLS before TLS 1.3.

Exercise 7.7 (SSLv2 key derivation). *SSL uses MD5 for key derivation. In this question, we explore the required properties from MD5 for the key derivation to be secure.*

1. *Show that it is not sufficient to assume that MD5 is collision-resistant, for the key derivation to be secure.*
2. *Repeat, for the one-way function property.*
3. *Repeat, for the randomness-extraction property.*
4. *Define a simple assumption regarding MD5, which ensures that key derivation is secure. The definition should be related to cryptographic functions and properties we defined and discussed.*

Exercise 7.8 (BEAST vulnerability). *Versions of SSL/TLS before TLS1.1, use CBC encryption in the following way. They select the IV randomly only for the first message m_0 in a connection; for subsequent messages, say m_i , the IV is simply the last ciphertext block of the previous message. This creates a vulnerability exploited, e.g., by the BEAST attack and few earlier works [19, 110].*

In this question we explore a simplified version of these attacks. For simplicity, assume that the attacker always knows the next IV to be used in encryption, and can specify plaintext message and receive its CBC encryption (using the known next IV). Assume known block length, e.g., 16 bytes.

1. Assume the attacker sees ciphertext (c_0, c_1) resulting from CBC encryption with c_0 being the IV, of a single-block message m , which can have only two known values: $m \in \{m_0, m_1\}$. To find if m was m_0 or m_1 , the adversary uses fact that it knows the next IV to be used, which we denote c'_0 , and asks for CBC encryption of a specially-crafted single-block message m' ; denote the returned ciphertext by the pair (c'_0, c'_1) , where c'_0 is the (previously known) IV, as indicated earlier. The adversary can now compute m' from c'_1 :

- a) What is the value of m' that the adversary will ask to encrypt?
- b) Fill the missing parts in the solution of the adversary:

$$m = \begin{cases} m_0 & \text{if } \underline{\hspace{2cm}} \\ m_1 & \text{if } \underline{\hspace{2cm}} \end{cases}$$

2. Show pseudo-code for the attacker algorithm used in the previous item.
3. Show pseudo-code for an attack that finds the last byte of message m . Hint: use the previous solution as a routine in your code.
4. Assume now that the attacker tries to find a long secret plaintext string x of length l bytes. Assume attacker can ask for encryption of messages $m = p + x$, where p is a plaintext string chosen by the attacker. Show pseudo-code for an attack that finds x . Hint: use previous solution as routine; it may help to begin considering fixed-length x , e.g., four bytes.

Sketch of solution to second part: Attacker makes query for encryption of some one-block message y , receives α_0, α_1 where $\alpha_1 = E_k(\alpha_0 \oplus y)$. Suppose that now, the attacker knows the value IV to be used for encryption of the next message. Attacker picks $m_0 = IV \oplus y \oplus \alpha_0$, and m_1 some random message. If the game picks bit $b = 0$, then attacker receives encryption of m_0 ; this encryption would be $E_k(IV \oplus m_0) = E_k(IV \oplus IV \oplus y \oplus \alpha_0) = E_k(y \oplus \alpha_0) = \alpha_1$ (and the IV). Otherwise, if the game picks $b = 1$, then the attacker receives some other string.

Sketch of solution to third part: solution to previous part allowed attacker to check if the plaintext was a given string; we now simply repeat this for the 256 different strings corresponding to all possible values of last byte of m_2 . \square

Exercise 7.9 (Non-random client/server). Some devices may not have a source of random bits; in this exercise, we explore possible resulting vulnerabilities, and a possible work-around.

1. Consider a IoT-lock, which receives the lock/unlock requests over SSL/TLS, as a web-server - but without access to a source of randomness. Show a replay attack allowing an eavesdropping attacker to open the lock by replaying messages of a legitimate user.
2. Consider a monitoring station displaying images from security cameras, by initiating SSL/TLS connections to the cameras and receiving the current images. Show a replay attack allowing an eavesdropping attacker to replay old images (while cracking the safe).
3. Assume that the devices have non-volatile memory. Show how it can be used to ensure secure interactions, even though the devices still do not have a source of random bits.

Your answer should present a sequence diagram and the relevant equations, for SSLv2 or another version of SSL/TLS.

Exercise 7.10 (SSLv2 client authentication). *The SSLv2 client authentication mechanism requires clients to sign a message containing three main fields: a random challenge sent by the server, the server's certificate, and the shared secret keys exchanged by the protocol. The signature uses RSA with the Hash-then-Sign paradigm, using the MD5 hash function. Furthermore, the client should encrypt the message containing the signature (using the just-established shared key). This is more complex than the later client-authentication designs; in this question, we explore attempts to remove some of these multiple requirements.*

1. Suppose that the input to the signature did not contain the server's certificate. Show a sequence diagram showing client-authentication fails.
2. Suppose that the signed message was not encrypted. Present a possible vulnerability of the protocol, where you are allowed to replace the use of MD5 with the use of any collision-resistant hash function h . Note: this may require h to have a vulnerability that we may not expect to find in MD5 or other well-designed cryptographic hash functions.

Exercise 7.11 (TLS handshake: resiliency to key exposure). *Fig. 7.10 presents the RSA-based SSL/TLS Handshake. This variant of the handshake protocol was popular in early versions, but later 'phased out' and completely removed in TLS 1.3. The main reason was the fact that this variant allows an attacker that obtains the server's public key, to decrypt all communication with the server using this key - before and after the exposure.*

1. Show, in a sequence diagram, how a MitM attacker who is given the private key of the server at time T_1 , can decrypt communication of the server at past time $T_0 < T_1$.
2. Show, in a sequence diagram, how TLS 1.3 avoids this impact of exposure of the private key.

3. Show, in a sequence diagram, how a MitM attacker who is given the private key of the server at time T_1 , can decrypt communication of the server at future time $T_2 > T_1$.
4. Explain which feature of TLS 1.3 can reduce the exposure of future communication, and how.

Exercise 7.12 (Protocol version downgrade-dance attack). *Implementations of TLS and SSL specify the version of the protocol in the ClientHello and ServerHello messages. If the server does not support the client's version, then it replies with an error message. When the client receives this error message ('version not supported'), it re-tries the handshake using the best-next version of TLS/SSL supported by the client. This method of ensuring backward compatibility with older versions of TLS/SSL is referred to as downgrade dance.*

1. Present a sequence diagram showing how a MitM attacker can exploit the downgrade dance mechanism, to cause the server and client to use an outdated version of the protocol, allowing the attacker to exploit vulnerabilities of that version.
2. The TLS Fallback Signaling Cipher Suite Value (SCSV) [237], discussed in subsection 7.5.6, is designed to mitigate this risk. Let v_c denote the TLS version run by the client and v_s denote the TLS version run by the server. Present a sequence diagrams showing TLS connections where (1) client and server support SCSV and $v_c > v_s$, (2) same, $v_c = v_s$, (3) same, $v_c < v_s$, (4) any of these, with a MitM attacker who tries to cause use of version $v_m < \min(v_c, v_s)$.

Note: See also Exercise 8.15.

Exercise 7.13 (Client-chosen cipher suite downgrade attack). *In many variants of the SSL/TLS handshake, e.g., the RSA-based handshake in Fig. 7.10, the authentication of the (previous) handshake messages in the Finish flows, is relied upon to prevent a MitM attacker from performing a downgrade attack and causing the client and server to use a less-preferred (and possibly less secure) cipher suite. However, in this process, the server can choose which of the client's cipher suites would be used. To ensure the use of the cipher suite most preferred by the client, even if less preferred by the server, some client implementations send only the most-preferred cipher suites. If none of these is acceptable to the server, then the server responds with an error message. In this case, the client will try to perform the handshake again, specifying now only the next-preferred cipher suite(s), and so on - referred to as downgrade dance.*

1. Show how a MitM attacker can exploit this mechanism to cause the server and client to use a cipher suite that both consider inferior.
2. Suggest a fix to the implementation of the client which achieves the same goal, yet is not vulnerable to this attack. Your fix should not require any change in the server.

Exercise 7.14 (TLS server without randomness). An IoT device provides http interface to clients, i.e., acts as a tiny web server. For authentication, clients send their commands together with a secret password, e.g., on, <password> and off, <password>. Communication is over TLS for security, with the RSA-based SSL/TLS handshake, as in Figure 7.10.

The IoT device does not have a source of randomness, hence, it computes the server-random r_S from the client-random, using a fixed symmetric key k_S (kept only by the device), as: $r_S = AES_{k_S}(r_C)$.

1. Present a message sequence diagram showing how an attacker, which can eavesdrop on a connection in which the client turned the device ‘on’, can later turn the device ‘on’ again, without the client being involved.
2. Would your answer change (and how), if the device supports ID-based session resumption? Ticket-based session resumption?
3. Show a secure method for the server to compute the server-random method, which will not require a source of randomness. The IoT device may use and update a state variable s ; your solution consists of the computation of the server-random: $r_S = \dots$ and of the update to the state variable performed at the end of every handshake: $s = \dots$.

Exercise 7.15 (DH Ephemeral (DHE)). Consider a client and server that use TLSv1.2 with DH Ephemeral (DHE) public keys, as in Fig. 7.12. Assume that the client and server run this protocol daily, at the beginning of every day i . (Within each day, they may use session resumption to avoid additional public key operations; but this is not relevant to the question). Assume that Mal can (1) eavesdrop on communication every day, (2) perform MitM attacks (only) every even day (i s.t. $i \equiv 0 \pmod{2}$), (3) is given all the keys known to the server on the fourth day. Note: the server erases any key once it is not longer in use (i.e., on fourth day, attacker is not given the ‘session keys’ established n previous days).

Fill the ‘Exposed on’ column of day i in Table 7.3, indicating the first day $j \geq i$ in which the adversary should be able to decrypt (expose) the traffic sent on day i between client and server. Write ‘never’ if the adversary should never be able to decrypt the traffic of day i . Briefly justify.

Day	Eavesdrop?	MitM?	Given keys?	Exposed on...	Justify
1	Yes	No	No		
2	Yes	Yes	No		
3	Yes	No	No		
4	Yes	Yes	Yes		
5	Yes	No	No		
6	Yes	Yes	No		
7	Yes	No	No		
8	Yes	Yes	No		

Table 7.3: Table for Exercise 7.15.

Exercise 7.16 (TLS with PRS). Consider a client that has three consecutive TLS connections to a server, using TLS 1.3. An attacker has different capabilities in each of these connections, as follows:

- In the first connection, attacker obtains all the information kept by the server (including all keys).
- In the second connection, attacker is disabled.
- In the third connection, attacker has MitM capabilities.

Is the communication between client and server exposed, during the third connection?

1. Show a sequence diagram showing that with TLS 1.3, communication during third connection is exposed to attacker.
2. Present an improvement to TLS 1.3 that will protect communication during third connection. Simplify your solution by assuming no attack during the second connection.
3. Further to provide same protection, even if the attacker can eavesdrop to the communication during the second connection.
4. How can your improvement be implemented using TLS 1.3, allowing backward compatibility, i.e., a ‘normal’ TLS 1.3 interaction when one of the two parties (client or server) does not support your improvement?

Exercise 7.17. A Pierpont prime is a prime number of the form $2^u \cdot 3^v + 1$, where u, v are non-negative integers; Pierpont primes are a generalization of Fermat primes. Assume that Alice’s browser sends, in the Client Hello message of TLS 1.3, the set of exponentiations $\{g_i^{a_i} \bmod p_i\}$, where for some i , say $i = 3$, the prime p_3 is a Pierpont prime.

1. Assume that the server BOB.COM selects to use this p_3 and $g_3^{a_3} \bmod p_3$, i.e., sends back $g_3^{b_3} \bmod p_3$ as part of the server-hello message. Present a sequence diagram showing how a MitM attacker would be able to eavesdrop and modify messages sent between Alice and Bob.
2. Assume that the server prefers p_2 , such that there is some prime q_2 such that $p_2 = 2 \cdot q_2 + 1$. Explain why the MitM attack of the previous item fails.
3. Present a sequence diagram showing that the attacker is still able to impersonate as the website.
4. Extend the impersonation attack to a complete MitM attack against Alice and BOB.COM, assuming typical user authentication (using cookie or userid/password).

5. Suppose the client hold a pre-shared key (and ticket); would this prevent the attack? explain.

Exercise 7.18. This exercise continues Exercise 2.43, please see the ANSI X9.31 design presented there.

Some TLS implementations use X9.31 as a PRG, to generate keys, nonces and IVs for encryption. Assume fixed key k (known to attacker), and that the values T_i are the current time, in seconds. Explain a possible vulnerability; you may make reasonable assumptions, e.g., on the use of different outputs of the PRG and on clock synchronization. Demonstrate how an attacker may exploit the vulnerability, using a sequence diagram. You may present the attack into any variant of TLS that you wish.

Exercise 7.19 (Protecting TLS 1.3 servers from computational DoS). TLS servers can be subject to a Denial-of-Service (DoS) attack, in which the attacker overload the server with Client_Hello messages, each time causing the server to perform computationally-intensive operations. In TLS 1.3, the attacker can cause the server to perform two or three computationally-intensive operations: signing the CertificateVerify message and computing the server's Diffie-Hellman key share, and possibly also computing the Diffie-Hellman shared key. All this requires minimal computational costs to the attacker.

1. Explain the attack, using sequence diagram.
2. Explain how the Cookie extension of TLS 1.3 can help against this attack, following the details in [272]. Identify assumptions/limitations of this defense.
3. An alternative way to defend against such DoS attacks, uses the pre-shared key to authenticate the Client_Hello message. This defense can allows the server to establish connections with clients which has a pre-shared key, even when, due to the attack, other clients cannot establish a connection. Such defense does not currently exist in TLS 1.3. Design such defense and explain it, using appropriate sequence diagrams. Your solution should not require any change in the TLS 1.3 handshake protocol.

Chapter 8

Public Key Infrastructure (PKI)

A big advantage of public key cryptography is that public keys are easier to distribute, as they are not secret; we only need to ensure their *authenticity*. The main mechanism for authenticating public keys is a *public key certificate* (or simply a certificate), signed by a trusted *Certificate Authority (CA)*.

In Chapter 7, we sketched how certificates are used by the TLS protocol, but without discussing certificates or CAs. This chapter focuses on certificates; we discuss how certificates are issued, validated and revoked, and how to determine which certificates, signed by which authorities, are trustworthy. This set of mechanisms is key to the use of public key, and therefore referred to as *Public Key Infrastructure (PKI)*. PKI is, therefore, an essential component for practical deployments of public key cryptography (PKC).

The basic concept of PKI is almost as old as the first publications of public key cryptography, and appeared in [197]. PKI was also standardized quite early, before any application of PKC; this was in the (first version of the) X.509 specifications [78]. The X.509 standard has evolved over the years; the most important change was with the publication of X.509 version 3 in 1997, often referred to simply as *X.509v3*. X.509v3 is still the most important and widely used version of certificate, and much of our discussion in this chapter is focused on it (specifically, Section 8.2 to Section 8.5). X.509v3 is designed for generality; there are several published *X.509 profiles*, which define different restrictions on the contents and use of X.509 certificates. We mostly focus on the most well-known profile, the *PKIX* profile, used by Internet protocols and most other deployed PKIs, and defined in RFC 5280 [86, 173].

We also discuss some extensions of X.509, mainly *OCSP* (RFC 6960) [287] and *Certificate Transparency (CT)* [209–211]). All of these (X.509 with the PKIX profile as well as CRLs, OCSP and CT) are used in common implementations of the *TLS/SSL* handshake protocols, which we discussed in Chapter 7, and in particular its application to secure the communication between browsers and web-servers; this particular application is often referred to as the *Web PKI*. The Web PKI is probably the most well-known, and possibly also most important, application of PKI; see subsection 8.1.3.

For many years, PKI was basically identified with X.509v3, and to a large extent, this is still mostly true; all significant PKI deployments and mechanisms follow X.509v3. One exception was the handling of *revocations*, where the original *CRL* proposal was found too inefficient, and other mechanisms have been deployed and proposed, with no dominant standard yet; see Section 8.4.

However, with the growing importance and use of the Web, there were growing concerns about vulnerabilities of the PKI system, due to repeated, high profile *PKI failures* (see subsection 8.5.1). Several proposals were made to improve PKI security, and one of them, *Certificate Transparency*, has been widely deployed, including by CAs and browsers, and is being standardized by the IETF. However, CT is also based on X.509, and only *extends* the X.509 specifications, as we discuss in Section 8.6.

Topics. PKI is important and involves many proposals, mechanisms and details; this chapter covers what seemed the most important aspects, and readers may want to further focus, at least in first reading, on the aspects most important to them. All readers should probably read the ‘PKI concepts and goal’ in Section 8.1. X.509 is covered mostly in Section 8.2, with the important aspect of intermediate-CAs and certificate-path in Section 8.3; readers may skip some of the details of the different extensions and even entire Section 8.3, if their goal is to get a more high-level understanding of PKI. Section 8.4 discusses certificate revocation, including both the *CRL* and OCSP standards, as well as other, ‘optimized’ designs. In Section 8.5, we discuss some of the criticisms of Web PKI, the main current deployed PKI system, and some propose improvements, while Section 8.6 focuses on the emerging Certificate Transparency extension to the X.509 PKI.

8.1 Introduction: PKI Concepts and Goals

Basic PKI entities: relying party, issuer (CA) and subject. Public Key Infrastructure (PKI) schemes distribute a public key pk together with a set $ATTR$ of *attributes* and a *signature* σ . The signature σ is the result of a signature algorithm applied to input containing both pk and $ATTR$. The tuple $(pk, ATTR, \sigma)$ is called a *public key certificate* or simply a *certificate*. The certificate is *issued* by an entity referred to as a *Certificate Authority (CA)* or as the *issuer*.

Most attributes refer to the *subject* of the certificate, i.e., the entity who knows (‘owns’) the private key corresponding to the certified public key pk . In addition, there are often additional attributes related to the certificate itself rather than to the subject, such as the certificate validity period and serial number.

The basic feature of public key cryptography is that the party that knows the *private key* is, usually, *different* than the party that *uses* the corresponding *public key*. To use the public key, we need to authenticate it; and usually this is done by verifying a certificate. Namely, the party that uses a public key *relies upon* the public key certificate, and on the PKI processes used to validate and

(at least one) trusted Certificate Authority (CA). Therefore, we refer to this party as the *relying party*.

The X.509 certificate life cycle. Figure 8.1 illustrates the X.509 PKI entities and certificate life cycle; some other PKIs have different life-cycles, sometimes with additional entities. In particular, Certificate Transparency has additional parties and a more complex life cycle; see Section 8.6. But for now, let's focus on the more basic scenario of X.509 PKI.

To request an X.509 certificate, the subject typically generates a (public, private) key pair, and then requests a CA to issue the certificate for the public key, with specific requested attributes, such as identifiers. The CA should *validate* that the public key was received from a subject which is entitled to the requested attributes; in subsection 8.2.8 we discuss the main validation methods: the ‘easy’ *domain validation*, the ‘classical’ *organization validation* and the extra-secure *extended-validation*. If the validation passes, the CA constructs the certificate, including the validated attributes from the relying party, as well as other attributes determined by the CA, such as serial number and validity period, and then signs it, using the CA’s private signing key.

After issuing the certificate, the CA sends it to the subject, who provides it to the relying party, often, and in particular for Web PKI, during the TLS/SSL handshake. The relying party should validate the certificate; this includes validation of the signature, using the public key of the CA and validation of attributes within the certificate (e.g., expiration time).

In the typical example of Figure 8.1, the subject is the website *bob.com*, and the relying party is Alice, or Alice’s browser. The figure shows the simple case of a typical identity certificate, issued to website *bob.com* *directly* by a CA trusted by the relying party. Such directly-trusted CAs are called ‘trust anchors’ or ‘root CAs’. In reality, most Web PKI certificates are *indirectly-issued*, i.e., issued by an intermediate-CA (Figure 8.6), or even by a path of multiple intermediate-CAs, as we discuss in Figure 8.7.

Usually, certificates are used until they expire, i.e., throughout their validity period, which is typically few months (rarely over a year). Around the expiration date, certificates are often *re-issued*. However, sometimes, a certificate should be *revoked*, i.e., invalidated before its planned and specified expiration time. Revocation is done by the CA, usually upon appropriate (and authenticated) request from the subject. A simple revocation mechanism called *Certificate Revocation List (CRL)* has been part of X.509 from its earliest versions [78]; however, practical, efficient certification turned out to be quite a challenge, and multiple designs were proposed; we discuss revocation mechanisms in Section 8.4.

While revocations can occur for administrative reasons, most revocations are due to security concerns, such as:

Subject key exposure: private keys should be well protected from exposure; however, exposures do happen. Normally, exposures are quite rare and sporadic. However, a discovery of a software vulnerability may cause

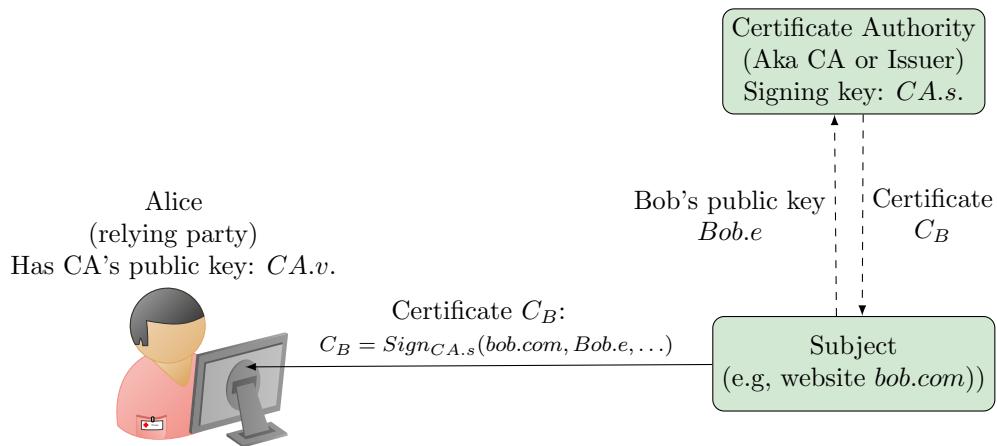


Figure 8.1: PKI Entities and typical application for server-authentication in Web PKI process. Here, we show the simple case of a typical identity certificate issued by a trusted CA ('trust anchor' or 'root CA') to website *bob.com*. The certificate is signed using the CA's private signing key, denoted *CA.s*, and validated by the relying party using the CA's public key *CA.v*, which should be known to the relying party. Dashed arrows represent the certificate issuing process, occurring once, before client connections. See also Figure 8.6 and Figure 8.7 for issuing with one or more intermediate-CAs, and Figure 8.5 for the certificate fields.

exposure of many private keys, as happened due to the *Heartbleed Bug* [77, 328].

CA failures: usually, certificate authorities have operated in a secure, trustworthy manner, and issued correct certificates to the rightful subjects - as required and expected. However, there have also been several incidents where CAs have failed in different ways, including *vulnerable subject identification*, e.g., insecure email validation, *issuing intermediate-CA certificates to untrusted entities*, e.g., to all customers, and even *CA compromise and issuing of rogue certificates* or what appears to be *intentional issuing of rogue certificates*. See subsection 8.5.1 and Table 8.5.

Cryptanalytical certificate forgery: certificate-based PKIs all use and depend on the *Hash-then-Sign* mechanism, and therefore become vulnerable if the signature scheme is vulnerable - or if the hash-function used is vulnerable. Specifically, certificate forgery was *demonstrated* when using hash functions vulnerable to *chosen-prefix collision* attacks, specifically, using MD5 [301], and later also using SHA-1 [216, 301]. See Chapter 3.

Subject key exposures are the most common reason for revocation, and typically results in a 'steady' rate of several dozens of revocations daily; however, software and CA vulnerabilities can result in exceptional 'waves' of revocations.

For example, the Heartbleed bug resulted in several days with many revocation, even more than 10,000 on one day [77, 328].

Identity certificates. Many certificates include an *identifying* attribute, i.e., an *identifier* of the subject; such certificates are referred to as *identity certificates*. In the typical server-authentication use by TLS/SSL of Web PKI, the relying-party is the browser, the subject is the website, and the relevant identifier is the *domain name* of the website, e.g., *bob.com*. This typical use-case is illustrated in Figure 8.1, where the certificate C_B contains a signature $Sign_{CA.s}(bob.com, Bob.e, \dots)$, using $CA.s$, the private signing key of the CA, over the identifier *bob.com*, the public key *Bob.e*, and other fields.

8.1.1 Rogue certificates

The basic goal of PKI is to allow a relying party to determine which public key to use, or whether to use a given public key - typically, included in a certificate. We use the term *rogue certificate* for a certificate which contains wrong or misleading information, and hence, should not be relied upon.

The basic goal of rogue certificates is to allow the attacker to mislead the user or security mechanisms, typically by impersonating as a trusted entity: impersonate as a trusted website (*website spoofing*), impersonate as a web-server of a trusted sender (*phishing email*), impersonate as a trusted software provider (*signed malware*).

Equivocating certificates. To mislead security mechanisms, the rogue certificates need to use exactly a specific name that ‘belongs’ to the legitimate ‘owner’, namely, an *equivocating* name. Equivocating (same-name) certificates contain exactly the same name as a legitimate domain name, but certified by an attacker - and, obviously, containing a public key chosen by the attacker. Equivocating certificates can be used to circumvent many important security mechanisms, including *Same-Origin-Policy (SOP)*, *blacklists*, *whitelists* and other *access-control* mechanisms.

Misleading (impersonating) certificates and domain-names. Many attacks focus on *misleading the user*, rather than misleading an automated security mechanism. These attacks take advantage of the fact that humans do not follow a precise algorithm for their trust decisions, in contrast with automated security mechanisms. The attacker’s goal is still, usually, impersonation; there is currently no mechanism that prevents rogue entities from obtaining domain names and certificates for non-impersonating nefarious purposes, such as scams - such mechanism is probably desirable, but seems very hard to establish. Hence, we focus on impersonating certificates. Notice that equivocating certificates can obviously be used for impersonating the subject, such as for phishing emails and spoofed (fake) websites; however, when the goal is to trick a human user, there are other ways which prove almost as effective, mainly:

Combo names ('*combosquatting*'): names which combine a trademark or a name associated with a legitimate, trusted entity, with another term which seem to either ‘make sense’ or simply to ‘appear meaningless/technical’. Combo names may be the most effective form of misleading names. Example: to impersonate as the website *bank.com*, use *accounts-bank.com* or *bank.accts.com*.

Domain-name hacking: the attacker uses a different domain name, which the attacker can register and control, but which users, usually not even aware of the structure of domain names, will not distinguish from a trusted domain name. Examples: to impersonate as the site *accounts.bank.com*, use *bank.accounts.com*, *accounts-bank.com* or *accounts.bank.co*. The last example also uses the human tendency to ignore the end and other minor deviations in text (our ‘built-in error correction’).

Homographic attack: the attacker uses names which appear *visually* to be the exact names of a legitimate, trusted entity, although, they actually just exploit visual similarities between different characters, typically using different fonts. A simple example are the names *paypal.com* and *paypaI.com* - in many fonts, the lower-case ‘L’ and capital ‘T’ are hard to distinguish; a more common method is to use fonts from different alphabets, e.g., some Cyrillic letters are visually indistinguishable from other Latin letters (e.g., P).

Typosquatting: These domain names exploit typical typos, such as due to typing and/or spelling errors, such as *banc.com*, *baank.com* and *banl.com*.

The high-level goal of PKI is to protect the relying parties from such rogue certificates, as well as CAs who issue rogue certificates intentionally or due to negligence. In the next subsection, we try to turn this high-level goal into more precise requirements. The concerns about misleading certificates and domain names are some of the many challenges which a designer faces, when trying to protect systems involving human users; we look a bit deeper into this important topic in Chapter 9.

8.1.2 Security goals of PKI schemes.

X.509 security goals. The basic, high-level goal of a public key infrastructure (PKI) is to allow relying parties to ensure that it uses a *valid public key* for its specific needs and application.

The mere fact that X.509 certificates are signed by the CA may seem to ensure this goal; however, there are two caveats. First, X.509 allows certification not only by *root CAs*, trusted directly by the relying party, but also by *intermediate CAs*, based on a precise policy of the root CA. Second, relying parties should not be fooled into relying on a revoked certificate. These two caveats imply two corresponding security requirements:

Accountability: assume a relying party validated a certificate c , optionally using some ‘additional data’ D (typically, additional certificates). Yet, assume that the entity identified as the subject in c , denies ‘owning’ the certified public key. Then we can identify an *accountable CA*, CA_A , which has signed some certificate c_A , whose subject denies ‘owning’ the public key certified in c_A . If c was issued by a root CA than D is unnecessary, CA_A is the root CA and $c_A = c$. See X.509’s certificate-path mechanisms in Section 8.3.

Revocation: assume that at time t , a CA revokes a certificate c (that it previously issued). Then after some bounded delay Δ , all relying parties will not consider the certificate as valid.

Post-X.509 security goals. There are several ‘post-X.509’ PKI designs which aim to address additional requirements, including:

Transparency: the set of all issued certificates is transparent, i.e., publicly known. See Section 8.6.

Revocation-status transparency: the *revocation status* of a certificate is publicly known, i.e. it is known whether a particular certificate was revoked.

Equivocation-prevention: there cannot exist two valid yet *equivocating* certificates, i.e., different identity certificates for the same identifier (e.g., domain name).

Equivocation detection any pair of equivocating certificates would be detected within bounded time after the second one is issued.

Relying party privacy: the PKI mechanism does not expose which certificates are validated by a given relying party. One case where this does *not* hold is when using the OCSP protocol (Section 8.4).

8.1.3 The Web PKI

In Chapter 7 we discussed the use of certificates by the SSL/TLS protocol, used to secure web traffic and other applications. The SSL/TLS client (often, the browser) receives a certificate $(pk, ATTR, \sigma)$ authenticating the server’s public key pk , and binding it to the server’s domain, e.g., *bob.com*, which is specified as one of the attributes in $ATTR$. The certificate contains a signature σ ; for the certificate to be valid, σ must be a signature over $(pk, ATTR)$, which validates correctly using the public validation key of some *trusted certificate authority* (CA).

In web security applications, each browser maintains and/or uses¹ a list of trusted *root certificate authorities* (*root CAs*). These *root CAs* can also certify

¹Often, browsers use a list of trusted root CAs maintained by the operating system, possibly combining it with browser-maintained list.

additional CAs, referred to as *intermediate CAs*; we explain the process later on. To support SSL/TLS, a web-server for a domain, e.g., *bob.com*, needs a certificate for that domain, signed by a root or intermediate CA, and, of course, to know and use the corresponding private key.

Note that SSL/TLS also supports (optional) *client authentication*; however, SSL/TLS client authentication requires a *client certificate*; only few traditional clients, such as browsers, have these client certificates, but their use is more common for IoT devices. Similarly, client certificates are required for end-to-end secure email services, e.g., using *S/MIME* [268]; again, only a tiny fraction of the users went through the process of obtaining a client certificate, and as a result, these secure email services are not widely used. The difficulties of obtaining client certificates are probably one reason for the fact that most secure messaging applications rely on authentication by the provider, and sometimes also by the peer user, but not on client certificates.

For further discussion, focusing on weaknesses of the current Web PKI and some solutions, see subsection 8.5.1.

8.2 The X.509 PKI

In this section, we discuss the basic notions of the X.509 PKI standard, which was developed as part of the X.500 global directory standard. X.509 is the most widely deployed PKI specification, and also includes some of the more advanced PKI concepts which we cover in the following sections.

8.2.1 The X.500 Global Directory Standard

X.500 [81] is an ambitious, extensive set of standards of the *International Telecommunication Union (ITU)*, a United Nations agency whose role is to facilitate international connectivity in communications networks. The goal of X.500 is to facilitate the interconnection of *directory services* provided by different organizations and systems. The first version of X.500 was published as early as 1988, and numerous extensions and updates were published over the years.

The basic idea of X.500 is to provide a *trusted, unified* and ideally *global* directory, by combining the data and services of its multiple component directories. Such a unified directory would be operated by cooperation between trustworthy providers, such as telecommunication companies. Significant aspects of X.500, such as the distinguished names, are deployed by LDAP and other directory services [174], although, these are far from the vision of a global directory. Among the possible reasons for that is the high complexity of the X.500 design, concerns that X.500 interoperability may cause exposure of sensitive information, and lack of sufficient trust among different directory providers.

However, some concepts from X.500 live on; we already mentioned LDAP as one example. More relevantly to our subject, the X.500 recommendation contributed extensively to the development of PKI schemes. The X.500 designers

observed that an interoperable directory should bind *standard identifiers* to *standard attributes*.

One important set of attributes defines the *public key(s)* of each entity. The entity's *public encryption key* allows relying parties to *encrypt* messages so that only the intended recipient may decrypt them. Similarly, the entity's *public validation key* allows relying parties to *validate* statements *signed* by the entity.

We next discuss the main form of *standard identifier* defined in X.500: the *distinguished name*.

8.2.2 The X.500 Distinguished Name

The design of X.500 was extensively informed by the experience of telecommunication companies at the time, which included provision of directory services to phone users. Phone directory services are mostly based on looking up the person's *common name*; the common name has the obvious advantage of being a *meaningful identifier* - we usually know the common name of a person when we ask the directory for that person's information. Phone directories would normally also allow specification of the relevant *area*, e.g., in form of *locality*; by limiting search to specific areas or localities, the directory services can be *decentralized*.

However, obviously, a common name is not a unique identifier - in fact, some common names are quite common, if you excuse the pun. In classical phone directories, this is addressed by returning a set of results containing all relevant entries, along with the relevant common name and other attributes (e.g., location).

The X.500 designers decided that in order to allow efficient use of large, global directories, returning multiple results is not a viable option. Instead, they decided to use a more refined identifier, with multiple keywords - where the common name will simply be one of these keywords. This identifier is the X.500 *Distinguished Name (DN)*. The distinguished name was designed to satisfy the following three main *goals for identifiers*:

Meaningful: identifiers should be meaningful and recognizable by humans.

This makes it easier to memorize the identifier, as well as to link it with off-net identifier, with potential legal and reputation implications.

Unique: identifiers should be unique, i.e., different subjects should have different identifiers, allowing each identifier to be mapped to a specific subject.

Decentralized management: multiple, 'distributed' issuers, can issue identifiers, without restrictions, i.e., any issuer is allowed to issue any identifier.

The uniqueness requirement is an obvious challenge, as common names are obviously not unique. To facilitate unique DNs for people sharing the same common name, X.500 distinguished names consist of a *sequence* of *several keyword-value pairs*. The inclusion of multiple keywords - also referred to as

C	Country
L	Locality
O	Organization name
OU	Organization unit
CN	common name

Table 8.1: Standard keywords/attributes in X.500 Distinguished Names

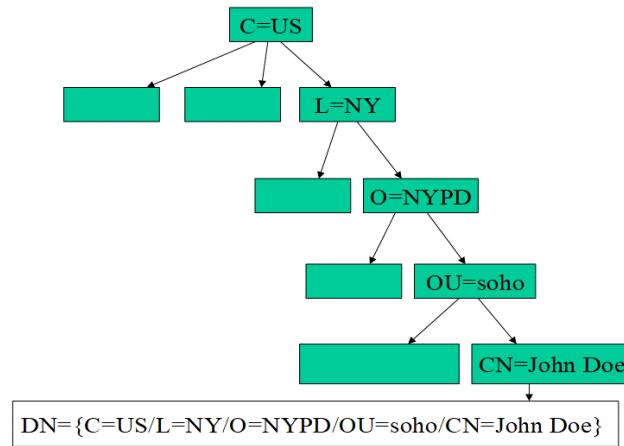


Figure 8.2: Example of the X.500 (and X.509) Distinguished Name (DN) Hierarchy.

attributes - helps to ensure unique identification, when combined with the common name. Typical, standard keywords are shown in Table 8.1; however, a directory is free to use any keyword it desires.

To satisfy the ‘meaningful’ goal, identifiers should have readable representations. RFC 1779 [192] specifies a popular string representation for distinguished names, where keyword-value pairs are separated by the equal sign, and different pairs are separated by comma or semicolon, optionally also with spaces. Other representations are possible too, e.g., Figure 8.2 includes encoding of a DN using slash for separation.

Let us give two simple examples of different legitimate interpretations (and implementations) of the RFC 1779 representation:

1. As illustrated in Figure 8.2, the distinguished name (DN) for a police officer named John Doe in the Soho precinct of the NYPD may be defined as: $C=US/L=NY/O=NYPD/OU=soho/CN=John\ Doe$.
2. The distinguished name (DN) for an IBM UK employee with the name Julian Jones may be written as: $CN=Julian\ Jones, O=IBM, C=GB$. Read below on the author’s experience with this (realistic) DN.

Note that the two examples use different *order* of keywords, with the most specific term being on the right in the first and on the left in the second; this kind of ambiguity is a source of implementation bugs, as indeed happened for few implementations; it may also cause vulnerabilities.

Note that keyword-value pairs comprising an X.500 distinguished name are specified in a sequence, i.e., as an *ordered* list. This allows the distinguished names to be organized as a hierarchy, using the sequence of keywords as the nodes, as illustrated in Figure 8.2. By assigning a specific, single entity to assign identifiers in a sub-tree of the X.500 DN hierarchy, this entity can ensure *uniqueness* by never allocating the same identifier (DN) to two different subjects, e.g., the Soho precinct of the NYPD may maintain its own sub-directory. This also allows queries over the entire set of distinguished names that begins with a particular prefix of keyword-value pairs.

However, this implies that X.500 distinguished names cannot be issued in an entirely decentralized manner - some control and coordination on the allocation of identifiers is required. Furthermore, there are also some caveats with respect to the other goals - unique and meaningful identifiers.

Let us first consider the goal of meaningful identifiers. The use of subdivisions such as ‘organization unit (OU)’ may help to reduce the likelihood of two persons with the same common name in the same ‘bin’, this possibility still exists. As a result, administrators may have to enforce uniqueness by ‘modifying’ the common name. For example, if there are multiple IBM UK employees with the name Julian Jones, one of them may be assigned the DN:

$CN=Julian\ Jones2, O=IBM, C=GB$

This results in *less meaningful distinguished names*; e.g., it is easy to confuse between the DNs of the two employees. For example, the author has sent to $CN=Julian\ Jones, O=IBM, C=GB$ messages intended for $CN=Julian\ Jones2, O=IBM, C=GB$, when using an email system that used distinguished names as email addresses. Luckily, both Julians were understanding of the mistake.

Another cause of mistakes and ambiguity is the fact that there are no rules governing the *order* of the keywords, i.e., the structure of the hierarchy, as is evident from the two examples we presented. In particular, some multinational organizations may use the country as the top level category, as in $CN=Julian\ Jones, O=IBM, C=GB$, while others may view the organization itself as the top-level category, as in $CN=Julian\ Jones, C=GB, O=IBM$. These two distinguished names are *different*; this distinction may not be obvious to a non-expert, further reducing from the goal of ‘meaningful’ names.

There are also cases where uniqueness is not guaranteed. Some namespaces are shared by design, and cannot be segregated with a single authority assigning identifiers in each segment. For example, consider Internet domain names; multiple registrars are authorized to assign names in several top-level domains such as `com` and `org`. There is a coordination process between registrars, but if not followed correctly, conflicts may occur.

This problem is more severe with respect to public key certificates for Internet domain names, which can be issued by multiple Certificate Authorities; any

faulty authority may issue a certificate to an entity who does not rightfully own the certified domain name. Such incidents occurred - often due to intentional attack; e.g., see Table 8.5. This is a major concern for Web PKI as well as PKI in general, and we discuss it further later in this chapter.

We conclude that X.500 distinguished names are not perfectly *meaningful* and definitely not *decentralized*; furthermore, sometimes, distinguished names may even not perfectly ensure *uniqueness*. Indeed, there seem to be an inherent challenge in satisfying all three goals, although achieving any two of these three properties is definitely feasible - a classical trilemma scenario.

The identifiers trilemma. We argued that X.500 distinguished names may fail to ensure each of the three goals defined above - *uniqueness*, *meaningfulness* and *decentralized management*. In contrast, several other identifiers ensure pairs of these three properties:

Common names are meaningful - and decentralized, as any person can decide on the name. However, they are definitely not unique.

Public keys and random identifiers are decentralized and (except for very rare collisions) unique. However, obviously, public keys are not directly meaningful to humans.

Email addresses are unique and meaningful. However, they are not decentralized, since each issuer can only assign identifiers (email addresses) in its own domain.

This begs the question: is there a scheme which will ensure identifiers which fully satisfy all three properties, i.e., would be unique, meaningful and managed and issued in a decentralized way? It seems that this may be hard or impossible, i.e., it may be possible to only fully ensure two of these three goals, but not all three. We refer to this challenge as the *Identifiers Trilemma*², and illustrate it in Figure 8.3.

Additional concerns regarding X.500 Distinguished Names. We conclude our discussion of X.500 distinguished names, by discussing few additional concerns.

Privacy. The inclusion of multiple categorizing fields in X.500 DNs, may expose information in an unnecessary, and sometimes undesired, manner. For example, employees may not always want to expose their location or organizational unit.

Flexibility. People may change locations, organization units and more; with X.500 DNs, this may result in ‘incorrect’ DN, or require change of the DN - both undesirable.

²This challenge is also referred to as *Zooko’s triangle*; however, Zooko has apparently referred to a different trilemma, albeit also related to identifiers. Specifically, Zooko considered the challenge of identifiers which will be distributed, meaningful for humans, and also *self-certifying*, allowing recipients to locally confirm the mapping from name to value.

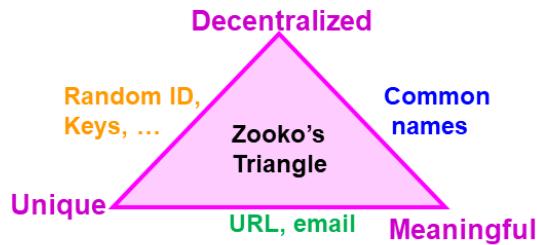


Figure 8.3: The Identifiers Trilemma: the challenge of co-ensuring unique, decentralized and meaningful identifiers.

Usability. X.500 DNs are designed to be *meaningful*, i.e., users can easily *understand* the different keywords and values. However, sometimes this may not suffice to ensure *usability*. In particular, consider two of the most important applications for public key cryptography and certificates: secure web-browsing and secure email/messaging.

Secure web-browsing: users, as well as hyperlinks, specify the desired website using an Internet domain name, and not a distinguished name. Hence, the relevant identifier for the website is that domain name - provided by the user or in the hyperlink. This requires mapping from the domain name to the distinguished name. A better solution is for the certificate to directly include the domain name; this is supported by the *SubjectAltName* extension, defined by PKIX, see subsection 8.2.6.

Secure email/messaging: users also do not use distinguished names to identify peers with whom they communicate using email and instant messaging applications. Instead, they use email addresses - or application-specific identification. This problem may not be as meaningful, since most end users do not have a public key certificate at all; and, again, PKIX allows certificates to directly specify email address.

8.2.3 X.509 Public Key Certificates

The X.500 standard included a dedicated sub-standard, X.509, which defined *authentication mechanisms*, allowing entities to authenticate themselves to the directory. X.509 defined multiple authentication mechanisms, e.g., the use of password based authentication. However, one of these authentication methods became a very important, widely used standard: the *X.509 public key certificate*.

Originally, the main goal of the X.509 authentication was to allow each entity to maintain its own record with the directory, e.g., to change address. However, it was soon realized that public key certificates allow many more applications,

Signed fields	Version
	Certificate serial number
	Signature Algorithm Object Identifier (OID)
	Issuer Distinguished Name (DN)
	Validity period
	Subject (user) Distinguished Name (DN)
	Subject public key information
	Public key Value Algorithm Obj. ID (OID)
Signature on the above fields	

Figure 8.4: X.509 version 1 certificate. Note the fields added in later versions, mainly version 3 of X.509 (Figure 8.5), most notably the extensions field.

since they allow recipients to authenticate the public key of a party without requiring any prior communication. As a result, X.509 certificates became a widely deployed standard, which is used for SSL/TLS, code-signing, secure email (S/MIME), IP-sec and more. All this use is in spite of complaints about the complexity of the X.509 specifications and encoding formats - obviously, the wide use is also one reason for the numerous complaints. For details of the encoding, see [187, 309]; see also [253].

The definition of the X.509 certificates did not change too much from the first version of X.509; the contents (fields) of that first version of X.509 are shown in Figure 8.4. These fields are, by their order in the certificate:

Version: the version of the X.509 certificate and protocol.

Certificate serial number: a serial number of the certificate, unique among all of the certificates issued by this CA. PKIX [86] specifies that the serial number should be a positive integer of up to 20 bytes, i.e., up to 159 bits. The best practice is to select the serial number *randomly*, not sequentially. The motivation are attacks [301, 302] that manipulate a CA into issuing a certificate whose hash collides with the contents of a different certificate, when using predictable sequence numbers, together with a hash-function which has the chosen-prefix collisions vulnerability, such as MD5 or SHA-1 (subsection 3.3.1).

Signature-process Object Identifier (OID): this is an identifier of the process used for signing the certificate, typically using the *Hash-then-Sign* paradigm. This identifier specifies both the underling public key signature algorithm, e.g., RSA, as well as the hash algorithm, e.g., SHA-256. The algorithm may be written as a string for readability, and standard string terms are used for widely used methods, e.g., sha256WithRSAEncryption;

notice the use of the term ‘RSA Encryption’ when referring to RSA signatures - a common misnomer. In the certificate itself, the algorithm is typically specified using the *Object-Identifier (OID)* standard; see Note 8.1.

Issuer Distinguished Name: the distinguished name of the certificate authority which issued, and signed, the certificate.

Validity period: the period of time during which the certificate is to be considered valid.

Subject Distinguished Name: the distinguished name of the *subject* of the certificate, i.e., the entity to whom the certificate was issued. This entity is expected to know the private key corresponding to the certified public key.

Subject public key information: this field contains the public key of the subject, and an *object identifier (OID)*, see Note 8.1) that identifies the algorithm with which the key is used, including key-length, e.g., RSA/2048. The allowed *usage* of the certified public key - e.g., to encrypt messages sent to the subject, or to validate signatures by the subject is specified in the *KeyUsage* extension (subsection 8.2.7), not in subject public key information field.

Signature: finally, this field contains the result of the application of the signature algorithm (identified by the signature-process OID field above), to all of the other fields in the certificate, using the private signing key of the issuer (certificate authority). The sequence of all these fields in the certificate, excluding the signature field itself, is referred to as the *to-be-signed* fields; see Figure 8.4 and Figure 8.5. This allows the *relying party* to validate the authenticity of the fields in the certificate, e.g., the validity period, the subject distinguished name, and the subject public key.

Exercise 8.1. *Provide a security motivation for the fact that the signature process is specified as one of the (signed) fields within the certificate. Do this by constructing two ‘artificial’ CRHFs, h^A and h^B ; to construct h^A and h^B , you may use a given CRHF h . Your constructions should allow you to show that it could be insecure to use certificates where the signature process (incl. hashing) is not clearly identified as part of the signed fields. Specifically, design h^A , h^B to show how an attacker may ask a CA to sign a certificate for one name, say Attacker, and then use the resulting signature over the certificate to forge a certificate for a different name, say Victim.*

X.509 Certificates: Versions 2 and 3. Following X.509 version 1, the X.509 certificates were extended by few additional fields; see Figure 8.5.

Version 2 of X.509 added two fields, both of them for unique identifiers - one for the subject and one for the issuer (CA). These fields were defined to

Note 8.1: Object identifiers (OIDs)

The joint An ITU and ISO ASN.1 standard [79,107] defines the concept of an object identifier (OID) as a unique identifier for arbitrary objects. Object identifiers are specified as a sequence of numbers, e.g., 1.16.180.1.45.34, separated by dots (as shown) or spaces. OID numbers are assigned hierarchically to organizations and to ‘individual objects’; when an organization is assigned a number, e.g., 1.16, it may assign OIDs whose prefix is 1.16 to other organizations or directly to objects, e.g., 1.16.180.1.45.34. The top level numbers are either zero (0), allocated to ITU, 1, allocated to ISO, or 2, allocated jointly to ISO and ITU. RFC 3279 [22] defines OIDs for many cryptographic algorithms and processes used in Internet protocols, e.g., RSA, DSA and elliptic-curve signature algorithms; when specifying a signature process, the OID normally also specifies both the underlying public key signature algorithm and key length, e.g., RSA/2048, and the hashing function, e.g., SHA-256, used to apply the ‘Hash-then-Sign’ process. X.509 uses OIDs to identify signature algorithms and other types of objects, e.g., extensions and issuer-policies. The use of OIDs allows identification of the specific type of each object, which helps interoperability between different implementations.

Signed fields	Version
	Certificate serial number
	Signature Algorithm Object Identifier (OID)
	Issuer Distinguished Name (DN)
	Validity period
	Subject (user) Distinguished Name (DN)
	Subject public key information
	Public key Value
	Algorithm Obj. ID (OID)
	Issuer unique identifier (from version 2)
	Subject unique identifier (from version 2)
	Extensions (from version 3)
	Signature on the above fields

Figure 8.5: X.509 version 3 certificate. Version 2 is identical, except for not having the *extensions* field; version 1 also does not have the two ‘unique identifier’ fields (Figure 8.4).

ensure uniqueness, in situations where the distinguished name may fail to ensure uniqueness, as discussed in subsection 8.2.2. However, these unique identifier fields are not in wide use, as they are entirely unrelated to the meaningful identifiers used in typical applications.

Version 3 of X.509 (*X.509v3*) is the one in practical use; the main reason for its wide success is that it dramatically increased the expressiveness of X.509 certificates. As can be seen in Figure 8.5, this dramatic improvement is due to just one new field added in version 3: the general-purpose *extensions* field. The extensions field provides extensive flexibility and expressiveness to certificates, and facilitates many applications and use cases; this field is typically much longer than all other fields combined. The X.509v3 extensions mechanism is the subject of the next subsection.

8.2.4 The X.509v3 Extensions Mechanism

As shown in Figure 8.5, X.509 certificates, from version 3, include a field that can contain one, or more, extensions. We discuss some specific, important extensions in the following subsections. But first, let us discuss the extensions mechanism itself, since this mechanism has a rather clever design, which cleverly balances between the need to allow extendibility, and the concern of using a certificate incorrectly (due to ignoring or incorrectly handling an extension).

Each extension has the following three components:

Extension identifier: specifies the type of the extension. The extension identifier is specified using an object identifier (OID), to facilitate interoperability. The following subsections discuss some important extensions, e.g., *key usage* and *name constraint*.

Extension value: this is an arbitrary string which provides the value of the extension. For example, a possible value for the key-usage extension would indicate that the certified key is to be used as a public encryption key, while a possible value for the name constraint extension may be *Permit C=GB*, allowing the subject of the certificate to issue its own certificates, but only with the value ‘GB’ (Great Britain) to their ‘C’ (country) keyword.

Criticality indicator: this is a binary flag, i.e., an extension can be marked as critical or as non-critical. The value of the criticality indicator flag in an extension instructs relying parties how to handle the certificate if the relying party is not familiar with this type of extension, as indicated by the extension identifier. A relying party *should not use* a certificate which includes an extension marked as critical, if the relying party is not familiar with this type of extension. Relying parties can use certificates even if it contains an extension of a type not known to the relying party, ignoring that extension, if the extension is marked as *non-critical*. When the relying party is familiar with the type of an extension, the value of the criticality indicator is not applicable.

The *criticality indicator* flag is a simple mechanism - but very valuable, by allowing both *critical extensions* and *non-critical extensions*. The flexibility offered by the ‘criticality indicator’ makes the X.509 extensions mechanism very versatile; it is a pity that this idea has not been adopted by other extension mechanisms. For example, TLS client and servers simply ignore unknown TLS extensions, i.e., treat them as non-critical, as discussed in Chapter 7. It would have been useful if TLS allowed also definition of critical extensions, i.e., instructing a TLS peer to refuse connection if the peer is sending a critical but unknown TLS extension. This can be achieved quite easily; see next exercise.

Exercise 8.2. *Design how TLS may be extended to support critical extensions. Could you achieve this using the existing TLS extensions mechanism?*

X.509, as well as PKIX and other X.509 profiles, define some extensions to be always marked critical, others to be always marked non-critical, and others to be marked differently depending on needs. We next present examples of each of these three types of extensions, focusing on standard extensions.

Example 8.1. *The TLS feature X.509 extension: defined to be used as a non-critical extension.*

An X.509 extension called *TLS feature* is defined in RFC 7633 [155]. This TLS feature extension is used in TLS server certificates, to indicate that the server supports a specific *TLS extension* (see subsection 7.4.3). The name chosen for this X.509 extension is *TLS feature*, rather than *TLS extension*, to make it clearer that the TLS feature is an X.509 extension, and only refers to the support for a specific TLS extension; unfortunately, confusion is still natural.

The *TLS feature* X.509 extension allows the server to indicate to the client that the server supports certain important TLS extensions (‘features’). Some TLS clients may not support the TLS feature X.509 extension, so if this extension would be marked critical, these clients would reject the certificate, and the connection would fail. Hence, the TLS feature X.509 extension should be marked as non-critical. Note that the TLS feature extension isn’t one of the standard extensions defined in either X.509 or PKIX; it was developed later, specifically, to allow a certificate to mark that the server always uses the *must-staple* TLS extension, see subsection 8.4.3.

Example 8.2. *The extended key usage extension: can be either critical or non-critical.*

The *extended key usage* extension allows the issuer to define allowed usage for the public key certified, which is in addition to or in place of the usage specified in the *key usage* extension. In some scenarios, the ‘extended key usage’ should be critical, e.g., to prevent incorrect usage based on the key usage extension, by clients not supporting extended key usage. In other scenarios, the extended key usage extension should be non-critical, e.g., when allowing some additional usage over that specified already in the key usage extension.

Example 8.3. *The key usage extension: always critical in PKIX, and a motivating attack.*

PKIX specifies that the key usage extension must be marked critical, while X.509 allows the key usage extension to be marked as either critical or ‘not-critical’. Let us first give a contrived example of a possible attack exploiting a certificate where key-usage was not marked as critical, causing a relying party who does not understand this extension, to make a critical security mistake. Assume that the parties (key-owner, i.e., certificate subject, and relying party) use ‘textbook RSA’ encryption, i.e., encrypt plaintext m_E by computing $c = m_E^e \bmod n$; and ‘textbook RSA’ signing, i.e., sign message m_S by outputting $\sigma = h(m_S)^d \bmod n$, i.e., ‘decrypting’ the hash of the message. Furthermore, assume the key-owner uses its decryption key to authenticate that it is active at a given time, by decrypting an arbitrary challenge ciphertext sent to it; this requires only a relatively weak form of ciphertext-attack resistance, where the attacker must ask for the decryption *before* seeing the challenge ciphertext it must decrypt, often referred to as *IND-CCA1 secure* and assumed for textbook RSA. A key-owner using this mechanism must use its key only for decrypting these challenges; assume it receives a certificate C_E for its encryption key e , with the key-usage extension correctly marking this as an encryption key, but *not* marked as critical.

An attacker may abuse this, together with the fact that key usage is not understood by some relying parties, to mislead these relying-parties into thinking that the key-owner *signed* some attacker-chosen-message m_A , as follows. The attacker computes $c_A = h(m_A)$ and sends it to the key-owner, as if it is a standard challenge ciphertext to be decrypted. The key-owner therefore decrypts c_A and outputs the decryption, $c_A^d \bmod n = h(m_A)^d \bmod n$, which we denote by σ_A , i.e., $\sigma_A \equiv h(m_A)^d \bmod n$. Now the attacker sends the pair (m_A, σ_A) , along with the certificate C_E , to the relying party, claiming m_A was signed by the key-owner with signature σ_A . Since the relying party is not familiar with the key-usage extension, and it was not marked critical in the key-owner’s certificate C_E , then the relying-party would validate (m_A, σ_A) , which would validate correctly, and thereby incorrectly consider m_A as validly-signed by the key-owner.

Let us also point out a more practical attack on a TLS 1.3 client that does not correctly implement the key usage extension. If the TLS server also has runs any older version of TLS/SSL that is vulnerable to some variant of the Bleichenbacher attack, then the attacker may be able to forge an RSA signature using the private (decryption) key of the old, vulnerable version. If the client ignores the key usage extension and uses the public key for verifying the signature, the attacker succeeds in a cross-protocol attack on TLS 1.3, even if the server has correctly separated between the TLS 1.3 signature-verification public key, and the public encryption key used by the old, vulnerable version.

8.2.5 Trust-Anchor Certificate Validation

Upon receiving a certificate, the relying party must decide whether it can rely on and use the certified public key, for a particular application. In this section, we focus on the case of certificates signed by a *trust anchor CA*, i.e., a CA trusted by the relying party. In this case, the relying party would apply the *certificate validation process*, using the public signature-validation key of the CA, $CA.v$, to determine if the given certificate is valid. If the certificate is not signed by a trust anchor, then the relying party should first perform the *certification path* validation process, to decide if to trust this certificate, based on additional certificates; we discuss this in Section 8.3.

Assume, therefore, that a relying party receives a certificate signed (issued) by a *trust anchor*, i.e., the relying party trusts the issuing CA, denoted I , and knows its public validation key $I.v$. To validate the certificate, the relying party uses $I.v$ and the contents of the certificate, as follows:

Issuer. The relying party verifies that the issuer I of the certificate, as identified by the *issuer distinguished name* field, is a trusted CA, i.e., a trust anchor (root-CA, for Web PKI).

Validity period. The relying party checks the validity period specified in the certificate. If the public key is used for encryption or to validate signatures on responses to challenges sent by the relying party, then the certificate should be valid at the relevant times, including at the current time. If the public key is used to validate signature generated at the past, then it should be valid at a time when these signatures already existed, possibly attested by supporting validation by trusted time-stamping services.

Subject. The relying party verifies that the subject, identified in the *subject* field using the distinguished name, is an entity that the relying party expected. For example, when the relying party is a browser and it receives a website certificate, then the relying party should confirm that the website identity (e.g., domain name) is the same as indicated in the ‘subject distinguished name’ field of the certificate.

Signature algorithms. The relying party confirms that it can apply and trust the validation algorithm of the signature scheme identified in the *signature algorithm OID* field of the certificate. If the certificate is signed using an unsupported algorithm, or an algorithm known or suspected to be insecure, validation fails.

Issuer and subject unique identifiers. From version 2, X.509 certificates also include fields for unique identifiers for the issuer and the subject, which the relying party should use to further confirm their identities. In PKIX, these identifiers are usually not used, and PKIX does not require their validation. This is probably since in PKIX, the issuer and subject identifiers are typically in corresponding extensions.

Extensions. The relying party validates that it is familiar with any extension marked as critical; the existence of any unrecognized extension, marked as critical, would invalidate the entire certificate. Then, the relying party validates the existence and contents of any extension that its policy requires. To avoid incompatibilities, relying parties and CAs usually follow agreed-upon policies for the required and permitted extensions, often referred to as *PKI profile*, such as PKIX from the IETF [86] and profiles defined by the CA/Browser Forum [128].

Validate signature. The relying party next uses the trusted public validation key of the CA, $CA.v$, and the signature-validation process as specified in the certificate, to validate the signature over all the ‘to be signed’ fields in the certificate, i.e., all fields except the signature itself.

8.2.6 The SubjectAltName and the IssuerAltName Extensions

Both X.509 and PKIX define the standard *SubjectAltName (SAN)* and *IssuerAltName (IAN)* extensions, providing alternative identification mechanisms (names) to complement or replace the *Distinguished Name* mechanism, providing identification for identifying, respectively, the subject and the issuer. These alternative fields allow the use of other forms of names, identifiers and addresses for the subject and/or the issuer. Note that a certificate may contain multiple SANs.

The most important form of an alternative name is a *Domain Name System (DNS)* name, referred to as *dNSName*, e.g., *example.com*. These dNSNames are used by most Internet protocols, and are familiar to most users. Also allowed but rarely used alternative names, include email addresses, IP addresses, and URIs.

In fact, the use of alternate names is so common, that in many PKIX certificates, the subject and the issuer distinguished-name fields are left empty. Indeed, PKIX (RFC 5280) specifies that this must be done, when the Certificate Authority can only validate one (or more) of the alternative name forms, which is often the case in practice. PKIX specifies that in such cases, where the SubjectAltName extensions is the only identification and the subject distinguished name is empty, then the extension should be marked as *critical*, and otherwise, when there is a subject distinguished name, it should be marked as non-critical.

Note that PKIX (RFC 5280) specifies that the *Issuer Alternative Name* extension should always be marked as non-critical. In contrast, the X.509 standard specifies that both alternative-name extensions, may be flagged as either *critical* or *non-critical*.

Also, note that implementations of the Secure Socket Layer (SSL) and Transport Layer Security (TLS) protocols, often allow certificates to include *wildcard certificates*, which, instead of specifying a specific domain name, use the *wildcard notation* to specify a set of domain name. For TLS, this support is clearly defined in RFC 6125 [284]. Wildcard domain names are domain names

where some of the alphanumeric strings are replaced with the *wildcard character* ‘*’; there are often restrictions on the location of the wildcard character, e.g., it may be allowed only in the complete left-most label of a DNS domain name, as in `*.example.com`. Wildcard domain names are not addressed in PKIX (RFC5280) or X.509, and RFC 6125 mentions several security concerns regarding their use.

8.2.7 Standard key-usage and policy extensions

We next discuss another set of standard extensions, defined in the X.509 standard with further details in PKIX and other PKI profiles. These extensions deal with the *usage* of the certified key and with the *certificate policies* related to the issuing and the usage of the certificate. Some of the more important extensions, and their recommended usage as per the PKIX profile, include:

The authority key identifier extension. Provides an identifier for the issuer’s public key, allowing the relying party to identify which public validation key to use to validate the certificate, if the issuer has multiple public keys. It is always non-critical.

The subject key identifier extension. Provides an identifier for the certified subject’s public key, allowing the relying party to identify that key when necessary, e.g., when validating a signature signed by one of few signature keys of the subject - including signatures on (other) certificates. It is always non-critical.

The key usage extension. The *key usage* extension defines the allowed usages of the certified public key of the subject, including for signing, encryption and key exchange. The specification allows the use of same key for multiple purposes, e.g., encryption and validating signatures, however, this should not be used, as the use of the same key for such different purposes may be vulnerable - security would not follow from the pure security definitions for encryption and for signatures. An exception is when using schemes designed specifically to allow both applications, such as signcryption schemes. The PKIX standard ([86] requires this extension to be marked as critical; see subsection 8.2.4).

The extended key usage extension. The *extended key usage* extension allows definition of specific purposes for which the key is to be used, as supported by relying parties. The specification also allows the CA to indicate that other uses, as defined by the key-usage extension, are also allowed; otherwise, only the specified purposes are allowed. This extension may be marked as critical or not; see subsection 8.2.4.

The private key usage period extension. This extension is relevant only for certification of signature-validation public keys; it indicates the allowed period of use of the private key (to generate signatures). Always marked non-critical.

The certificate policies extension. This extension identifies one or more *certificate policies* which apply to the certificate; for brief discussion of certificate policies, see subsection 8.2.8. The extension identifies certificate policies using object identifiers (OID). In particular, the policy OID in the certificate policies extension, is the main mechanism to identify the type of validation of the legitimacy of the certificate, performed by the CA before it issued the certificate - *Domain Validation (DV)*, *Organization Validation (OV)* or *Extended Validation (EV)*. For more discussion on certificate policies and on the three types of validation, see subsection 8.2.8. The certificate policies extension may be marked as critical or as non-critical.

The policy mappings extension. This extension is used only in certificates issued to another CA, called *CA certificates*. It specifies that one of the issuer's certificate policies can be considered equivalent to a given (different) certificate policy used by the subject (certified) CA. This extension may be marked as critical or as non-critical.

Exercise 8.3. *Some of the extensions presented in this subsection should always be non-critical, while others may be marked either critical or non-critical. Justify each of these designations by appropriate examples.*

8.2.8 Certificate policy (CP) and Domain/Organization/Extended Validation

A *certificate policy (CP)* is a set of rules that indicate the applicability of the certificate to a particular use, such as indicating a particular community of relying parties that may rely on the certificate, and/or a class of relying party applications or security requirements, which may rely on the certificate. Certificate policies inform relying parties of the level of confidence they may have in the validity of the bindings between the certified public key and the information in the certificates regarding the subject, including the subject identifiers. Namely, the Certificate Policy provides information which may assist the relying party to decide whether or not to trust a certificate for a particular purpose. The certificate policy may also be viewed as a legally-meaningful document, which may define, and often limit, the liability and obligations of the issuer (CA) for potential inaccuracies in the certificate, and define statutes to which the CA, subject, and relying parties should conform; however, these legal aspects are beyond our scope.

Standard certificate policies and types of validations: DV, OV and EV. The certificate policies extension is often used to identify a *standard policy*; the policy is specified by the *policy OID* field. Such standard policies are defined by the CA-Brower Forum (CABF), and specific policies are identified in [128]; see Table 8.2. Standard policies often identify the type of validation performed by the CA before issuing the certificate.

Name	Usage	OID: 2.23.140...	Validation requirements	Price	Chrome	IE
TLS-DV	TLS	...1.2.1	Domain validated (confirmation email)	\$14		
TLS-OV	TLS	...1.2.2	Organization validation (registration, phone...)	\$30		
TLS-EV	TLS	...1.1	Extended validation (more verifications)	\$135		UBS AG [CH]
Code-Sign OV	Code signing	...1.3	Organization validation (registration, phone...)	\$82.5	N/A	N/A
Code-Sign EV	Code signing	...1.3	Extended validation (more verifications)	\$291	N/A	N/A

Table 8.2: Standard certificate policies.

Prices for a yearly certificate, from <https://www.thesslstore.com>, June 2021.

For the important case of website SSL/TLS certificates, three types of validation are defined: *Domain Validation (DV)* and *Organization Validation (OV)* and *Extended Validation (EV)* (in order of increasing validation).

Domain Validation and Organization Validation usually follow the ‘Baseline certificate requirements’ defined by the CA/Browser Forum in [126]. *Domain Validation (DV)* is a fully-automated - but not very secure - validation process. It involves sending a request to an address associated with the domain, and validating the response. The address may be an *IP address* or *email address* (sometimes referred to as *email validation*). Domain Validation is vulnerable to *MitM* attacks; it is also vulnerable to *off-path* attacks exploiting weaknesses of the *domain name system (DNS)* or of the *routing* infrastructure; see [161].

(OV) also requires validation of the organization which is certified, i.e., the *subject* of the certificate. Most CAs perform limited validation, typically involving validation that the organization is registered in the specified location, and that the request is validated by a phone call to a registered number.

Extended Validation involve additional validation requirements, usually following the ‘Extended-Validation certificate guidelines’ defined by the CA/Browser Forum in [127]. These include registration in official registries, physical address and more.

The type of validation *could* be used by relying parties to determine their use of the certificate; it is conceivable that fraudsters would be less likely to obtain EV or even OV certificates, due to their higher costs and stronger validation requirements. The risk of a rogue certificate depends on the validation requirements (Table 8.2); it is highest for domain-validated certificates, which are only validated by automated email to the address listed in the Whois records, a process vulnerable to off-path and routing attacks [54, 68].

However, the current popular web-browsers do not appear to treat certificates differently based on their validation method. Until around 2019, most major browsers displayed a visible indication in the location bar for EV certificates, e.g., as shown in Table 8.2 for the IE browser, based on research such as [163].

However, this was mostly abandoned in the recent years, and currently, most browsers make minimal use of the type of validation. Browsers display the same indicator for all TLS-protected websites (as shown for Chrome in Table 8.2), and the validation type can only be identified by users using the user-interface to look up the details of the certificate. The main justifications given [148] are that these indications were found ineffective, and interfere with the browser approach of presenting a warning against sites which are *not* protected by TLS.

Exercise 8.4. *Compare the user-interface indications of the certificate validation method of two browsers. Check the certificates for at least three websites, e.g., a bank, a newspaper and a browser download web-page.*

8.3 Intermediate-CAs and Certificate Path Validation

PKI schemes require the relying parties to *trust* the contents of the certificate, mainly, the binding between the public key and the identifier. In the simple case, the certificate is signed by a CA trusted directly by the relying parties, as in Figure 8.1. Such a CA, which is directly trusted by a relying party, is called a *trust anchor* or *root CA* of that relying party.

Direct trust in one or more trust-anchor (directly trusted) CAs might suffice for small, simple PKI systems. However, many PKI systems are more complex. For example, browsers typically directly trust dozens of trust anchor CAs, referred to in browsers as *root CAs*, and also browsers also *indirectly trust* certificates signed by other CAs, referred to as *intermediate CA*; an intermediate CA must be certified by root CA, or by a properly-certified, indirectly-trusted intermediate CA.

Relying parties and PKIs may apply different conditions for determining which certificates (and CAs) to trust. For example, in the PGP *Web-of-trust* PKI [134], every party can certify other parties. One party, say Bob, may decide to indirectly trust another party, say Alice, if Alice is properly certified by a ‘sufficient’ number of Bob’s trust anchors, or by a ‘sufficient’ number of parties which Bob trusts indirectly. The trust decision may also be based on *ratings* specified in certificates, indicating the amount of trust in a peer. Some designs may also allow ‘negative ratings’, i.e., one party recommending *not* to trust another party. The determination of whether to trust an entity based on a set of certificates - and/or other credentials and inputs - is referred to as the *trust establishment* or *trust management* problem, and studied extensively; see [60, 61, 166, 167, 219] and citations of and within these publications.

We focus on the simpler case, where *a single valid certification path* suffices to establish trust in a certificate; a *certificate path* is a series of certificates C_1, C_2, \dots where each of them is signed by the public key certified in the previous one, and the first one is signed by a root CA (trust anchor). Different relying parties may validate a certificate path differently, based on their different trust anchors and different policies for trusting certificates certified by an intermediate CA, using a certificate path. The same CA, say CA_A , may be a trust anchor for Alice, and an intermediate CA for Bob, who has a different trust anchor, say

CA_B . This is the mechanism deployed in most PKI systems and by most relying parties, and specified in X.509, and specifically in PKIX and Web-PKI. The validation of the certificate path is based on several *certificate path constraints extensions*, which we discuss in the following subsections.

8.3.1 The certificate path constraints extensions

In this subsection, we present the three *certificate path constraints* extensions that are defined in X.509 and PKIX: *basic constraints*, *name constraints* and *policy constraints*. These constraints are relevant only for certificates issued to a subject, e.g., *www.bob.com*, by some *intermediate CA (ICA)*, i.e., *ICA* is *not* directly trusted by the relying party (say Alice), i.e., it is not one of Alice's *trust anchors*.

Since an *intermediate CA (ICA)* is not a trust anchor for Alice (the relying party), then Alice would only trust certificates issued by the ICA if the ICA is 'properly certified' by some trust anchor CA; we use *TACA* to refer to a specific *Trust Anchor CA* which Alice trusts, and based on this trust, may or may not trust a given ICA.

In the simple case, illustrated in Figure 8.6, the relying party (Alice) receives two certificates: a certificate for the subject, e.g., the website *www.bob.com*, signed by some Intermediate CA, which we denote ICA; and a certificate for ICA, signed by the trust anchor CA, TACA. In this case, we will say that the subject, *www.bob.com*, has a *single-hop certification path* from TACA, since ICA is certified by the trust anchor TACA. In this case, therefore, the certification path consists of two certificates: C_{ICA} , the certificate issued by the trust anchor TACA to the intermediate CA ICA, and C_B , the certificate issued by the intermediate CA ICA to the subject (*www.bob.com*).

In more complex scenarios there are additional Intermediate CAs in the *certification path* from the trust anchor to the subject, i.e., the certification path is indirect, or in other words, contains multiple hops. For example, Figure 8.7 illustrates a scenario where the subject, *www.bob.com*, is certified via an indirect certification path with three hops, i.e., including three intermediate CAs: ICA1, ICA2 and ICA3. The subject *www.bob.com* is certified by ICA3, which is certified by ICA2, which is certified by ICA1, and only ICA1 is certified by a trust anchor CA, TACA. Hence, in this example, the certification path consists of *four* certificates: (1) C_{ICA1} , the certificate issued by the trust anchor TACA to the intermediate CA ICA1, (2) and (3), the two certificates C_{ICA2} and C_{ICA3} , issued by the intermediate CAs ICA1 and ICA2, respectively, to the intermediate CAs ICA2 and ICA3, respectively, and finally (4) C_B , the certificate issued by the intermediate CA ICA3 to the subject (*www.bob.com*).

We use the terms *subsequent certificates* to refer to the certificates in a certification path which were issued by intermediate CAs, and the terms *root certificate* or *trust-anchor certificate* to refer to the 'first' certificate on the path, i.e., the one issued by the trust-anchor CA. The second certificate along the path is certified by the intermediate CA certified by the trust anchor (in the trust-anchor certificate); and any following certificate along the path, say the

i th certificate along the path (for $i > 1$), is certified by the intermediate CA which was certified in the $(i - 1)$ th certificate in the path. The *length* of a certificate path is the number of intermediate CAs along it, which is one less than the number of certificates along the path.

Note that, somewhat contrary to their name, the certification path constraints cannot prevent or prohibit Intermediate CAs from signing certificates which do not comply with these constraints; the constraints only provide information for the *relying party*, say Alice, instructing Alice to trust a certificate signed by ICA_i , only if it conforms with the constraints specified in the certificates issued to the intermediate CAs.

8.3.2 The basic constraints extension

The *basic constraints* extension defines whether the subject of the certificate, say *example.com*, is allowed to be a CA itself, i.e., if *example.com* may also sign certificates (e.g., for other domains or for employees). More specifically, the extension defines two values: a Boolean flag denoted simply cA (with this non-standard capitalization), and an integer called $pathLenConstraint$ (again, with this capitalization).

The cA flag indicates if the subject (*example.com*) is ‘allowed’ to issue certificates, i.e., act as a CA; if $cA = TRUE$, then *example.com* may issue certificates, and if $cA = FALSE$, then it is not ‘allowed’ to issue certificates. Recall that this is really just a signal to the relying parties receiving certificates signed by *example.com*; also, this only restricts the use of the certificate that I issued to *example.com* for validation of certificates issued by *example.com*, it does not prevent or prohibit *example.com* from issuing certificates, which a relying party may still trust, either since it directly trusts *example.com* (i.e., it is a *trust anchor*), or since it receives also an additional certificate for *example.com* signed by a different trusted CA, and that certificate allows *example.com* to be a CA, e.g., by having the value *TRUE* to the cA flag in the basic constraints extension.

The value of the $pathLenConstraint$ is relevant only when there is a ‘path’ of *more* than one intermediate CA, between the Trust Anchor CA and the subject. For example, it is relevant only in Figure 8.7, and not in Figure 8.6.

For example, in both Figure 8.6 and Figure 8.7, the Trust Anchor CA (TACA) signs certificate C_{ICA1} , where it should specify the $ICA1$ is a trusted (intermediate) CA. Namely, it must set the cA flag in the basic-constraints extension of C_{ICA1} to *TRUE*. However, in Figure 8.7, $ICA1$ further certifies $ICA2$ which certifies $ICA3$ - and only $ICA3$ certifies the subject (*www.bob.com*). Therefore, for the relying party to ‘trust’ certificate C_B for the subject, signed by $ICA3$, it is required that C_{ICA1} will also contain the path-length ($pathLen$) parameter in the basic constraint extension, and this parameter must be at least 2 - allowing two more CAs till certification of the subject. Similarly, the certificate issued by $ICA1$ to $ICA2$ must contain the basic constraints extension, indicating cA as *TRUE*, as well as value of 1 at least for the $pathLen$ parameter. Unfortunately, currently, essentially all browsers do not enforce path-length

constraints on the root CAs. Root CAs sometimes do enforce path-length constraints on intermediate CAs, however, these are usually rather long, e.g., 3, leaving wide room for an end-entity to receive, by mistake, a certificate allowing it to issue certificates. Of course, in most cases, end-entity certificates will not allow issuing certificates, typically since their basic-constraints will indicate that they are not a CA.

Browsers usually enforce basic constraint, although, failures may happen, esp. since this kind of flaw - lack of validation - is not likely to be detected by normal user.

Exercise 8.5 (IE failure to validate basic constraint). *Old versions of the IE browser failed to validate the basic constraint field. Show a sequence diagram for an attack exploiting this vulnerability, allowing a MitM attacker to collect the user's password to trusted sites which authenticate the user using user-id and password, protected using SSL/TLS.*

Exercise 8.6. *Assume that TACA is concerned that subject-CAs may issue certificates to end-entities (e.g., websites) and neglect to include a basic constraint extension, to prevent the end entity from issuing certificates. Explain how TACA may achieve this, for the scenarios in Figure 8.6 and in Figure 8.7. Identify any remaining potential for such failure by one of the intermediate CAs in these figures.*

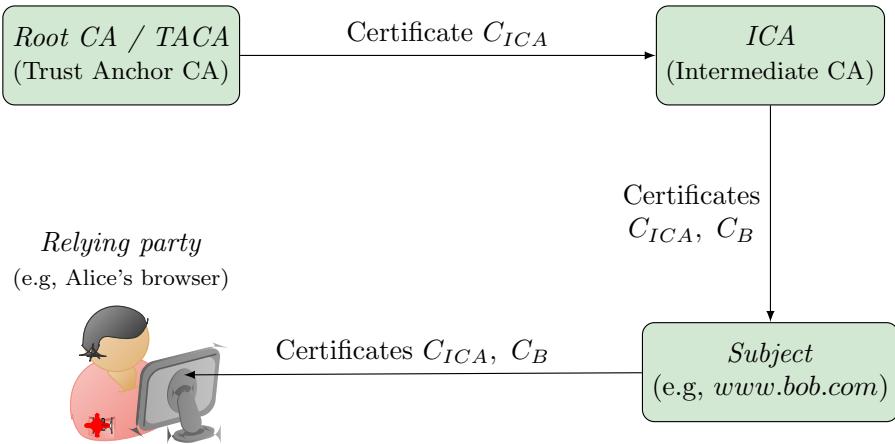
8.3.3 The name constraint extension

The *name constraint* extension is used in certificates issued to a subject CA, such as the intermediate CAs in Figure 8.6 and Figure 8.7. The name constraint extension restricts the set of subject-names to be certified by the subject CA, as well as by any subsequent CA. For example, in Figure 8.7, name constraint included in certificate C_{ICA1} issued by TACA to $ICA1$, would restrict certificates issued by $ICA1$, $ICA2$ and $ICA3$ ³.

The name constraint extension has two possible parameters, which we denote by the names⁴ *permit* (to define permitted name spaces) and *exclude* (to forbid name spaces, typically within the permitted name space). Focusing on the PKIX profile, both parameters are identifiers for names, usually a domain name; we focus on this case. When a domain name is specified, this is taken to include sub-domains, e.g., if a name constraints contain parameter *permit* (only) for domain name *com*, then this allows subdomains such as *google.com*, but not names in other top-level domains such as *x.org*. The *exclude* parameter takes precedence; i.e., if a certificate contains both *permit* for domain name, say *edu*, and *exclude* for subdomain *uconn.edu*, then this allows subsequent certificates

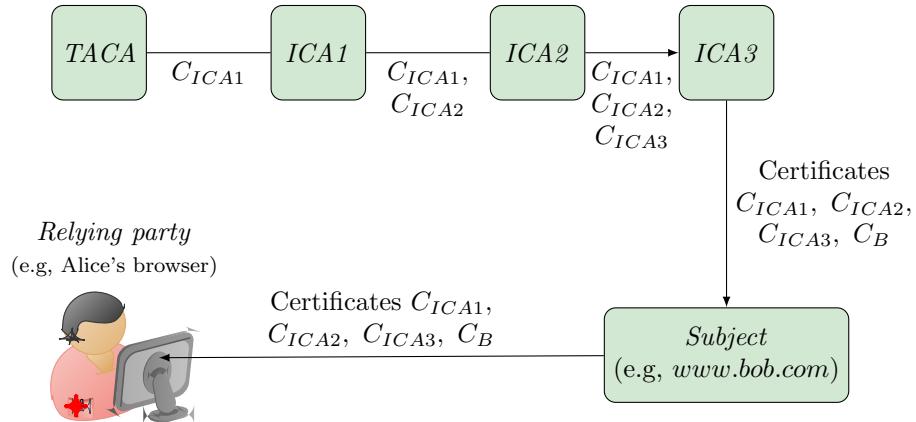
³The name constraints in C_{ICA1} would also restrict certificates issued by the subject (*www.bob.com*); we didn't list this above, since the subject's certificate, C_B , should prevent the subject from issuing certificates, using the *basic constraints* extension.

⁴The actual parameter names are *permittedSubtrees* and *excludedSubtrees*, which are a bit cumbersome.



	CICA constraints extensions					CB valid?
	Basic		Name	Policy		
	cA	pathLen	Permit	Exclude	Req. Policy	
1	No	(any)	(any)	(any)	(any)	No
2	Yes	(any)	bob.com	none or x.bob.com	none or > 1	Yes
3	Yes	(any)	cat.com	(any)	(any)	No
4	Yes	(any)	bob.com	www.bob.com	(any)	No
5	Yes	(any)	(any)	(any)	0	No
6	Yes	(any)	(any)	bob.com	(any)	No

Figure 8.6: A *single-hop* (length one) certificate-path, consisting of trust-anchor CA *TACA*, an intermediate CA *ICA*, and a subject (e.g., website *www.bob.com*). The table shows the impact of six examples of certificate path constraints extensions in certificate *CICA*, on the validity of certificate *CB* issued by *ICA*. In these examples, *CB* is for domain name *www.bob.com*, has no certificate policies extension and has basic constraints indicating *cA = No* (not a CA); and the value *(any)* in a field, indicates that the example holds for any value in this field. Each row is one example of the constraints in *CICA*. In example (row) 1, *CICA* does not have the *cA* flag set (true); namely, *CICA* does *not* indicate that *ICA* is a CA, and hence *CB* is invalid. In contrast, in example 2, certificate *CB* is valid, since the *cA* flag is *true*, the Name-constraints permit *bob.com* and does not exclude *www.bob.com*, and either there is no policy-constraint or its value is more than 1. See discussion in subsection 8.3.1.



	C_{ICA1} constraints extensions					C_B valid?
	Basic		Name		Policy	
	cA	pathLen	Permit	Exclude	Req. Policy	
1	Yes	< 2	(any)	(any)	(any)	No
2	Yes	none or ≥ 2	<i>bob.com</i>	none or <i>x.bob.com</i>	none or > 3	Yes
3	Yes	(any)	(any)	(any)	≤ 3	No
4	Yes	(any)	<i>cat.com</i>	(any)	(any)	No
5	Yes	(any)	(none)	<i>bob.com</i>	(any)	No

Figure 8.7: A *length 3* certificate-path, consisting of trust-anchor CA *TACA*, three intermediate CAs (*ICA1*, *ICA2*, *ICA3*), and a subject (e.g., website www.bob.com). The table shows the impact of the five example values of the certificate path constraints extensions (see subsection 8.3.1), in particular, of the *pathLen* (path length) parameter of the basic constraints extension. For the examples in the table, assume that none of the certificates has the certificate policies extension, and that the intermediate certificates C_{ICA1} , C_{ICA2} , C_{ICA3} all have the *cA* flag set in ‘Basic constraints’, and that C_{ICA2} , C_{ICA3} do not have any other constraints. For example, in row 1, C_B is invalid, since the *pathLen* field in the Basic-constraints extensions of C_{ICA1} is set to less than 2 (and the path from *ICA1* to *ICA3* is of length two). In contrast, in row 2, the *pathLen* constraint does not exist (or is satisfied), and the other constraints in C_{ICA1} are also set to allow the certificate path to be valid (compare to the examples in Figure 8.6).

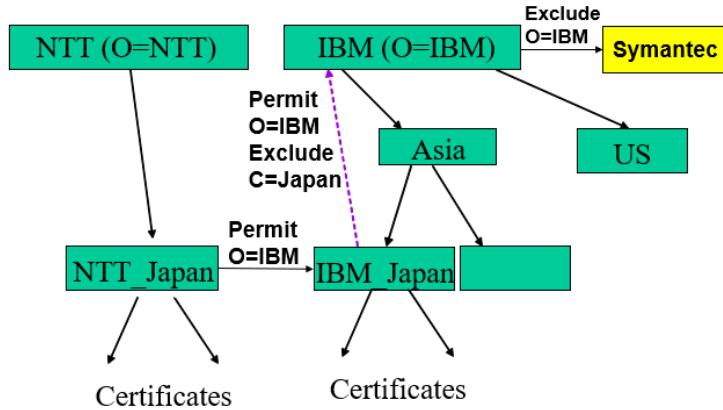


Figure 8.8: Example of the use of the *name constraint* extension, where constraints are over distinguished name keywords. NTT Japan issues a certificate to IBM Japan, with the *name constraint Permit O=IBM*, i.e., allowing it to certify only distinguished names with the value ‘IBM’ to the ‘O’ (organization) keyword, since NTT Japan does not trust IBM Japan to certify other organizations. IBM Japan certifies the global IBM, only for names in the IBM organization (*Permit O=IBM*), and excluding names in Japan (*Exclude C=Japan*). As a result, NTT trusts certificates issued by IBM to different parts of IBM, e.g., IBM US, but would not trust certificates issued by IBM to IBM Japan or to other companies, e.g., Symantec. Similarly, IBM certifies Symantec for all names, except names in the IBM organization.

only for domains in the *edu* top-level domain, and excludes domains in the subdomain *uconn.edu*. See examples in the tables in Figure 8.6 and Figure 8.7.

Note that these examples focus on the typical case of DNS domain names, however, the restrictions may apply to other types of names, e.g., email addresses or X.509 distinguished names.

Figure 8.8 presents an example of a typical application of the *name constraint* extension, using X.509 domain names. In this example, the NTT Japan CA issues a certificate to IBM Japan, allowing the IBM Japan CA to certify any certificates with the value ‘IBM’ for the organization (O) keyword - implying that IBM Japan cannot certify other organizations. Also, see IBM Japan certifying the ‘main’, corporate IBM CA, but excluding sites where the value of the country (C) keyword is Japan, i.e., not allowing corporate IBM CA to certify sites in Japan, even IBM sites. Notice that such certificate issued by corporate IBM would also be trusted by relying parties using only NTT Japan as a trust anchor, provided that other relevant constraints such as certificate path length are satisfied (or not specified).

Figure 8.9 presents a similar example, but using DNS domain names instead of X.509 distinguished names.

Unfortunately, currently, essentially all browsers do not enforce any Name

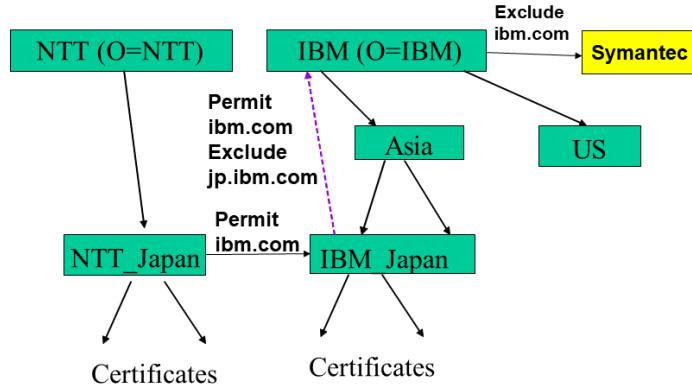


Figure 8.9: Example of the use of the *name constraint* extension, with similar constraints to the ones in Figure 8.8, but here using DNS names (dNSName).

constraints on the root CAs, and root CAs rarely enforce Name constraints on intermediate CAs. Therefore, although we believe most browsers do *support* name constraints, these are rarely actually deployed in practice.

8.3.4 The policy constraints extension

In addition to the *basic constraints* and *name constraint* extensions, X.509 and PKIX also define a third standard extension that defines additional constraints on subsequent certificates. This is the *policy constraints* extension, which is related to the certificate policies and certificate policy mappings extensions; see subsection 8.2.8.

The policy constraints extension allows the CA to define two requirements which must hold, for subsequent certificates in a certificate path to be considered valid:

requireExplicitPolicy: if specified as a number n , and the path length is longer than n , then *all* certificates in the path must have a policy required by the user.

inhibitPolicyMapping: if specified as a number n , and the certificate path is longer than n , say $C_1, \dots, C_n, C_{n+1}, \dots$, then C_{n+1} and any subsequent certificate, should not have a *policy mapping* extension.

8.4 Certificate Revocation

In several scenarios, it becomes necessary to revoke an issued certificate, prior to its planned expiration date. Different reasons for revoking a certificate are

listed in the PKIX [86] and X.509 [177] standard; we roughly categorize⁵ them as follows:

Revocation due to security concerns: these revocations are due to potential compromise of the certified key, discovery that the certificate was requested without authorization, that the certificate was misused or could be misleading, or that the subject violated its obligations or Terms of Use.

Revocation due to change: these revocations are due to a change which invalidates the information in the certificate. This can be a change to the certified name or to other certified attributes, e.g., removal of some (certified) privilege. Other reasons for change are when the certified entity ceases to operate, or just stops using the certified public-private key pair, for a benign reason (such as change of business or change to a more secure key or algorithm).

Other revocations: some revocations may be for other reasons, such as a request by the subject, a legal obligation of the CA, or some mistake or failure of the CA. For example, the Let's Encrypt CA had to revoke multiple certificates, when they detected a bug in their *CAA* (*Certificate Authority Authorization*) validation code, affecting over 3 million certificates [1], most of which were revoked. (Read about CAA in subsection 8.5.2 and [156].)

Revocation mechanisms: CRLs, OCSP and others. Revocation mechanisms are methods to inform the relying parties that a certificate was revoked. This turns out to be a significant challenge - definitely much larger than originally anticipated.

The early X.509 design [78] only offered one method for revocation, the *Certificate Revocation List (CRL)*, which is, essentially, a list of the revoked certificates, typically signed by the issuing CA; see subsection 8.4.1. CRLs are still widely *implemented* and supported by CAs, however, most relying parties prefer to use other mechanisms to check for revocations, since CRLs often have excessive overhead, mainly in terms of bandwidth. A daily download of, say, 100MB to 1GB for each relying party, is problematic for the CAs - as well as for many relying parties.

Another revocation mechanism which was standardized is the *Online Certificate Revocation Protocol (OCSP)*, see subsection 8.4.2 (and the ‘Stapled-OCSP’ mechanism in subsection 8.4.3). However, OCSP also has its own set of drawbacks, including delay, loss of privacy and of availability, overhead and vulnerability to *Denial-of-Service (DoS)* attacks, as we discuss.

Indeed, there is still no consensus on the ‘best’ revocation mechanism. In subsection 8.4.5 we discuss several other, non-standardized revocation mechanisms.

⁵While X.509 specifies an optional field to specify the reason for revocation, most certificates do not include such indication, hence, unfortunately, we do not know the distribution of reasons.

Method	Freshness	Delay	Compute	Bandwidth	Storage	Concerns
CRL (periodic)	Hours/days	None (local)	One signature, one verification	Very high: $r_{All} \cdot L_{CRL}$	High: $r_{All} \cdot L_{CRL}$	Bandwidth
$\Delta-CRL$	Hours/days	None (local)		Usually: $> r_D \cdot L_{CRL}$, sometimes: $r_{All} \cdot L_{CRL}$	High: $r_{All} \cdot L_{CRL}$	Complexity, adoption, storage
OCSP (ignore cache)	Seconds (or less)	Seconds	q_{CA} signatures, q_{RP} verifications	Medium/Low: $q_{CA} \cdot L_O$	None	Delay, overhead, availability, DoS, privacy exposure
Stapled OCSP	T_S minutes	None (in TLS)	$c \cdot \frac{24 \cdot 60}{T_S}$ signatures, q_{RP} verifications	Medium/Low: $q_{CA} \cdot L_O$	None	Delay, overhead, availability, DoS, privacy exposure
OneCRL, CRLset	Hours/days	None (local)	One signature, one verification	Medium/Low	Medium/Low	Partial coverage, proprietary, another TTP
CRV	Hours/days	None (local)		Low: $r_{All} \cdot \log c$	Low: $r_{All} \cdot \log c$	Proposed, not yet deployed
Δ -CRV	Hours/days	None (local)		Very low: $r_D \cdot \log c$	Low: $r_{All} \cdot \log c$	

Table 8.3: Comparison of revocation-checking mechanisms. Bandwidth and computations are for a 24 hours period. Computation focuses on the public key signature operations: verifying (done by the relying party) and signing. Signing is done by the CA, except for OneCRL/CRLset, where signing is done by the vendor. The parameters (c, L_{CRL}, \dots) are described in Table 8.4. Values are rough approximations (simplifications).

Parameter	Notation	Typical value
Length of CRL entry	L_{CRL}	100 bytes
Length of OCSP response	L_O	1000 bytes
Number of certificates	c	10^8
Total number of revocations	r_{All}	10^6
Revocations in 24 hours	r_D	1000
Validation-queries by a relying-party in 24 hours	q_{RP}	50
Validation-queries to a CA in 24 hours	q_{CA}	10^7
Time between OCSP requests by website (stapled OCSP)	T_S	10 minutes

Table 8.4: Revocation-related parameters. Actual values differ significantly between different PKIs; the values given are just examples.

This includes the (deployed, proprietary) OneCRL and CRLset mechanisms, the CRV and Δ -CRV mechanisms proposed in [299]. In subsection 8.4.4, we also discuss some non-standardized OCSP optimizations.

Table 8.3 compares the CRL , OCSP, OneCRL/CRLset and CRV/ Δ -CRV revocation mechanisms.

Validation and freshness of revocation information. Revocation information, sent as CRL , as OCSP response or otherwise, should be *validated* for authenticity and freshness. All of the revocation mechanisms we discuss are based on provision of signed and time-stamped *revocation information*, which

allows any party to validate it at any time (not just immediately). Furthermore, this allows the relying party to *prove* later that it received particular revocation information, which can help justify its resulting actions, such as performing a signed order which the relying party validated using the certified public key (not revoked based on information available to the relying party at that time), or refusing to perform some order (since the key *was* revoked). The correct action may also depend on the *time (and date)* of revocation, which is, therefore, often part of the information for a revoked certificate.

What is fresh revocation information? Note that revocation information may change over time, even while in transit to the relying party. Therefore, it is possible that the relying party receives, at time t information indicating a certificate is still valid, while the certificate was revoked at some time $t' \leq t$. This should happen only if t' is ‘sufficiently close’ to t , i.e., $t - t' \leq \Delta$ where Δ is the allowed period to use a certificate after the timestamp in the revocation information. The value of Δ may be defined by an extension in the certificate or within the revocation information; for example, this is the case for *CRL* and OCSP. Alternatively the allowed Δ may be the same for all certificates (defined by a specific CA, or by any CA), as is done by several (non-standard) revocation mechanisms such as OneCRL, CRLsets and CRVs.

Distribution of revocation information. Often, the relying party receives the revocation information directly from the CA. However, since the revocation information is signed and time-stamped, then it could also be relayed by third parties. This fact is utilized by *OCSP stapling*, where the revocation information is provided by the *subject*, ‘stapled’ to the certificate. OCSP stapling is specified for the common case where the certificate is provided by a TLS Server, and in this case, it is provided in the Server Hello message. We discuss OCSP stapling in subsection 8.4.3.

Retrieving revocation information: periodically or as-needed (online)? There are two main options for the retrieving revocation information: *periodically*, e.g., daily, or *as-needed*, i.e., when the relying party needs the revocation information to validate a specific certificate. Online (as-needed) retrieval may reduce the bandwidth overhead, since it avoids downloading unnecessary revocation information; on the other hand, the relying party must wait for the revocation information to arrive, introducing delay (waiting for the response) and the risk of communication failures. Online retrieval may also allow an attacker to perform a Denial-of-Service (DoS) attack against the CA or other OCSP server, by a ‘flood’ of revocation-queries; and there are also privacy concerns, due to the exposure of the identity of the certificates used by the relying party. To reduce the overhead, some relying parties use a cached OCSP response if available, and perform online retrieval only when they do not have a valid OCSP response in cache.

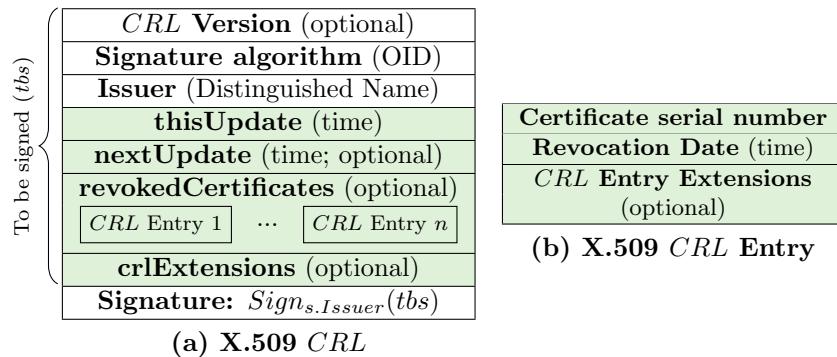


Figure 8.10: (a) X.509 CRL fields (b) X.509 CRL Entry fields. Fields with white background have corresponding fields in the X.509v3 certificate.

The impact of the periodical retrieval vs. as-needed retrieval can be clearly seen in Table 8.3. OCSP, as its name implies, retrieves revocation information as-needed (online), unless it is cached (in OCSP implementations that cache responses). Most other revocation mechanisms, including CRLs, usually retrieve revocation information periodically, although some implementations, mainly of CRLs, also retrieve them only as-needed, to save bandwidth by not downloading unnecessary CRLs. Table 8.3 does not include the less-typical options of OCSP which caches responses and of retrieving CRLs only as-needed.

8.4.1 Certificate Revocation List (CRL)

The X.509 designers probably expected revocation to be a rare incident, with a small number of certificates which were revoked (but not yet expired) at any given time. In this case, a simple solution is for the CA to periodically sign and distribute a time-stamped list of all revoked certificates, which is called a *Certificate Revocation List (CRL)*. The CA may also authorize another entity to issue CRLs; the term *CRL issuer* can be used, to refer to the entity issuing the CRLs - the CA or an entity authorized by the CA. However, for simplicity, we mostly refer to the typical case where the CA is also the *CRL issuer*.

CRLs are defined as part of the X.509 standard, already from its early versions [78]; their contents were enhanced with later versions. Figure 8.10 shows the contents of the widely used *CRL* defined in [176]. As can be seen, this *CRL* shares quite a lot with the X.509v3 certificate (Figure 8.5); in particular, it has similar version⁶, OID, issuer Distinguished Name (DN), subject DN and

⁶Confusingly, the *CRL* specification support extensions from *CRL* version 2, not 3 (as might be expected for X.509v3 certificate). Furthermore, in CRLs, the version field is optional; when the version field is absent, this indicates version 1 (which does not support extensions). In practice (and this book), CRLs are always version 2, i.e., contain both version and extension fields.

extensions fields. The fields which are unique to CRLs are:

thisUpdate: the time at which the *CRL* was issued (signed).

nextUpdate: If specified, the *nextUpdate* bounds the time when an updated *CRL* will be issued. Usually, relying parties request a new certificate prior to the nextUpdate time, i.e., it serves, essentially, as the ‘expiration date’ for the *CRL*.

revokedCertificates: this field lists one or more *CRL* Entries. The contents of *CRL* Entries are shown in Figure 8.10 (b). Each *CRL* Entry contains (1) the serial number of the revoked certificate, (2) the revocation date (and time), and (3) optional extensions, much like the X.509v3 certificate (and the *CRL*) extensions.

crlExtensions: an optional field, that may contain extensions to the *CRL*, much like the X.509 certificate extensions.

A relying party should use a valid, non-expired *CRL* to check if a certificate issued by the CA was revoked. Almost always, the *CRL* is cached until it is replaced by a more-recently-issued *CRL* (from the same issuer and set of certificates). The relying party typically requests the *CRL*, either periodically, to ensure a fresh *CRL* is available when needed, or only when needed to validate a certificate.

The CRL Distribution Points (‘*cRLDistributionPoints*’ certificate extension.) To retrieve the *CRL*, the relying party usually uses the *cRLDistributionPoints* certificate extension, defined in [176]; this extension defines how (using what protocol) to retrieve the *CRL*, and what address to use. Specifically, this information is provided by one or more *DistributionPoints* entries, each of which defines a *URI* (Universal Resource Locator), which defines the protocol and location for downloading the *CRL*.

Bandwidth overhead is a major concern with CRLs, since CRLs can often be quite large. The size of a single *CRL* entry, which we denote L_{CRL} , can differ significantly among providers, but most are close to 100 bytes; and the total number of revocations, which we denote r_{All} , could be a million or even more. This results in total *CRL* length of $L_{CRL} \cdot r_{All}$ of around 100 million bytes per CA, considerable overhead for both CAs and relying parties. Measurements of *CRL* overhead were reported in [328], who found median *CRL* of 51KB and maximal *CRL* of 76MB (yes, this is in Mega Bytes!); and [299], who found average *CRL* of 173KB. The reason for that is that the number of revocations may be surprisingly high; specifically, [328] found that about 8% of the non-expired certificates were revoked, mostly due to the Heartbleed bug [77], and Let’s Revoke discovered a bug in their issuing which required revocation of three million certificates [252]. However, even looking at measurements for periods without such events, we see that about 1% of the non-expired certificates are revoked - which can still result in excessively long CRLs.

Three standard X.509 CRL extensions are designed to reduce the bandwidth overhead of CRLs:

The *Issuing Distribution Point (IDP) CRL extension* and *CRL scopes*.

A *CRL*, available from some *DistributionPoint*, should contain all revoked certificates which were issued by the CA and not yet expired, belonging to the *scope* of the *CRL*. By default, all certificates issued by the CA have the same scope, i.e., are available from the same *DistributionPoint*. However, often, CAs prefer to use multiple smaller CRLs, by splitting the set of certificates into separate scopes, e.g., based on issuing time. X.509 [176] specifies that in this case, the *CRL* must contain the standard, critical *CRL* extension called *Issuing Distribution Point (IDP)*, which will define the relevant scope. The main motivation for a CA to use multiple CRLs, each with distinct scope (defined using the IDP *CRL* extension) and *DistributionPoint*, is to reduce the bandwidth overhead. The use of multiple *DistributionPoints* (and scopes) reduces the length of each *CRL*, at the cost of requiring the CA to sign and distribute multiple CRLs. When relying parties download CRLs only as-needed, this may reduce the required bandwidth, but at the cost of reduced likelihood that the required *CRL* is cached, i.e., more cases where the relying party must download the *CRL* to validate a certificate, which can cause significant delay and availability concerns. OCSP can be seen as an extreme case, with each certificate requiring a separate request/response.

The *Authorities Revocation List (ARL) extension* lists only revocations of CA certificates. This is essentially equivalent to placing CA certificates in a dedicated distribution point, and becomes meaningful mainly when relying parties download only the ARL, and use other mechanisms, such as OCSP, for non-CA certificates.

The *Delta CRL extension* lists only new revocations, which occurred since last base-*CRL*, which is retrieved as needed. To validate that a given certificate is not revoked, check if it is contained either in the Delta-CRL or in a base-CRL, issued not earlier than the time specified in the Delta-*CRL*. For this method to be effective, relying parties should cache, and periodically download, the base-*CRL*, so, the storage requirements are the same as when using ‘regular’ CRLs, and sometimes also the bandwidth requirements. Also, implementation is more complex, especially if the relying party may need to ‘prove’ to a third party, in the future, that she relied on a certificate that was not revoked at the time. Possibly due to such concerns, Delta-CRLs are not widely deployed.

Even with such optimizations, CRLs may still introduce significant bandwidth overhead.

When to download CRLs: periodically (in advance) or as-needed?
The original X.509 *CRL* design, was to download all CRLs *in advance*, in a

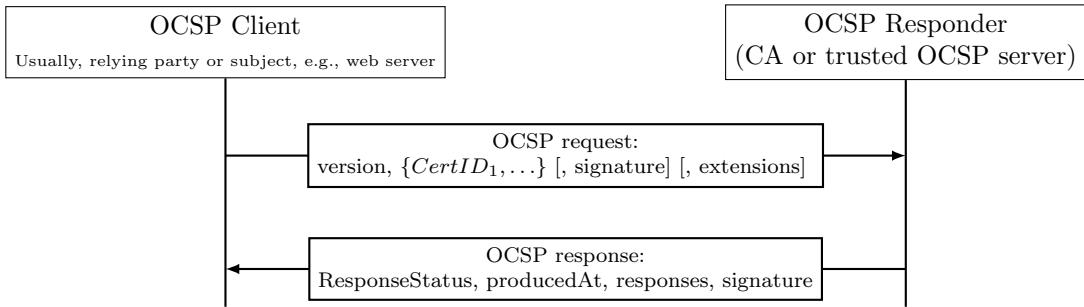


Figure 8.11: The *Online Certificate Status Protocol (OCSP)*. The request includes one or more certificate identifiers $\{CertID_1, \dots\}$; requests are optionally signed. The OCSP response is signed by the responder, and includes response for each $CertID$ in the request. Each of these ‘individual responses’ includes the $CertID$, cert-status, time of this update, time of the next update, and optional extensions. Cert-status is either *revoked*, *good* or *unknown*.

periodical process, e.g., daily [78]. This periodical process should be done with sufficient frequency, to make sure that the revocation information is reasonably updated (fresh). Since CRLs can be quite long, and many CRLs are not required in any given day, this results in considerable overhead.

Therefore, implementations often fetch the CRLs only *as-needed* (online). However, this may cause increased delay (waiting for information to arrive), reduced reliability (what to do if revocation information is unavailable), and privacy concerns (e.g., exposing the website being visited). As a result, the use of CRLs has become less and less common; e.g., it is not done, currently, by major browsers.

A standardized alternative to CRLs is the *Online Certificate Status Protocol (OCSP)* standard, which we discuss in subsection 8.4.2, subsection 8.4.3. We later also discuss other, non-standardized alternatives, in subsection 8.4.4 and subsection 8.4.5.

8.4.2 Online Certificate Status Protocol (OCSP)

OCSP (Online Certificate Status Protocol) [287], shown in Figure 8.11 is a request-response protocol, providing a secure, *signed* indication to the relying party, showing the ‘current’ status of certificates (details below). The protocol involves two entities: the *OCSP client*, who sends an *OCSP request* to request the status of one or more certificates, and the *OCSP responder* (server), who responds with a (signed) *OCSP response*, indicating the status of the certificate(s).

The *OCSP client*, i.e., the entity that sends the OCSP request, is either the relying party or another party. In this subsection, we focus on the ‘classical’ OCSP deployment, where the relying party, e.g., browser, sends the OCSP request to the CA (or other OCSP responder), as in Figure 8.12; in this case, the

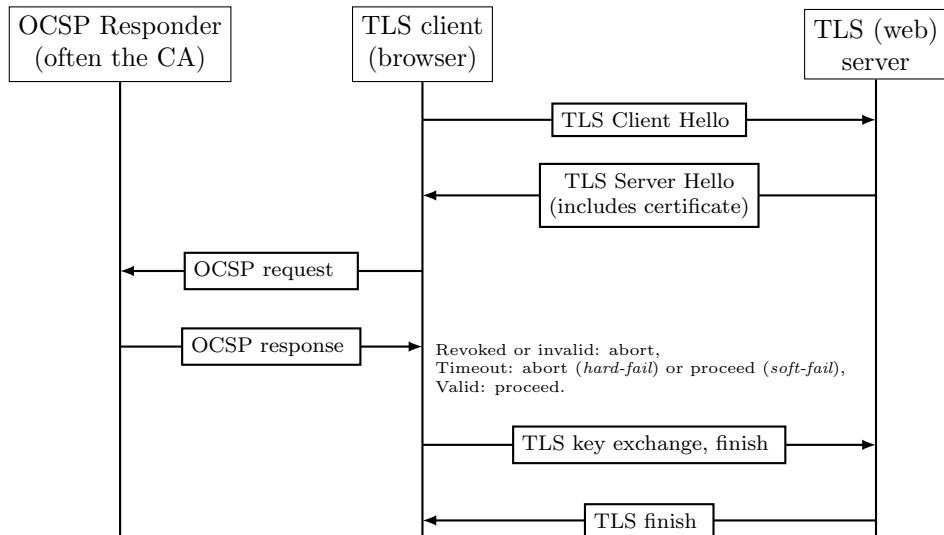


Figure 8.12: OCSP used by relying party (as OCSP client). There are several concerns with this form of using OCSP, including privacy exposure, overhead on CA, and handling of delayed/missing OCSP response by the client/browser. This last concern, illustrated in Figure 8.13, motivated updated browsers to support and prefer *OCSP-stapling* (see Figure 8.14), where the TLS/web server makes the OCSP request, instead of the client/browser, and ‘staples’ the OCSP response to the TLS server hello message.

relying party (often browser) acts as the OCSP client. Later, in subsection 8.4.3, we discuss the *stapled-OCSP* deployment, where it is the subject, e.g., website, who sends the OCSP request, i.e., the subject (often website) acts as the OCSP client.

The *OCSP responder*, i.e., the entity that processes OCSP requests and sends responses, is an entity trusted by the relying party; we will assume this is the CA itself, although it could also be another entity, delegated by the CA. Each OCSP response message is signed by the OCSP responder or the CA, allowing the relying party to validate it, even if received via an untrusted intermediary, e.g., the subject (website).

Improving efficiency with multi-cert OCSP requests. To improve efficiency, a single OCSP request may specify (request status for) multiple certificates (*CertIDs*)⁷. Correspondingly, a single OCSP response, using a single signature, may include (signed) responses for multiple certificates. The support for OCSP requests and responses for multiple certificates, is especially

⁷Certificate identifiers (*CertIDs*) may be specified using the hash of the issuer name and key, and a certificate serial number.

Note 8.2: Using OCSP to validate a Certificate-Path (CP)

An *indirectly trusted* certificate, certified via a certificate chain, consisting of certificates issued by (one or more) intermediate CAs, may be invalidated via revocation of any of these certificates. A relying party wishing to validate the status of the indirectly-trusted certificate, needs to check for revocation of the intermediate-CA certificates in the chain, not only of the indirectly-trusted certificate itself. Since Intermediate-CAs are critical elements of the PKI, and their number is much smaller than end-entities, relying parties may use other mechanisms to check for revocation of intermediate-CA certificates. Specifically, intermediate-CA certificates are often validated using (special) CRLs or proprietary mechanisms such as OneCRL or CRLset. However, sometimes, their validity should be checked using OCSP. The fact that an OCSP request may include multiple certificates, allows this process to be more efficient; a single OCSP request-response interaction may suffice to obtain updated status for all of these certificates, provided that the same OCSP responder is able to provide (signed) OCSP responses for all of these certificates (issued by different CAs). The original OCSP stapling specification, RFC 6066 [117], does not support stapling of multiple certificates. This is addressed in TLS 1.3 [272], which allows the RFC 6066 information to be attached to every certificate in the chain sent by the server. Alternatively, implementations of older versions of TLS can use the (later-defined) ‘multiple certificate status’ extension, RFC 6961 [260].

important when certificates are signed by intermediate CAs, using a Certificate-Path; see Note 8.2.

OCSP vs. CRLs. The length of an OCSP response is linear in the number of *CertIDs* in the corresponding OCSP request, rather than a function of the total number of revoked certificates of this CA, as is the case for CRLs. Furthermore, the computation required for sending an OCSP response is just one signature operation, plus some hash function applications, regardless of the number of revoked certificates or the number of certificates whose status is requested in this OCSP request. In the common case where the total number of revoked certificates may be large, this significantly reduces the overhead of generating and distributing often large *CRL* responses. Namely, OCSP provides an alternative which is often more efficient than CRLs; with CRLs, the CA must ‘push’ the list of all revocations to all relying parties, while with OCSP, a relying party receives information only about relevant certificates. In addition, OCSP responses are sent on a timely fashion, when the relying party is validating the relevant certificate - which may provide a more ‘fresh’ indication compared to the periodical *CRL*. As a result of these advantages, OCSP appears to be deployed more than CRLs. However, OCSP has its own set of challenges, so it is also not widely used. Let us discuss these challenges.

OCSP Challenges: ambiguity, failures and delay. OCSP status responses for each certificate may specify one of three values: *revoked*, *good* or

unknown. The ‘unknown’ response is typically sent when the OCSP responder does not serve OCSP requests for the issuer of the certificate in question, or cannot resolve their status at the time (e.g., due to lack of response from the CA). These *unknown* responses are *ambiguous*; relying parties are left to decide how to interpret and respond to it. These *ambiguous* responses are quite problematic, as we explain below. But first let us discuss another OCSP scenario that also leads to similar ambiguity: *failed requests*.

An OCSP request may fail in multiple ways. One way is when the OCSP client fails to establish communication or receive response from the OCSP server. Another reason is when the OCSP responder sends back an *OCSP failure return code*, indicating a reason for failure. These reasons include:

- Lack of signature on OCSP request (when required by OCSP responder)
- Request not properly authorized/authenticated, e.g., not from known IP address, or missing/incorrect authentication information, when required by OCSP responder. Authentication information should be provided by the client in an appropriate OCSP extension.
- Technical reasons, such as overload or internal error.

Recall now that in the ‘classical’ OCSP deployment, the OCSP client is the relying party, typically, the browser, as in Figure 8.12. However, this creates a dilemma for the browser (or other relying party): how should the relying party respond to OCSP failures and ambiguous responses, e.g., when a response does not arrive (within reasonable time) or indicates an OCSP failure? The following are the main options - and why each of them seems unsatisfactory:

Wait: if the problem is timeout, then the relying party may simply continue waiting for the OCSP response, possibly resending the request periodically, and never ‘giving up’. However, OCSP servers could fail or become inaccessible forever, or for extremely long, leaving the relying party in this state. We do not believe any relying party has taken or will take this approach; also, it does not address the other types of OCSP ambiguities. In fact, even when the relying party ‘time-outs’ if the OCSP response is not received within reasonable time, the *delay* of waiting for the OCSP response is often a concern.

Hard-fail: abort the connection (and inform the user). That is clearly a ‘safe’ alternative, i.e., prevent use of a revoked certificate. However, the OCSP interaction may often fail or return ambiguous response due to benign reasons, such as network connectivity issues or overload of the OCSP responder. In particular, usually, the OCSP responder is the CA, and CAs often do not have sufficient resources to handle high load of OCSP requests. Therefore, this approach is not widely adopted.

Ask user: the relying party may, after some timeout, invoke a user-interface dialog and ask the user to decide if to continue with the connection or

abort it. For example, a browser may invoke a dialog, informing the user that the certificate-validation process is taking longer than usually, and ask the user what action it should take. While this option may seem to empower the user, in reality, users are rarely able to understand the situation and make an informed decision, and are very likely to continue with the connection; see discussion of usability in Chapter 9. Hence, except for ‘shifting the responsibility’ to the user, this option is inferior to direct soft-fail, discussed next.

Soft-fail: finally, the relying party may simply continue as if it received a valid OCSP response. By far, this is the most widely-adopted option. In the typical case of a benign failure to receive the OCSP response, there is no harm in picking this option. However, this choice leaves the user vulnerable to an impersonation attack using a revoked certificate, when the attacker can block the OCSP response; see Figure 8.13. Since our need for cryptography is mainly due to concerns about a *Man-in-the-Middle* attacker, who can surely block communication, this option results in vulnerability.

As Figure 8.13 shows, the soft-fail approach essentially nullifies the value of OCSP validation - against an attacker that can block or sufficiently delay the OCSP request/response, if the attacker has exposed the private key of the TLS (web) server, or has obtained a fake certificate for the server’s domain (that was later revoked). Both exposing of the private key and obtaining a fake certificate are challenging attacks. However, such attacks do occur, which is one reason we need revocations; see examples of such attacks in Table 8.5. The other condition, of being able to block the OCSP response, is often surprisingly easy for an attacker, e.g., by sending an excessive number of OCSP requests to the OCSP responder (e.g., the CA) at the same time as the OCSP request from the relying party. In particular, an attacker is likely to be able to launch such attack by intentionally invoking appropriate links from a website controlled by the attacker, in a so called *web-puppet* attack; see the web-security chapter of [161]. In spite of this, soft-fail is common choice of browsers and most other relying parties, basically, since developers give more weight to *user-experience (UX)* considerations, than to security considerations - see the *UX>security* precedence rule (Note 8.3). Unfortunately, as we explained, this allows attackers to circumvent OCSP and use revoked certificates, by intentionally causing a failure to the OCSP challenge-response communication.

There are several additional problems with the use of ‘classical’ OCSP deployment, where the OCSP request is sent by the relying party (often, browser):

Delay: since OCSP is an online, request-response protocol, its deployment at the beginning of a connection often results in considerable delay.

Privacy exposure: the stream of OCSP requests (and responses) may expose the identities of websites visited by the user to the OCSP responder, or

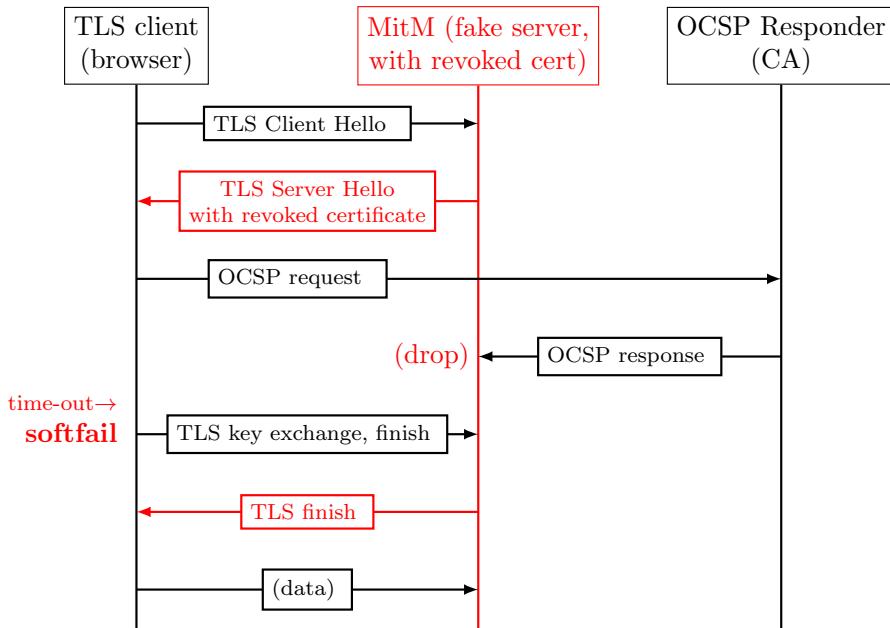


Figure 8.13: The MitM soft-fail Attack on a TLS connection using OCSP. The attack assumes ‘classical’ OCSP deployment, where the TLS-client (browser) sends the OCSP request (acts as OCSP client), and (vulnerable) soft-fail handling of timeouts and ambiguous OCSP responses. The attacker is impersonating as a website, to which the attacker has the private key; the corresponding certificate is already revoked, but the attack allows the attacker to trick the browser into accepting it anyway, allowing the impersonation attack to succeed. The browser queries the CA (or other OCSP server) to receive a fresh certificate-status. However, the attacker ‘kills’ the OCSP request, or the OCSP response (figure illustrates dropping of the response). After waiting for some time, the browser times-out, and accepts the revoked certificate sent by the impersonating website, although no OCSP response was received. This soft-fail behavior is used by most browsers, since the alternatives (very long timeout, asking the user, or hard-fail) are not well received by users.

Note 8.3: The UX>Security Precedence Rule

In the *OCSP soft-fail vulnerability*, as described in Section 8.4.2, most browsers support OCSP, but only using *soft-fail*, namely, if the OCSP-response is not received within some time, then the browser simply continues with the connection, i.e., ‘gives up’ on the OCSP validation and continues using the received certificate, basically, assumes that the certificate was not revoked. It is well understood that this allows a MitM attacker to foil the OCSP validation, i.e., the use of the soft-fail approach results in a known vulnerability. Still, browser developers usually prefer to have this vulnerability, to the secure alternative of *hard-fail*, namely, aborting a connection after ‘giving up’ on the OCSP response. The reason is that there are also benign reasons that may cause the OCSP response not to arrive, such as unusually high delay due to network congestion or high load on the OCSP responder (typically, the CA). Aborting a connection in such cases would result in loss of availability. If the response is only delayed and eventually arrives, waiting for a long time would result in poor performance.

Loss of availability, performance, reliability and functionality, are all immediately visible to the end users, i.e., they harm the *user experience (UX)*. User experience has a direct, immediate impact on the success of a product. In contrast, security and privacy considerations are rarely visible to the users. As a result, even when vendors and developers care about security and privacy, they usually prefer to compromise on these goals, to avoid harming the *user experience (UX)* aspects: *availability, functionality, performance, usability and reliability*. We refer to this as the *UX>Security Precedence Rule*.

Principle 16 (The UX>Security Precedence Rule). *Vendors and developers give precedence to the user experience (UX) considerations (availability, functionality, performance, usability and reliability), than to the security and privacy considerations.*

Of course, the UX>Security Precedence Rule is just a simplification; real decisions are more complex, and some vulnerabilities will be considered so critical, that developers will prefer to fix them, even at the cost of some reduction in UX. However, usually, the challenge for designers and researchers is to find solutions which will ensure sufficient security, but avoiding or minimizing harm to the user experience (UX).

to other agents able to inspect the network traffic. By default, OCSP requests and responses are not encrypted, exposing this information even to an eavesdropper; but even if encryption is used, privacy is at risk. First, the CA is still exposed to the identities of websites visited by a particular user. Second, even with encryption of OCSP requests and responses, the timing patterns create a *side-channel* that may allow an eavesdropper to identify visited websites.

Processing overhead: while OCSP often reduces overhead significantly compared to CRLs, it still requires each response to be signed, which is computational burden on the OCSP responder. In addition to this computational overhead, there is the overhead of *processing* each of the (many) OCSP requests; this overhead remains even when applying optimizations that reduce the OCSP computational overhead, e.g., as in subsection 8.4.4 and Exercise 8.20.

The processing overhead is especially a concern for the OCSP responder. Consider the typical case, of a CA providing OCSP responder service; the signatures in OCSP responses imply significant processing overhead, which can be a significant concern to the CA. Normally, CAs cannot charge for the overhead of handling these OCSP requests; and to provide reliable service, they should be ready to respond to a *Flash Crowd*⁸ of requests, from visitors of a (suddenly popular) website, or to respond to request sent as part of an intentional *Denial-of-Service* attack (on the CA or on a subject of a certificate).

Due to the overhead concerns, an OCSP responder may limit its services to authorized OCSP clients. To support this, OCSP requests may be signed; some servers may use other ways to authenticate their clients, e.g., using the optional extensions mechanism supported by OCSP requests.

We next describe *OCSP stapling*, where the OCSP client is the *subject* of the certificate rather than the relying party. The goal of OCSP stapling is to mitigate these security, privacy and efficiency concerns. In subsection 8.4.4 we discuss additional methods to reduce the computational overhead of OCSP.

8.4.3 OCSP Stapling and the Must-Staple Extension

In the previous subsection, we have seen several disadvantages of the ‘classical’ OCSP deployment, where the relying party sends the OCSP requests (i.e., acts as the OCSP client). In this section we discuss an alternative approach, the *OCSP stapling* deployment, where the OCSP request is sent by the *subject*, typically the website, acting as the OCSP client. Namely, this design moves the responsibility to obtain ‘fresh’ OCSP signed responses to the *subject* (e.g., web-server), rather than placing this responsibility (and burden) on every client (e.g., browser). This addresses the privacy exposure and reduces the overhead on the OCSP responder (typically, the CA), since it now needs only to send a

⁸The term Flash Crowd is the name of a sci-fi novella by Larry Niven, describing ‘physical flash crowd’ due to the use of a transfer booth.

single signed OCSP response to each subject (website) - much less overhead than sending to every relying party (browser). Furthermore, since now only the subject is supposed to make OCSP requests, the CA may limit the service to its customers, the subjects.

Therefore, of all the concerns discussed for the relying-party-based OCSP, only one remains: handling of ambiguous OCSP responses, and in particular, the *MitM soft-fail* attack (Figure 8.13). We discuss two variants of OCSP stapling, which handle, in two different ways, such ambiguities and failures.

OCSP Stapling. *OCSP stapling* is a different way to deploy OCSP, where the *subject* runs the OCSP client and periodically sends OCSP requests to the OCSP responder for an OCSP response for the server's certificate, e.g., C_B .

Let us focus on the typical scenario, where the relying party is a browser running TLS, who receives a certificate C_B from the web (and TLS) server, e.g., *bob.com*, who is the *subject* of the certificate C_B . In *OCSP Stapling*, the subject (web server) *periodically* sends an OCSP request to the OCSP responder (CA). The web-server does this periodically, *without waiting* for the TLS Client Hello message from the client. The CA (or other OCSP responder) sends back the OCSP response; usually, the response indicates that C_B is still Ok (not revoked), at the current time $time(\cdot)$. We denote this response as σ ; importantly, $\sigma = \text{Sign}_{CA.s}(C_B \text{OK}; time(\cdot))$, i.e., contains a signature by the private signing key $CA.s$ of the CA, on the web-server's certificate C_B and the current time. This response should satisfy browsers (as relying parties), at least until *bob.com* will 'refresh' it by again sending OCSP request for C_B . The web-server, e.g., *bob.com*, keeps the response σ , providing it to all connections by OCSP-stapling-supporting browsers, until it would request and receive a newer OCSP response, in the next period.

When an OCSP-stapling-supporting browser connects to *bob.com*, it indicates its support for OCSP-stapling by including the *CSR TLS extension*; CSR stands for the *Certificate Status Response* TLS-extension. If the server supports stapling and has a valid OCSP response σ , then *staples* (includes) the OCSP response σ , which it places in the CSR TLS-extension, sent in the server's response. See this scenario in Figure 8.14.

Note that we discuss here the variant of OCSP deployment, where stapling is *optional*; i.e., the web-server may *not staple* an OCSP response, e.g., if the web-server did not receive the OCSP response from the OCSP responder. We later discuss *OCSP Must-Staple*, a variant of OCSP deployment where the subject *commits* to sending a valid OCSP response.

Once the browser receives the OCSP response (in the CSR TLS-extension), it validates it, i.e., validates the signature of the CA (using the CA's public validation key $CA.v$), and then validating that the response indicates non-revocation (which we marked by OK) and that the *time* indicated is 'recent enough'. When all is Ok, the browser completes the TLS handshake with *bob.com* and then continues with the TLS connection.

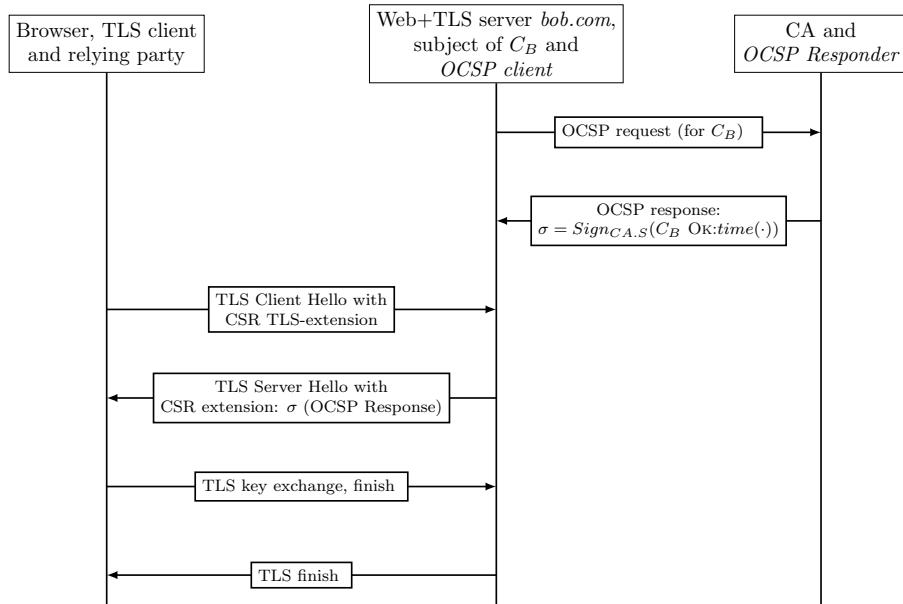


Figure 8.14: (Optional) OCSP stapling in the TLS protocol, using the *Certificate Status Request (CSR)* TLS extension, for a typical TLS connection between browser and web-server *bob.com*, the subject of certificate C_B . *bob.com* received C_B from the CA (not shown); the CA is also the OCSP responder. The web (and TLS) server *bob.com* periodically sends OCSP requests to the CA (also OCSP responder), requesting the status of its own certificate C_B . The CA sends back the OCSP response, $\sigma = \text{Sign}_{CA,S}(C_B \text{ OK}:time(\cdot))$, signaling that C_B was not revoked up to time $time(\cdot)$. The browser sends the TLS CSR extension to *bob.com* with TLS Client Hello, to request OCSP-stapling. The server sends back σ , the OCSP response, also in the CSR extension. The TLS handshake now completes as usual.

We described the OCSP-stapling process for a TLS connection between a browser and a web-server, for the case where the certificate was issued by a root CA (directly trusted by the browser). However, the process is exactly the same for other TLS clients and servers, and the modifications for the (typical) case of intermediate CA are simple, following the multi-cert OCSP request-response as discussed earlier, including in Note 8.2.

Handling Ambiguous OCSP responses and the MitM soft-fail attack. Let us now return to discuss the handling of ambiguous OCSP responses, and in particular, handling of the case where no OCSP response is received. For stapled OCSP, such failure may happen either between subject of the certificate, typically the web-server, who acts as the OCSP client, and the CA (OCSP responder); or between the relying party, typically the browser, and the subject

(web-server). In particular, this will happen if the web-server does not support OCSP stapling.

In any case, the bottom line is that the browser does not receive a stapled OCSP response from the web-server. In the ‘optional’ OCSP stapling design, this simply directs the browser to attempt to resolve the revocation situation by itself. Typically, the browser would now perform an OCSP query directly with the OCSP responder (typically, the CA), or even request the *CRL*.

However, now we are basically back in the ‘classical OCSP’ deployment, where OCSP (and/or *CRL*) are deployed by the relying party. So, let us consider again the browser’s response if it fails to receive a response to its OCSP (or *CRL*) request. This places the browser in similar dilemma to the one discussed earlier - and most implementations would adopt the *soft-fail* approach, i.e., use the certificate *assuming* that it was not revoked.

Unfortunately, this implies we are again vulnerable to an *MitM soft-fail* attack, similar to the one presented earlier (Figure 8.13). The attack is only slightly modified due to the failed effort for OCSP stapling, and should probably be quite clear from Figure 8.15.

One way to defend against the *MitM soft-fail* attack (Figure 8.15), is using the *Must-Staple* extension to the server’s X.509 certificate, which we discuss next.

The Must-Staple X.509 extension: enforcing OCSP stapled response. The attacks of Figure 8.13 and Figure 8.15 show the risk of adopting the *soft-fail* approach. The soft-fail mechanism is the equivalent of deciding to allow bypassing of airport security screening, whenever the line becomes too long. A likely outcome of such policy would be that an attacker will find ways to cause the line to be congested, and then use the bypass to avoid screening and perform an attack. We sum this up with the following principle.

Principle 17 (Soft-fail security is insecure). *Defenses should not be bypassed due to failures:* if defenses are bypassed upon failure, attacker will cause failures to bypass defenses. *Namely,* soft-fail security is insecurity.

Awareness of the risk of the soft-fail approach, motivates adoption of the harsher, *hard-fail* approach. However, this conflicts with the *UX>Security* precedence rule (Principle 16). Definitely, it would be absurd for a browser to refuse connection to a website, only since it does not receive the OCSP response; this is very likely due to a benign reason, such as that the website does not support OCSP stapling!

The *TLS-feature* X.509 extension [155] is the standard solution to this dilemma. This extension to the website’s X.509 certificate can be used to indicate that the *website always staples* OCSP responses. To a large extent, this moves the UX vs. Security decision from the browser to the website: the browser would apply the ‘must-staple’ policy, only to a website that requests it, by using the ‘must-staple’ extension in its X.509 certificate. As shown in Figure 8.16, this foils the *MitM soft-fail* attack on OCSP-stapling TLS client

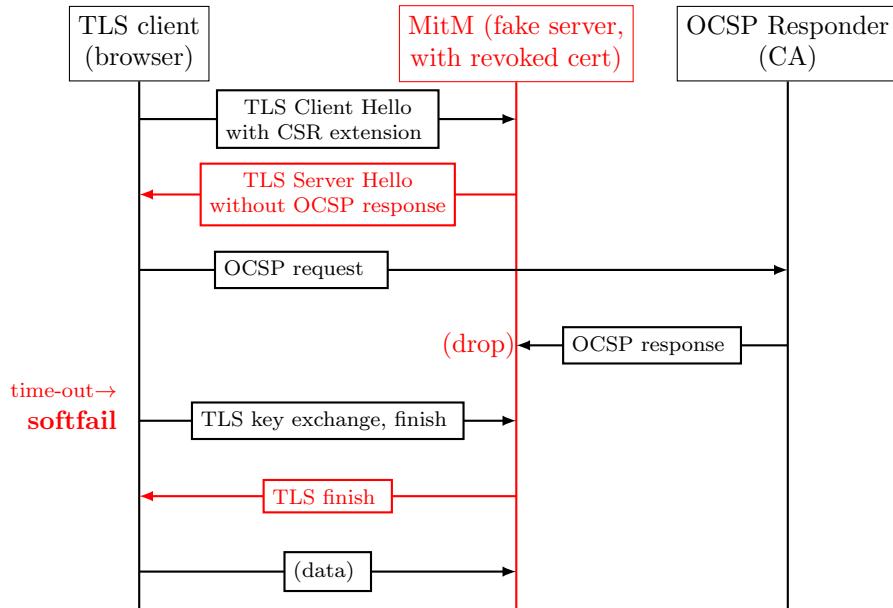


Figure 8.15: *MitM soft-fail attack on OCSP-stapling TLS client (browser), using a revoked TLS server (website) certificate; assume that the attacker has the certified (and revoked) private key.* The browser sends the CSR TLS extension; however, the website's certificate does not have the X.509 Must-Staple extension, or the client does not respect this extension. The attacker impersonates as the web-server, and sends the TLS server-hello and certificate messages; the attacker does not send the OCSP response (which would have indicated revocation). The client is misled into thinking that the server does not support OCSP stapling. The client may now send an OCSP request to the appropriate OCSP responder, e.g., the relevant CA, but the MitM attacker would ‘kill’ the OCSP request or response (the figure shows killing of the response). After time-out, the client ‘gives up’ on the OCSP response, and ‘soft-fails’, i.e., accepts the certificate and establishes the connection with the impersonated website).

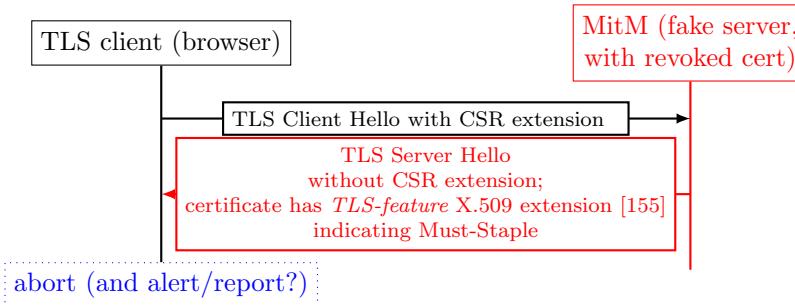


Figure 8.16: The use of the *TLS-feature* X.509 extension [155], to indicate *Must-Staple*, defends against the *MitM soft-fail* attack on OCSP-stapling TLS client of Figure 8.15. As in Figure 8.15, the attacker tries to impersonate a website, to which the attacker has the private key and the corresponding certificate, which was already revoked. As in Figure 8.15, the client sends Client-Hello request, with the CSR TLS extension, i.e., asking the server to staple OCSP response. As in Figure 8.15, the attacker responds without the CSR extension, i.e., trying to mislead the client into falling back to sending an OCSP request (and then soft-failing). However, the Must-Staple extension instructs the client to refuse to continue without the OCSP response from the server.

of Figure 8.15. Note, however, that the *TLS-feature* is only effective when the attacker tries to abuse a certificate issued to the legitimate website (with *TLS-feature* extension) and later revoked, e.g., after key-exposure was detected or suspected. In the common case where the attacker is able to get a CA to issue a certificate for a request sent by the attacker, the attacker can surely ask *not* to include this extension, allowing the attacker to avoid the must-staple mechanism.

Mandatory Must-Staple? The inclusion of the Must-Staple certificate extension in a certificate C prevents an attacker from abusing C after C was revoked, when C was revoked due to (suspected or detected) exposure of the private key. However, the Must-Staple extension does not prevent an attacker from abusing a rogue certificate $C_R \neq C$, e.g., a certificate with a misleading domain-name (subsection 8.1.1), even after the CA revokes C_R (and/or C), if the rogue certificate C_R does *not* include the must-staple extension. Such rogue certificate C_R can still be used to attack a client which does not request and wait for OCSP approval. One way to prevent this would be a *mandatory* Must-Staple extension, but this seems unlikely to happen. In fact, there are significant challenges to the adoption of the Must-Staple extension, as we now explain.

Must-Staple Adoption Challenges. The UX>Security precedence rule (Principle 16) applies also to websites; website developers would be reluctant

to adopt the *Must-Staple* extension, if they believe this may jeopardize the availability of their website. That may be due to different reasons, such as clients processing the extension incorrectly, web-servers not supporting the extension or the OCSP process correctly, or to not receiving the OCSP response from the OCSP responder (usually, the CA).

Unfortunately, measurements of adoption published so far were not very encouraging [83]. One possible reason is that CA may be reluctant to support Must-Staple; indeed, with the proliferation of websites, it is likely that the use of Must-Staple will result in increase rate of OCSP responses by CAs, each requiring a new signature - a potentially significant overhead. See some (non-standardized) possible optimizations in subsection 8.4.4.

However, we still hope that Must-Staple will be gradually adopted, as it offers significant security advantages, with high efficiency to relying parties and subjects. There does not appear to be any technical reason for either the incorrect processing or for failures of the web-servers to receive OCSP responses (and then provide them to the browsers).

Indeed, this is an example of the significant *adoption challenges* facing designers of new Internet and web security mechanisms. Adoption considerations should be an important part of the design process. In the following exercise, we discuss some issues which may help - or hinder - the adoption of the OCSP *Must-Staple* extension.

Exercise 8.7. *For each of the following variants of the OCSP Must-Staple extension process, explain possible impacts on adoption, security and performance:*

1. *Mark the Must-Staple extension as a critical X.509 extension.*
2. *Mark the Must-Staple extension as a non-critical X.509 extension.*
3. *When a browser receives from a website a certificate with Must-Staple extension, but without the stapled OCSP response, then the browser would not abort the connection, but request a certificate from the CA, and abort the connection only if this request also fails.*
4. *Same as previous item, however, the website/CA will have the ability to indicate if the client should try sending OCSP request to the CA (if it does not receive it stapled from the web-server). Consider three ways to indicate this: (a) an option of the OCSP Must-Staple extension, (b) a separate extension, or (c) an option indicated in a TLS extension returned by the web server.*

Notice that *Must-Staple* extension requires support by the CA, to include it in the web-server's certificate, and to provide sufficiently-reliable OCSP service. An alternative solution which does not require such special certificate-extension is discussed in Exercise 8.18.

8.4.4 Reducing OCSP Computational Overhead

As can be seen in Table 8.3, OCSP performance is, typically, better in most aspects: low-delay, relatively low bandwidth, and no storage required. However, OCSP *computational* overhead can be higher. For the CA, this overhead is mainly due to the signature operations; in the relying party, it is mostly due to signature-verifications.

In this subsection, we discuss several non-standard optimizing-variants of OCSP, which can reduce its computational overhead.

The Certificate-Hash-Tree. This OCSP variant uses the Merkle-tree scheme, introduced in Section 3.9, to allow the CA or OCSP responder to periodically perform a *single* signature operation, to provide OCSP responses indicating status for any OCSP requests. Assume that the CA issued a large set of certificates c_1, c_2, \dots, c_n , but each OCSP request will contain only one or few certificate-identifiers.

As shown in Figure 8.17, the signature is computed over the result of a hash-tree applied to the entire set of certificates issued by the CA (and their statuses), concatenated with the current time. The leaves of the hash-tree are the pairs of individual certificates c_1, \dots, c_n and their corresponding statuses s_1, \dots, s_n . The construction uses a collision-resistant hash function (CRHF) denoted h .

The OCSP response for a query for status of certificate c_i , consists of this signature, the *time* of signing, and the values of ‘few’ internal nodes, essentially, one node per layer of the hash tree. This allows the OCSP client to recompute the result of the hash tree, and then validate the signature. For example, to validate the value of c_6 , the response should include h_5 , h_{7-8} and h_{1-4} . To validate, compute $h_6 = h(c_6)$, then $h_{5-6} = h(h_5 + h_6)$, then $h_{5-8} = h(h_{5-6} + h_{7-8})$, then $h_{1-8} = h(h_{1-4} + h_{5-8})$ and finally verify the signature over h_{1-8} and *time*, by validating that $\text{verify}_{CA.v}(\sigma_{1-8}, h_{1-8} + \text{time})$ returns *true*. We refer to this set of values (e.g., $c_6, h_5, h_{7-8}, h_{1-4}$) as *Proof-of-Inclusion* (*PoI*) of c_6 .

Exercise 8.8. Consider the Certificates-Merkle-tree variant of OCSP, described above and illustrated in Figure 8.17.

1. Present pseudo-code for the validation of the OCSP responses by a client (relying party), when using this variant.
2. Let c be the number of certificates issued by a CA, r_{All} be the number of revoked certificates, and i be the number of certificate-identifiers sent in a given OCSP request. Note that $r < n$ and, typically, $i \ll r$. What is the number of signature and hash operations, required to (a) produce and send a CRL, (b) produce and send an OCSP response, (c) produce a certificate-hash-tree OCSP response.
3. This variant uses a (keyless) collision-resistant hash function (CRHF) h . Explain a disadvantage of this requirement and suggest a change to the design that will avoid this disadvantage.

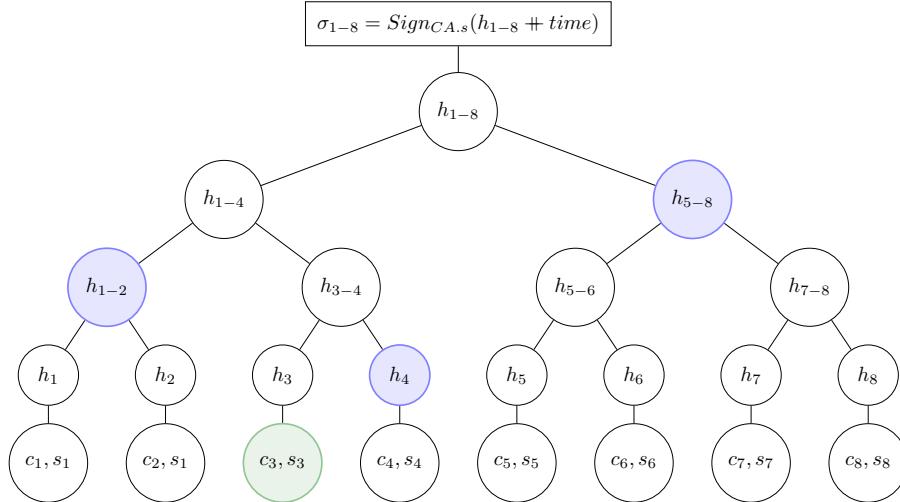


Figure 8.17: Certificates-Merkle-tree variant of OCSP: optimizing OCSP response, by signing the digest of a Merkle-tree whose leaves are the certificates c_i and their statuses $s_i \in \{\text{good, revoked, unknown}\}$ (Section 3.9). The root is the signature over the hash-tree and the time. Every internal node is the hash of its children; in particular, for every i holds $h_i = h(c_i, s_i)$, and $h_{i-(i+1)} = h(h_i \# h_{i+1})$. To validate any certificate, say c_3 , provide the signature of the certificate hash-tree, i.e., σ_{1-8} , the time-of-signing and the digest scheme’s *Proof-of-Inclusion (PoI)*, i.e., the values of internal hash nodes required to validate the signed hash, namely h_4 , h_{1-2} and h_{5-8} .

Signed Revocations-Status Merkle-Tree. We can further significantly reduce the overhead of OCSP, by using a Merkle-tree of *revocations status* instead of a Merkle-tree of certificates. This Merkle tree will still contain one leaf per certificate. However, the value of leaf i will be a *bit* b_i , corresponding to the revocation of certificate i ; i.e., $b_i = 1$ if certificate i is revoked, and $b_i = 0$ otherwise.

The CA applied the Merkle-tree scheme to these leafs, and obtains the digest of the entire tree of revocations, which it signs. To provide the OCSP response for a query to the status of a particular certificate, say certificate i , the CA includes in the response this signature, together with a *Proof-of-Inclusion (PoI)* of the value b_i as the i^{th} leaf of the tree.

This PoI can be further optimized by observing that revocations are not very common, i.e., most leafs will be zero (not revoked). There is no need to include the hash of any subtree whose leaves are all zero (i.e., none of the certificates in it was revoked).

Revoked-certificates Merkle tree. The disadvantage of the revocation-status approach is that it provides information about *all* certificates. Assuming that the only a very small fraction of the certificates are revoked, other op-

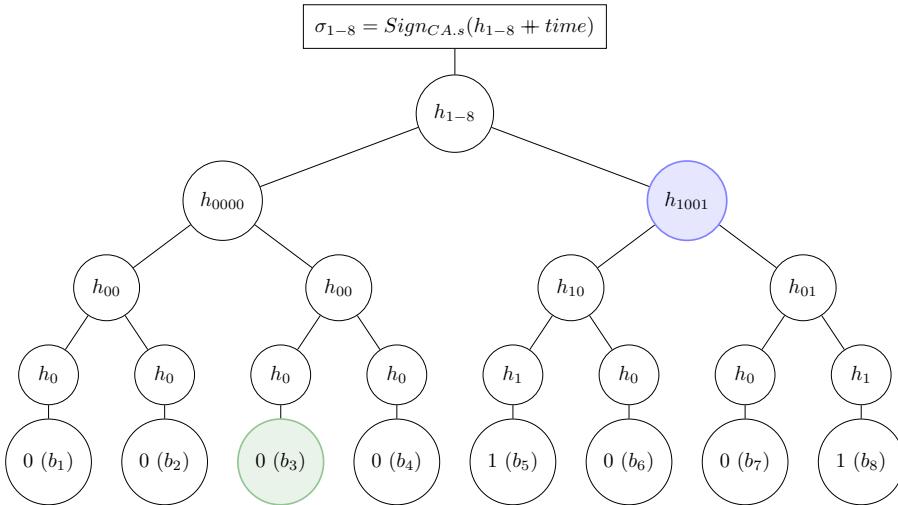


Figure 8.18: Signed revocations-status Merkle-tree; leaf i contains bit b_i which is 1 for revoked certificate, zero for non-revoked. The Proof-of-Inclusion (PoI) process is optimized by taking advantage of the fact that leaves contain only a bit (0 or 1), and that most certificates are not revoked, i.e., most leaves are zero. Hence, to prove that $b_3 = 0$, i.e., that certificate number 3 was not revoked, it suffices to send h_{1001} , together with the signature on digest of the tree h_{1-8} (and a timestamp). Recipients can precompute the value of intermediate node such as h_{0000} . Further optimizations may be possible, e.g., send 1001 instead of sending h_{1001} , and have client compute h_{1001} by itself. Optimizations should be carefully evaluated for their impact on computation time of the OCSP servers and clients, as well as their impact on bandwidth.

timizations are possible - and possibly even more effective. For example, we present the *revoked-certificates Merkle tree* approach. This approach is also interesting since it introduces an additional optional mechanism for Merkle digest schemes: a *Proof of Non-Inclusion (PoNI)*.

A revoked-certificates Merkle tree is a digest applied to all *revoked certificate-identifiers, sorted* by certificate identifier (number)⁹. Since the tree is sorted, it can provide *efficient proof of non-revocation* of a certificate. For example, assume we use certificate serial numbers to identify certificates, both in OCSP requests, and as the key for sorting the revoked identifiers hash-tree. Assume that an OCSP query contains a single certificate serial number, say i . The OCSP response will include the signed revoked-certificates hash tree, together with:

If i was revoked: proof of inclusion of i in the tree, similar to the one illustrated in Figure 8.17.

⁹Unfortunately, this requires certificates to be mapped to sequential (consecutive) numbers, which is often not the case

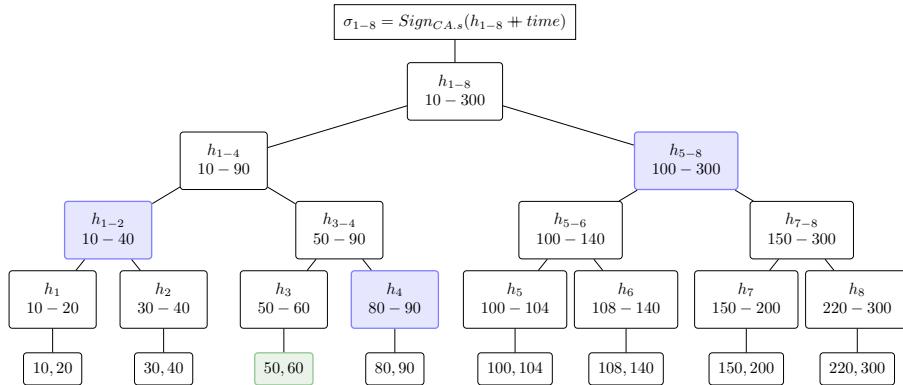


Figure 8.19: Optimizing OCSP response, using tree-of-revocations and Proof-of-Non-Inclusion. Each leaf contains serial numbers of revoked certificates; leafs are ordered by certificate serial numbers, assumed here to be consecutive. In this example, we prove that there is no revoked certificate between 51 and 59; in this special case, it suffices to present a PoI for the leaf containing 50, 60, since any value between 50 and 60 should have appeared between these two (values are sorted). The ranges are included with the hashes of internal nodes, to provide PoNI for values which fall between nodes, e.g., PoNI for a value between 61 and 79, which will use the ranges indicated with h_3 and h_4 . The ranges also prevent an intentionally misleading tree structure which allows both PoI and PoNI to the same value. Details omitted.

If i was not revoked: proof of inclusion of i' and i'' in the tree, where $i' = \max\{i' < i \wedge i' \text{ was revoked}\}$ and $i'' = \min\{i'' > i \wedge i'' \text{ was revoked}\}$. Since the identifiers of revoked certificates are *sorted*, this provides proof that i was not revoked. Namely, we take advantage of a *Proof of Non-Inclusion* (*PoNI*).

Exercise 8.9 (Revoked-certificates hash-tree OCSP-variant). *Following the brief outline in the previous paragraph:*

1. *Design and analyze an improved OCSP-variant using hash-tree of revoked certificates.*
2. *Extend your design to also incorporate the hash-chain technique (Exercise 8.20).*

Hint: see Figure 8.19.

Hash-chain OCSP. Finally, Exercise 8.20 discusses an additional possible optimization of OCSP responses, using hash-chain. The goal is to avoid *any* signature in the typical case that the certificate is *not revoked*.

8.4.5 Optimized Periodic Revocation Status: OneCRL, CRLsets and CRVs

The large overhead of CRLs, esp. when downloaded periodically for all CAs, motivates *optimized* Periodic Revocation Status mechanisms. One simple approach is to prefetch a *subset* of the revocation information, focusing on revocation information of ‘important’ certificates; this essentially follows the ARL *CRL* extension approach, mentioned above.

Currently, such optimizations are deployed by several major browsers, albeit in proprietary manner. This includes Google Chrome’s *CRLsets* and Mozilla’s *OneCRL* [147]. Both CRLsets and OneCRL prefetch revocation information only for a small subset of certificates, mostly CA certificates; in 2015, Liu et al. reported in [328] that only 0.35% of the revoked certificates were reported in CRLsets. The revocation information is collected by the vendor of these browsers, and retrieved by the browser from the vendor, typically daily, typically as part of the daily software update process.

Several recent papers [208, 299] also investigated optimized prefetch mechanisms, which may allow prefetching to extend to much larger collections of certificates - with acceptable overhead. Let us describe the Certificate Revocation Vector (*CRV*) design, proposed by Smith et al. [299].

Certificate Revocation Vector (*CRV*). The Certificate Revocation Vector (*CRV*) is a bit vector which contains ‘1’ for revoked certificates and ‘0’ for non-revoked certificates. Most bits in the *CRV* would normally be ‘0’ (not revoked), hence, it is transmitted and used in compressed form. If certificates are identified by sequential numbers, then, even without compression, CRV may be significantly more efficient than *CRL*, when a significant fraction of the certificates is revoked (e.g., 8%).

Compression makes CRV significantly even more efficient, esp. for typical, low revocation percentages. For example, consider the following simple compression method. Let V be a vector of c entries (certificates) with r bits turned on (and the rest of the bits, $c - r$, turned off). Instead of storing or transmitting V as c bits, we can represent it by the list: $\{c_1, c_2, \dots, c_R\}$, where c_1 is the number of ‘zero’ entries in V until the first entry, and c_i is the number of ‘zero’ entries between the i^{th} ‘one’ bit in V and the $(i + 1)^{th}$ ‘one’ bit in V . Such list can be encoded by at most $r \cdot \log c$ bits (in the worst case, we will need $\log c$ bits for each entry). When the percentage of revoked certificates, i.e., r/c , is small, as is the common case, then $r \cdot \log c \ll c$. Real compression schemes are even more efficient.

Furthermore, CRVs can be updated even more efficiently. Let $CRV(d)$ denote the value of the *CRV* of period (often, day) d . To update from $CRV(d-1)$ to $CRV(d)$, we can distribute the *Delta CRV*, which is simply the bit-vector which contains ‘1’ only for certificates which were revoked *during period (day) d*. Namely:

$$CRV(d) = CRV(d-1) \oplus \Delta CRV(d) \quad (8.1)$$

The ΔCRV vector is usually *very* sparse, therefore compresses *very* well, as the number of revocation in day d , denoted r_d , is usually not more than few dozens, resulting in very short $\Delta CRVs$ (of $r_d \cdot \log c$ or so). Even the peaks of daily revocations, as due to the Heartbleed bug, were not much more than 10,000 revocations, which still results in fairly short compressed ΔCRV [77, 328]. For more details on the CRV mechanism, see [299].

Unfortunately, in practice, certificates are usually not identified by sequential numbers; in fact, the best practice is that certificate serial numbers are random 20-bytes (159 bits) positive integers (subsection 8.2.3). We could, of course, use CRVs for the space of $c = 2^{159}$ serial numbers; however, then we have $\log c = 159$, making compressed CRVs and delta CRVs significantly longer ($159 \cdot r$ or $159 \cdot r_d$, respectively).

8.5 Web PKI: Vulnerabilities and Improvements

In this section, we discuss on the application of PKI techniques to protect web communication, i.e., Web PKI, as deployed by popular browsers. In subsection 8.5.1 we discuss failures of Web PKI certificate authorities, and their causes; some major failures are summarized in Table 8.5. Then, in subsection 8.5.2, we discuss defenses against CA corruptions and failures. In the next section, we focus on the most ambitious defense - the *Certificate Transparency (CT) PKI*, which is a significant extension of X.509/PKIX, and adopted by some of the most popular browsers as well as by many CAs and websites.

8.5.1 Web PKI Certificate Authority Failures

Ideally, a certificate could be issued only by one of a few, highly trusted CAs, with a clear focus and ability to provide secure certification service for a specific name space, and preventing any misleading, equivocating or unauthorized certificates. However, the reality, at least in the case of Web PKI - the main application of PKI so far - is very different. In particular, browsers appear to trust too many CAs (and hence, certificates signed by them). For example, a study from 2013 [111] found 1832 browser-trusted signing certificates, each representing a trusted CA (mostly, intermediate CAs). Out of these, 40% were academic institutions, and only 20% were commercial CAs. Only seven of these CAs were restricted to a particular name-space (name constraints), and most did not have any length constraints either.

The following seem to be the main weaknesses of the classical Web PKI system:

Rogue and negligent CAs: browsers, the most common type of relying party, are distributed with a list of many - typically, around a hundred or more - ‘root CAs’, i.e., trust-anchor CAs. The reason for this excessively-long list may be that browser manufacturers do not want to take the responsibility for judging the trustworthiness of different CAs, a

2001	VeriSign issues false Microsoft code-signing certificates [125].
2008	Researcher Mike Zusman shows validation failures of Thawte (issues unauthorized certificate for Microsoft's Live.com) and StartSSL [330].
2008	CertStar, a reseller of Comodo, issued certificates without <i>any</i> validation (demonstrated for Mozilla.com). [218]
2011	Comodo not performing domain validation [249].
2011	DigiNotar compromised, 531 rogue cert's (discovered); a rogue cert' for *.google.com used for MitM against 300,000 Iranian users [322].
2011	TurkTrust issued intermediate-CA certs to users, abused to issue certificate for *.google.com (detected on Dec. 2013). [204].
2012	Trustwave issued intermediate-CA certificate for eavesdropping (in enterprises) [269].
2013	ANSSI, the French Network and Information Security Agency, issued intermediate-CA certificate to MitM traffic management device [325].
2014	India CCA / NIC compromised (and issued rogue certs) [206].
2015	CNNIC (China) issued CA-cert to MCS (Egypt), who issued rogue cert's [115, 242].
2013-17	Audio driver of Savitech installed root CA in Windows [317].
2015-17	Symantec issued unauthorized certs for over 176 domains [255].
2019	Kazakhstan forces all its web users to (manually) install its own root CA (Qaznet), to obtain MitM capabilities <i>customer-installed</i>) [267].
2019	Mozilla, Google revoke intermediate-CA of surveillance firm DarkMatter [52], and refuse to make it a root-CA.
2020	Let's Encrypt detected a bug in their CAA-validation code, affecting over 3 million certificates [1].
2022	TrustCor, a root CA for common browsers, exposed as related to Spyware [229].

Table 8.5: Some notable Web PKI Certificate Authority Failures. Several of the CAs mentioned, and many others, were removed from the root CA programs; e.g., see Mozilla's list of removed root CAs [241].

role that may imply liability and potential anti-trust concerns. Indeed, the role of a browser is to provide functionality, not necessarily to represent the trust of individual users; ideally, each *user* should determine which CAs are ‘trust anchors’. Browsers typically provide a user interface allowing users to edit the list of trusted CAs; however, it is not realistic to expect users to really use this and modify the default list of trusted CAs, and while we are not aware of this being measured yet, we are quite sure that essentially nobody is doing that. The bottom line is that some root-CAs may not be sufficiently careful (negligent CAs), or, worse, may be subject to coercion or otherwise under control of attackers (rogue CAs). Note that such attacks may be done intentionally by nation-states, which may mean that a CA may have to comply and cooperate. Unfortunately, the legal and business challenges do not leave significant hope for a change that will ensure trustworthy CAs, and/or reduce the number of root CAs. For few examples of notable CA failures, see Table 8.5 and the cited sources.

No restrictions on certified names/domains: not only are there (too) many ‘root’ (trust-anchor) CAs, but, furthermore, any root-CA may certify any domain, without any restrictions. Furthermore, TLS/SSL certificates issued by root-CAs to intermediate CAs rarely include ‘name constraint’. Without ‘name constraint’, intermediate CAs can publish certificates for all domains. A name constraint could have restricted the damage from a rogue intermediate CA. One reason that name constraints are rarely applied is that, currently, every domain-name-registrar may allocate unallocated top-level and second-level domains; it would be hard to justify limiting the domains that a given CA can certify.

Insufficient validation, esp. of Domain-validated certificates: to ensure security of certificates, it is crucial for the certificate authority to properly authenticate requests for certificates. However, proper authentication requires costly, time-consuming validation of the requesting entity and its right to assert the requested identifier (typically, domain name). To reduce costs and provide quicker, easier service to customers (requesting certificates), many certificate authorities use automated mechanisms for authentication, often based on the weakly-secure *domain validation*; see subsection 8.2.8.

Equivocation: X.509 does not include any mechanisms to prevent or detect *certificate equivocation*, i.e., existence of multiple different certificates for the identifier. For the Web PKI, a MitM attacker can impersonate a TLS/SSL website *bob.com*, if it is able to obtain an equivocating certificate for *bob.com*. For CAs that provide domain-validated certificates, the defense against equivocating certificates is usually merely the weak security of the domain-validation process (subsection 8.2.8).

Misleading domain names: there are no standard requirements for validation of domain names by CAs or domain registrars. Such validation would

help to prevent certificates for *misleading* domain names, typically used for *scam* and *phishing* campaigns; see subsection 8.1.1. Some CAs perform some validation, however, as there are also many less-reputable CAs, obtaining such a domain name, and then a (domain-validated) certificates is still not very challenging.

Ineffective Rogue-Certificates Detection and Accountability. X.509 ensures accountability - but only in the sense that once a rogue certificate is *detected*, it is possible - even easy - to identify the ‘culprit’ CA. However, X.509 (and Web PKI) does not provide any mechanism to *detect* the rogue CA. Of course, detection is *possible*; for example, the Google Chrome browser has built-in mechanisms that detect receipt of a rogue certificate for the Google domain, which were invoked in the DigiNotar and other incidents (Table 8.5). But this is an exception that proves the rule; if we could rely on browsers knowing the public keys, we wouldn’t need certificates at all! This specific weakness is addressed by the Certificate Transparency (*CT*) mechanism, which we discuss in Section 8.6.

8.5.2 X.509/PKIX Defenses against Corrupt/Negligent CAs

The naïve view of PKI security is that certificate authorities are fully trusted, honest entities, that never fail to operate correctly, and in particular, to carefully vet any request for a certificate. However, in view of the known failures (e.g., Table 8.5) and the weaknesses outlined above, a different approach seems advisable: PKI systems should ensure some security guarantees, even assuming that some CAs may be corrupt or negligent.

For example, a basic property that public-key certificates ensure is *accountability*. Namely, each certificate, simply since it is *signed*, is an evidence to the fact that the certificate was signed (issued) using the private-key corresponding to a given public key, i.e., identifying the signing CA. Namely, once we find a rogue certificate, properly signed by some CA, i.e., that validates as ‘correct’ using the public key of a CA, the CA becomes *accountable* for this certificate. Such *accountability* can be viewed as a necessary requirement¹⁰ for a secure PKI system.

Indeed, considering the reality of possibly *corrupt or negligent CAs*, there are additional proposed and deployed defenses against fraudulent certificates - beyond the basic CA accountability, directly assured by the signed certificate. The most significant in these proposals and efforts is probably *Certificate Transparency*, which provides a public, auditable log of certificates; we discuss it in the next section. Below, we discuss several other proposed and deployed defenses.

¹⁰It is possible to define the accountability requirement, as well as other security requirements from PKI schemes, similarly to the definitions presented in earlier chapters for different cryptographic schemes such as encryption. Such definitions allow provably-secure PKI schemes; for more details on this approach, see [214].

Use name constraints. One possible improvement to the security of the Web PKI system is simply to *adopt and deploy* the certificate path constraints already defined in X.509 and PKIX, most notably, the *name constraint*, discussed in subsection 8.3.3. This would allow restriction of the name space for certificates issued by a given CA, e.g., based on the DNS *top-level domain country codes (TLDcc)*, as defined in RFC 1591 [264]. For example, a Canadian CA may be restricted to the TLDcc for Canada (.ca). In particular, this would prevent the government in a different country, from coercing a CA she controls into issuing an (unauthorized) certificate for a domain name in the .ca domain.

This idea is simple and is based on a standard extension; however, there are significant deployment concerns. Let us present two major concerns. First, the use of name constraint requires a major change in the existing Web PKI system, which would be hard to enforce. Second, there does not appear to be a natural way to place name constraint on domain names which are in one of the generic top level domains (gTLD) - and most domains belong to a gTLD, mainly *com*, *org*, *gov*, *edu* and *biz*, not to mention that new top level domains can be defined essentially by anyone. Restricting the certification of gTLD certificates to only specific CAs may result in significant objections by other parties (and countries).

Public key and/or certificate pinning. Fraudulent certificates can be foiled and/or detected by the relying party, typically the browser, when the browser is aware of the correct public key or certificate. For example, the Chrome browser had the public key of Google ‘burned-in’, allowing the browser to detect a fraudulent public key for Google (see the DigiNotar incident in Table 8.5). Such ‘burned-in’ public keys are usually referred to as *static pinning* (or static key pinning).

Dynamic key/certificate pinning and HPKP. Static pinning is problematic, since keys do change *sometimes*, e.g., due to unexpected revocations. Furthermore, static pinning is based on decisions of the browser’s vendor, which may not be relevant to the usage of a particular user. This motivates the use of *dynamic pinning*, where the *website* specifies the pinning of the public key/certificate, for limited time. *Dynamic pinning* of public keys or certificates allows secure sites to direct the browser to ‘pin’ the given public key or certificate to the domain, for a specified period; see specifications for *HTTP public key pinning (HPKP)* in RFC 7469 [119].

HPKP, and dynamic key/certificate pinning in general, would foil and detect fraudulent keys/certificates, even if attacker obtains a rogue certificate properly signed by a CA trusted by the relying party. With dynamic pinning, the server essentially declares *I will use this PK (or certificate) for at least the specified period*. The client remembers this indication and refuses the use of other keys/certificates, for the specified period.

In addition to, or instead of, *refusing* ‘other’ keys, the site may request clients to report cases where they receive a key or certificate which differs from

the pinned one, i.e., use this mechanism for *detection*.

However, there are drawbacks to the use of key/certificate pinning, including:

1. If the private key is lost to the server, e.g., it is only kept on the server on a disk which fails, then there is a risk of losing the ability to serve the website, until the pinning times-out. To address this (and the next) concern, a backup copy of the secret key must be kept by the owner of the website in secure, reliable storage.
2. Suppose that an attacker penetrates the server's site, exposes the private key *and* corrupts the copy of the key used by the server. This may allow the attacker to serve the site to unsuspecting users; but, if the attacker now also uses HPKP to indicate key pinning, it can *prevent* the legitimate server from changing its key. Note that this problem may exist even for a site that does not use HPKP for key/certificate pinning, since the pinning is performed by the *attacker*. If the key is also lost, this allows the attacker to prevent access to the legitimate site - basically, a *Denial-of-Service (DoS)* attack, and to blackmail the owner of the website.
3. Revoking a pinned key is difficult, since a replacement key is typically not pinned. One option is to wait for the pinning to time-out, but that implies period of unavailability of the site. An alternative is for the implementation to cancel key pinning once the certificate is revoked; however, the current specification does not require this behavior.
4. Key-pinning may be applied only for long-lived public keys, and may interfere with the use of rotating keys, and to prevent the use of ephemeral master keys, e.g., to recover security and ensure perfect forward secrecy (PFS).

Due to these disadvantages or for other reasons, the use of HPKP (or other methods of key/certificate pinning) is not common.

CA, CT and (must-staple) policy pinning. Pinning can be also applied to other security-related properties, in particular, to the allowed certificate authorities (CAs) and to the use of *Certificate Transparency (CT)*, which we discuss in the next section (Section 8.6), or of other security policies, most notably, *must-staple*. In particular, a web-site can signal that it always use CT certificates, by including the Expect-CT HTTP header [300], thereby ‘pinning’ the use of CT certificates. A similar TLS extension could allow CT pinning for non-web (i.e., non-HTTP) TLS servers.

It would similarly be useful to ‘pin’ the OCSP-must-staple policy. This would foil a rogue certificate issued without OCSP-stapling, which is important, since if such certificate is revoked, a MitM attacker can still use it by dropping the OCSP request or response, in the (common) case where the client soft-fails on OCSP timeout, except for ‘must-staple’ certificates. See Exercise 8.18. Similarly, there may be a value in *CA pinning*, i.e., pinning of the identity of specific

certificate authorities which the website would use (say, in the coming year), foiling attacks using certificates signed by rogue or negligent CAs. However, certificate and CA pinning can also be provided by the *CAA* mechanism, which we describe next.

DNS-based key, certificate and CA binding: DANE and CAA. Another approach to the concern of corrupt and negligent CAs, is to use an *alternative* set of trusted third parties. Specifically, this approach takes advantage of the *Domain Name System (DNS)*, which maps domain-names to the corresponding IP address and other information. The DNS system is provided by a hierarchy of *name servers*, and can be protected using the *DNSsec* extension [11].

Two standardized DNS-based mechanisms are defined: *DANE* (DNS-Based Authentication for TLS) [170] and *CAA* (*Certificate Authority Authorization*) [156]. *DANE* provides an alternative way to validate the public key of a given domain, complementing or providing an alternative to PKIX certificates. *CAA* allows *CA-binding*, i.e., allows a domain-owner to define the set of CAs allowed to issue certificates for the domain. Both mechanisms depend on the *DNSsec* standard [11] for signature-based security, in particular, against MitM attackers.

The Trust-On-First-Use (TOFU) Adversary model. Unauthenticated dynamic pinning, of security-policies, CAs or certificates/keys, is based on the hope that *the first connection is secure*. We refer to this assumption (or hope) as the *Trust On First Use (TOFU)* adversary model; assuming this adversary model provides a way to establish security, without relying on trusted third parties (such as a CA), or as an additional security mechanism for the case where a trusted party fails, as when using HPKP. For example, TOFU can be used instead of the use of certificates altogether, by having the parties, in their first connection, exchange non-certified public keys, or use the DH key exchange to establish a shared secret key; and then use these established key(s) to secure future connections.

8.6 Certificate Transparency (CT)

Certificate Transparency (CT) [151, 209–211, 213] is a proposal, originating from Google, which significantly improves the transparency and security of the Web PKI - and of PKI in general. CT is backward-compatible with PKIX, i.e., a *CT-certificate* is compatible (acceptable) by relying parties using ‘regular’ PKIX. There is an extensive standardization, implementation and deployment effort for CT, and a variant of CT is already enforced by popular browsers, including Chrome, Edge and Safari. As a result, many websites and CAs deploy CT, making CT the most important development in PKI since X.509. However, CT is still an evolving technology, including two (significantly different) draft

standards [210,211], several implementations (which often differ from both draft standards), and several proposed variants.

In the rest of this section, we discuss three variants of CT: *HL-CT (Honest-Logger Certificate Transparency)*, which roughly corresponds to CT as currently deployed; *AnG-CT (Audit-and-Gossip Certificate Transparency)*, which uses auditing to provide some defense against rogue loggers; and, finally, *NS-CT (NTTP-Security Certificate Transparency)*, which completely defends against rogue logger. The term *NTTP* stands for *No Trusted Third Party*; NTTP was actually mentioned among the CT design goals [151,205,209]. The HL-CT and AnG-CT variants are close to the CT specifications of [210,211], while NS-CT is a simplification of the design of [213].

We begin with some basics which are common to all variants of CT.

8.6.1 CT: concepts, entities and goals

In this subsection, we present the main concepts, relevant entities (mainly, loggers and monitors), and, most significantly, goals of CT. But first, let us analyze the problem - what is missing in X.509?

Analysis: what is missing in X.509? The basic goal of X.509 is *accountability*, i.e., the ability of relying parties to identify the issuer of a certificate. Accountability allows a relying party to decide which certificates to trust, e.g., by checking if the issuing CA is one of the CAs it trusts ‘directly’ (trust anchor or root CA). But let’s focus on the other use of accountability; as the term implies, accountability ensures that a CA can be held *accountable* for issuing incorrect or rogue certificates. In Web PKI, this mostly implies that a negligent or rogue CA could be distrusted - in particular, it may be removed from the *root programs*, i.e., not be a root CA anymore, and it could be revoked, if it was an intermediate CA. See Table 8.5 for some examples.

Accountability is important, to eliminate rogue or negligent CAs, and to motivate CAs to ensure their security and reliability, i.e., to *deter* rogue or negligent operations. However, *deterrence is not* fully achieved by the X.509 accountability, since X.509 does not provide mechanisms for *detecting* rogue or incorrect certificates. Of course, a relying party that receives a rogue certificate could publish it; but this certainly is not a viable method of detection - indeed, if relying parties would be able to identify rogue certificates on their own, they wouldn’t need the certificate in the first place!

Indeed, it was only by the insight of Google engineers that included in Chrome a check for the public key received for Google, that allowed Google to detect the 2011 MitM attack on Iranian users (Table 8.5) and few other attacks. Over 300,000 users were attacked, and *not a single user* detected the attack; in particular, users using other browsers, that did not include this non-standard check, did not detect it. This incident apparently was a significant motivation to Google’s Certificate-Transparency project.

CT: new Functions and Entities. Certificate Transparency introduces several new functions to the PKI ecosystem:

Logging. Logging is the basic function of CT: provide public logs of certificates. This function is performed by a new PKI entity, naturally called a *logger*. A CA has to send each certificate to a logger, who adds the certificate to its log, and provide the CA with a ‘proof’ that the CA can present with the certificate, allowing relying parties to know that the certificate was logged (by that logger). This ‘proof’ is what turns the certificate into a CT-certificate. As a result, the logger becomes *accountable* for the public availability (transparency) of this certificate.

Monitoring. Monitoring is the process of inspecting logs, mainly to detect suspect certificates. For better efficiency and flexibility, monitoring is not performed directly by the loggers; instead it is done by another new PKI entity, the *monitor*, each typically monitoring the logs of multiple loggers. Typically, monitors are contracted by a domain owner, to alert the domain owner of any ‘suspect rogue’ certificates, which may mislead people trying to access the owner’s service (usually, website). Following the discussion in subsection 8.1.1, monitoring may be for new certificates with the same domain name (equivocating), for a domain name which includes a given trademark or specific term (Combo), for domain names which appear to be potential ‘domain name hacking’, for domain names which may be homographic or typosquatting attacks, or for other ‘suspect’ domain names. If the domain owner does not fully trust a single monitor, it could use multiple monitors.

In NTTP-Security Certificate Transparency (NS-CT), monitors also assist relying parties to detect misbehaving loggers and monitors, and to disseminate information about rogue loggers and monitors.

Auditing. A CT-certificate is a *commitment* of the logger to add the certificate to the log, and, if and when requested, provide it to the monitors. Auditing is the process that validates that the logger indeed added a given CT-certificate to the log, and provided it to the monitors if requested. Early CT documents [151, 209] mention another new PKI entity to perform the audit function, the *auditors*; however, the auditor functionality was not fully specified, and [151] says most auditors will likely be built into browsers. The current CT specifications [210, 211] provide limited auditing functionality, and current relying party implementations (in Chrome and other browsers) do not implement auditing at all. We include auditing, only by the relying parties, in AnG-CT and NS-CT, and omit it in HL-CT.

Periodic Gossip and Updates. CT involves periodic dissemination of information, such as of logs and *digests* of logs. This dissemination is done by *periodic updates* (from logger to monitor) and, in NS-CT and AnG-CT, by *periodic gossiping* between monitors.

CT Goals and Variants. One of the main weaknesses of the current Web PKI, discussed in subsection 8.5.1, was that it is ineffective in detection of rogue-certificates. This also implies that its accountability guarantee is not effective: to identify the failure of the CA, we must find and detect the rogue certificate - but a rogue certificate may not be detected, especially if used in a limited way. *Effective accountability requires transparency!*

The basic goals of CT are to provide this missing transparency. Namely, CT is designed to make it easy to:

Detect equivocating certificates by monitoring specific domain name:

CT allows detection of certificates issued for a given domain name (or, in general, identifier). This allows the ‘owner’ of the domain to detect unauthorized equivocating certificates issued to its domain name.

Detect suspect certificates: CT also allows monitoring of *all* certificates issued - by a particular logger, multiple loggers or all loggers. This allows the monitors and domain-owners to detect other types of suspect certificates, such as the misleading certificates discussed in subsection 8.1.1, as well as certificates not conforming with different specifications (that should be enforced by CAs and loggers).

Detect suspect CAs: by monitoring all certificates issued by a CA, which may require monitoring all loggers (used by this CA), it becomes possible to detect suspect CAs.

The basic approach allowing CT to achieve these goals is simple: *every certificate issued, should be included in a publicly-available log*. One way to implement this idea would have been for each CA to maintain such publicly-available log of the certificates it issued. However, recall the *risk of rogue and negligent CAs*, also listed in subsection 8.5.1 as a major Web PKI weakness. Indeed, our last goal was to allow efficient detection of suspect CAs; we clearly cannot rely on the CA itself to provide us with the necessary information.

Due to this concern about non-trustworthy CAs, the CT designers separated the log functionality to a new type of entity, called a *logger*. The CA must send each certificate it issues to the logger, who provides back a *signed response*, which should be attached to the certificate, turning it into a *valid CT-certificate*.

In HL-CT, NS-CT and the CT specifications [210, 211], the response is a commitment by the logger to include the certificate in the log by specific time; this commitment is referred to as an *SCT (Signed Certificate Timestamp)*. In contrast, in NS-CT and [213], the response consists of the *STH (Signed Transparency-tree Head)*, which is a timestamped digest of the log, and a *PoI (Proof-of-Inclusion)*, proving that the certificate is included in the log.

Does this mean that the trust of the user now simply moved from the CA to the logger? The original CT documents [151, 205, 209] clearly have a more ambitious goal: *no trusted third party*. In particular, the loggers are not required to be trusted. Indeed, that’s clearly the motivation for the use, in CT, of the audit and gossip mechanisms.

So, is this ambitious goal of *no trusted third party* achieved, and how? This depends on the exact CT variant. In each of the three following subsections, we describe one of these variants, as follows:

Honest-Logger Certificate Transparency (HL-CT): the HL-CT design assumes an honest logger; a CT certificate simply consists of a ‘regular’ X.509 certificate, signed by a CA, together with an SCT, signed by a trusted logger. A relying party may require the use of multiple loggers to achieve some protection via redundancy. This is quite close to the CT deployment by browsers, which mostly require, indeed, some redundancy (multiple SCTs by different loggers). This also roughly corresponds to the logger mechanisms described in [210, 211]; indeed, [211] explicitly mentioned the assumption of honest logger. Note, however, that both documents also mention auditing and gossip, which are not required assuming honest logger. Therefore, the HL-CT design is a simplification, which only presents the mechanisms required assuming honest loggers.

Audit-and-Gossip Certificate Transparency (AnG-CT): the AnG-CT design adds audit and gossip mechanisms to HL-CT, with the goal of mitigating rogue loggers; this roughly follows the ideas in [151, 209–211], but since some details were missing in these publications, we tried to extrapolate these details. As we show later, AnG-CT achieves only *partial* protection against rogue loggers, and results in a privacy-exposure: the logger can learn, essentially, which websites the client visited.

NTTP-Security Certificate Transparency (NS-CT): the NS-CT design presents a different way for performing auditing and gossip, which ensures full protection against rogue loggers, and also protects the privacy of the relying-party.

8.6.2 The Honest-Logger Certificate Transparency (HL-CT) design

The Honest-Logger Certificate Transparency (HL-CT) design focuses on ensuring transparency - assuming an honest logger. The logger has three basic functions:

Log certificates submitted by CA: the logger receives certificates sent to it by a CA, typically during their issuing process, and returns a signed, time-stamped receipt, which is called the *Signed Certificate Timestamp (SCT)*. This process is illustrated in Figure 8.20, and the *SCT* is calculated as: $SCT_B \equiv Sign_{L.s}(C_B \# date)$ where C_B is the certificate received.

Validate certificates: both logger and CA are required to uphold to industry standard regarding the certificates; the specific standards may be identified, e.g., in the certificate-policies extension. The logger should provide the *SCT* only after it confirms that the certificates conform with the standards.

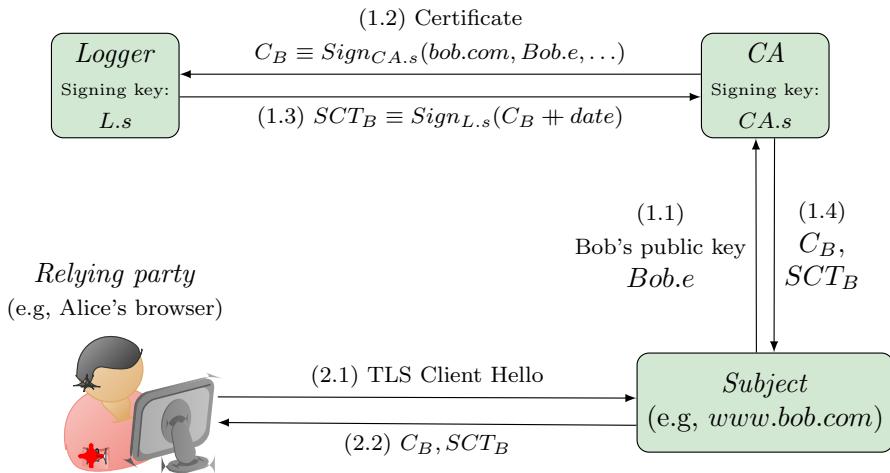


Figure 8.20: The issue process for Honest-Logger Certificate Transparency (HL-CT) and Audit-and-Gossip Certificate Transparency (AnG-CT), flows (1.1)-(1.4), and first two flows of typical TLS server authentication, flows (2.1)-(2.2). The CA sends every newly issued certificate to one or multiple loggers (only one showed). A logger receiving a certificate, validates it conforms with required standards, and sends back an *SCT* (*Signed Certificate Timestamp*), SCT_B , which is a signature over the certificate C_B and the current time. The CA provides both certificate and *SCT* to the subject (e.g., website), who sends them to the relying party, e.g., as part of the TLS handshake. The relying party validates the signature on the *SCT*, in addition to the regular validation of the X.509 certificate. In AnG-CT, the relying party may also perform *auditing*; see subsection 8.6.3.

Provide log entries: loggers provide log entries upon requests from monitors (or, possibly, auditors). The protocol may allow requests for specific ranges of log entries, request for entries from a particular data/time, or for report of the current range of log entries.

This monitoring process of HL-CT is illustrated in Figure 8.21. The figure shows certification of the ‘real’ certificate of domain BOB.COM, and of a rogue equivocating certificate for the same domain, which includes the public key *Mal.e* of the attacker (*Mal*). We simplify the interaction between the monitor and the website, and assume that the (legitimate) website ‘subscribes’ with the monitor for updates on certificates issued for the domain BOB.COM. As a result, the monitor informs the websites of the two certificates as soon as the monitor is informed about them by the logger.

The Honest-Logger Certificate Transparency (HL-CT) scheme may fail to

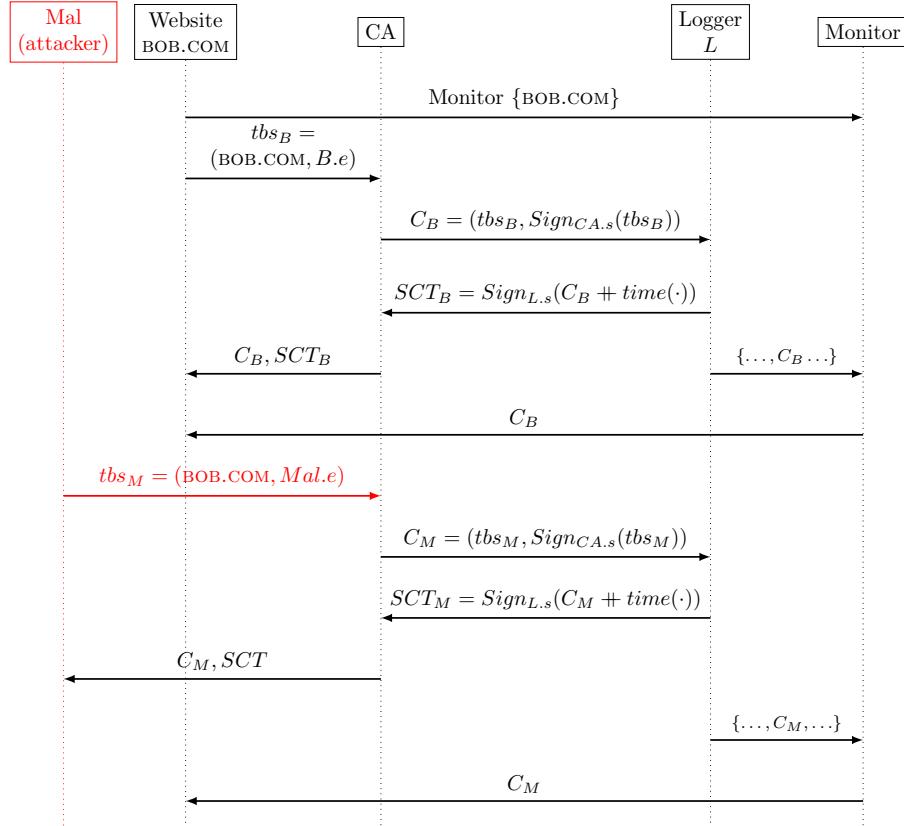


Figure 8.21: Monitoring in Honest-Logger Certificate Transparency (HL-CT). Website BOB.COM requests monitoring of its domain (BOB.COM) from a monitor, and requests from a CA, a certificate C_B binding its domain BOB.COM with its public key $B.e$. The CA first logs the certificate with the logger L , who sends back a signed SCT (SCT_B). The logger periodically updates the monitor with all newly issued certificates - including C_B ; and the monitor accordingly updates its client, the website BOB.COM. The figure then shows attacker Mal receiving rogue certificate C_M , binding BOB.COM to Mal 's public key, $Mal.e$. Once the logger updates the monitor about the issuing of C_M , the monitor alerts the legitimate domain owner BOB.COM, i.e., the rogue certificate - and the failure of the CA - are made public.

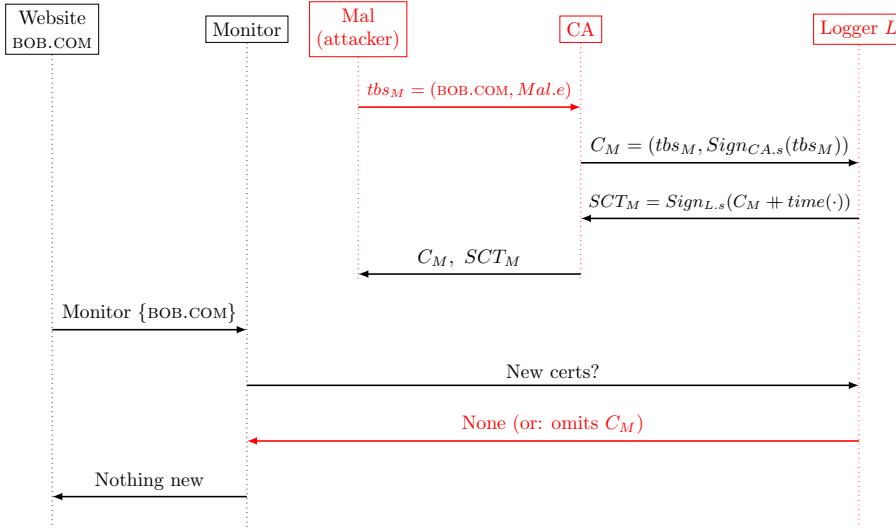


Figure 8.22: Omitted-certificate attack by a rogue logger and rogue CA. The attack is effective against Honest-Logger Certificate Transparency (HL-CT), since HL-CT does not include mechanism to detected that a certificate was omitted from the log.

ensure transparency, if using a rogue, dishonest logger. Specifically, a rogue logger may simply fail to report a rogue certificate to the monitor(s). We refer to this as an *Omitted-Certificate attack*, and illustrate it in Figure 8.22.

Security against Colluding Rogue Logger and Rogue CA. The omitted-certificate attack is a simple, practical attack against HL-CT, by a rogue logger colluding with a rogue CA. Such collusion is quite possible, since loggers are often operated by a CA - currently, this seems to be a common scenario, and from the operational and economical point of view, it seems likely to remain a common scenario. Indeed, it is not clear that the browser vendors will have justifications to be overly selective in their approval of loggers to be trusted by the browsers; it is quite possible that as CT becomes popular, most CAs will become loggers.

Once that happens, the additional security provided by HL-CT, compared to the ‘regular’ X.509 system, would be limited. HL-CT seems to offer little in terms of defense against intentionally malicious collusion of CA and logger.

To ensure security against rogue loggers - and their collusion with rogue CA - one may use the other variants of CT, AnG-CT or NS-CT, which we discuss in the following subsections. There is, however, also the option of using HL-CT, but requiring each certificate to come with *multiple SCTs*, signed by different loggers; namely, to provide additional security by *redundancy*: transparency is assured, as long as at least one of these loggers is honest.

Redundancy of Loggers. This is a simple approach to protect against a rogue logger: instead of requiring an *SCT* signed by an honest, trusted logger, we require, for each certificate, *multiple SCTs*, each signed by a distinct logger. Clearly, this prevents attack by any single logger. This is the approach deployed in practice by the popular Chrome browser.

Obviously, this solution involves some overhead - in the certificate issuing process, as well as in the validation and transmission of certificates. However, as long as the number of redundant loggers is kept fairly small, the overhead appears reasonable. In practice, browsers deploying CT, e.g., Chrome, mostly require only two loggers.

A security concern is that the CA may get to *choose* the two (or more) loggers, allowing a rogue CA to potentially choose colluding loggers. Possibly to deal with this concern, Chrome requires that Google's own logger be one of the loggers used. Therefore, the security of this mechanism seems to basically depend on the assumption that Google acts honestly. While other browsers could, theoretically, enforce the use of a different logger (not chosen by the CA), this would imply further overhead (mainly on the CA), and is unlikely to happen. Of course, we don't claim that Google would *not* act honestly, but requiring relying parties to essentially trust one specific 'third party' (Google), seems to contradict the original CT goal of *no trusted third party*, and could re-introduce concerns of coercion of Google, and the CA, by a government (in this case, the US). This motivates the use of AnG-CT or NS-CT, which include mechanisms to protect against a rogue logger, beyond simple redundancy.

Notice that for solutions that defend against a rogue logger, such as AnG-CT and NS-CT, there is still motivation for the CT to log certificates with multiple logger. Namely, this provides *resiliency*; even if one logger is detected to be rogue and distrusted, the certificate would remain valid since it was also logged by another logger (who is still trusted). Of course, this is a different motivation than that of using redundancy in HL-CT.

8.6.3 The Audit-and-Gossip Certificate Transparency (AnG-CT) design

Audit-and-Gossip Certificate Transparency (AnG-CT) extends HL-CT with additional mechanisms, designed to detect rogue loggers.

Issuing and Monitoring in AnG-CT. The CA-issuing process of AnG-CT, and the monitoring process of AnG-CT, are exactly the same as the corresponding processes of HL-CT, as shown in Figure 8.20 and Figure 8.21.

There are some differences, though, in the operation of the logger. The most basic change is that in AnG-CT, the logger is obliged to use a *Merkle tree* construction, which is applied to the entire log of certificates, as we next explain.

AnG-CT: use of Merkle Tree: STH , PoI , PoC and MMD . In AnG-CT, the logger is obliged to use a *Merkle tree*, which is applied to the entire log of certificates; see Section 3.9 for details about the *Merkle tree* construction.

The AnG-CT scheme makes extensive use of the Merkle scheme functions, including the digest (Δ), *Proof-of-Inclusion* (PoI) and *Proof-of-Consistency* (PoC) functions, as well as the corresponding verification functions ($VerPoI$ and $VerPoC$), all defined in *Merkle digest scheme* (Section 3.9).

Specifically, to commit to the current contents of the log, the logger computes the *Signed Tree Head* (STH), i.e., a signature over the digest of the log concatenated with the current time:

$$STH \equiv \text{Sign}_{L.s}(m.\Delta(Log(date)) + date) \quad (8.2)$$

Note that when we discuss CT, we adopt the term *tree head* to refer to the *digest* of the log, in order to be consistent with terminology used in the CT publications.

A AnG-CT logger should produce a new STH at least once every period of length MMD (Maximal Merge Delay). A small MMD would result in overhead, and a long MMD will reduce transparency; typical MMD would be an hour or few hours.

To prove that a certificate is included as item number i in the current log, the logger computes the PoI as follows:

$$PoI \equiv m.PoI(Log(date), i) \quad (8.3)$$

To validate the PoI , use the $VerPoI$ function.

Finally, we use the PoC (*Proof-of-Consistency*) and $VerPoC$ functions, for validation that a later (signed) tree head, corresponds to the same log as that of an earlier (signed) tree head, except for possible addition of additional certificates.

The Merkle tree functions facilitate the *audit* and *gossip* protocols of AnG-CT; we describe these protocols briefly below. Note that in AnG-CT, relying parties invoke audit as part of the validation process of certificates.

Auditing in AnG-CT. The *audit* mechanism allows detection of some of the possible misbehaviors of loggers, related to the *omitted-certificate attack* (Figure 8.22). For this goal, the audit mechanism requires loggers to respond to several requests; the requests could arrive from any entity, in particular, *monitors*, *relying parties* and *auditors*. These requests include:

Request for PoI+STH: request contains a certificate (or its identifier); the logger should respond with an STH , and a *Proof-of-Inclusion* (PoI) of the certificate in the STH . This is typically issued by a relying party (or an auditor), after receiving a certificate and STH , e.g., as part of an SSL/TLS handshake.

Request for PoC: request contains the identifiers of two (issued) STH s; the logger should respond with a *Proof-of-Consistency* (PoC), showing that

the later STH is a continuation of the earlier STH, i.e., is the result of adding additional certificates to the previous set of certificates. This request may be issued by a monitor, to check for consistency between different STHs received from the logger, or, similarly, by relying party (or other auditor).

Request for STH: requests the logger to send back the current (latest) *Signed Tree Head*. This is usually used for monitoring the logger.

Request for range of Certificates: requests the logger to send all certificates logged, between given start and end time.

The first misbehavior is when a logger sends an *SCT* for a certificate, but omits that certificate from a later-sent *STH*; the second misbehavior is when the logger sends two STHs which are *inconsistent*, i.e., the set of certificates whose digest is in the later-issued STH, is not a superset of the set of certificates in the earlier-issued STH.

Split-world attack on AnG-CT. The audit mechanism allows detection of a rogue logger, when the logger sends two conflicting signed responses - specifically, two conflicting STHs. A rogue logger may attempt to escape such detection, by performing a variant of the *omitted-certificate attack*. This variant of the attack is called a *split world attack*, since the main technique is to send *different STHs to different entities*, making it appear to each party as if the log is consistent among its different interactions with the logger - but presenting a *different log* to different parties. For example, such rogue logger may present to a victim relying party the appearance that a (rogue) certificate C_M appears in the log, while presenting to a monitor the appearance that the log does not contain C_M .

Figure 8.23 illustrates the split-world attack. A rogue logger issues rogue certificate C_M ; to avoid providing proof-of-misbehavior to the monitor, it ‘omits’ C_M from the certificates it sends to the monitor, much like in the *omitted-certificate attack* (Figure 8.22). However, in AnG-CT, the rogue logger is expected to respond to STH+PoI requests from relying parties, as shown in Figure 8.23! As shown, in the split-world attack, the rogue logger responds with an *STH* and *PoI* which are consistent with the certificate and *SCT* that the relying party received, i.e., which reflect a log which indeed contains the rogue certificate C_M .

However, to avoid detection of the rogue certificate by a monitor, the rogue logger responds to *STH* and certificates requests from monitors, which reflect a different log - a log which *does not contain* the rogue certificate C_M , i.e., where that certificate is ‘omitted’. See Figure 8.23; notice that to reduce clutter in the figure, it presents a ‘merged’ request from the monitor, for the STH *and* for the relevant (new) certificates.

Figure 8.23 also shows how the AnG-CT *gossip* mechanism, which is designed to foil the attack, as we next explain.

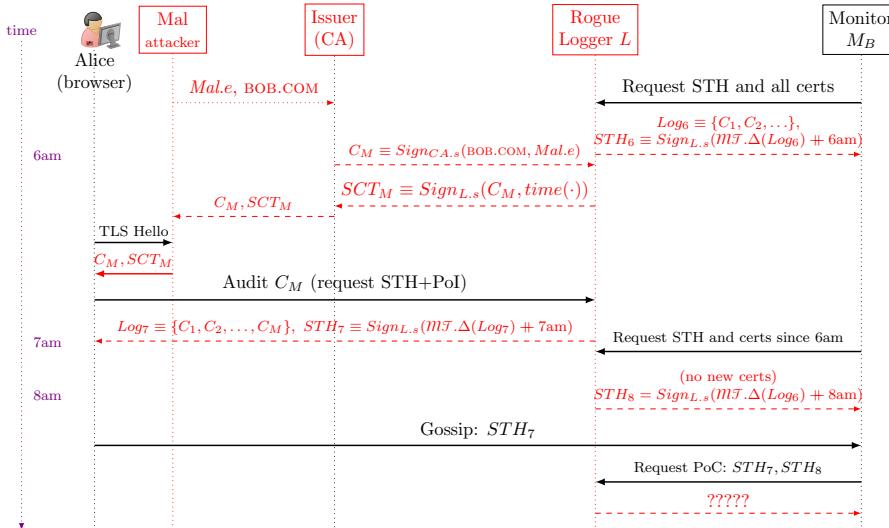


Figure 8.23: Split-world attack on AnG-CT, by attacker, Mal, and rogue logger and CA; the attack is detected by monitor M_B after gossip from relying party (Alice). In this figure, assume a (typical) MMD of an hour, with STH issued every round hour. The scenario begins just before 6am, with Mal requesting the issuer to issue a (rogue) certificate for domain BOB.COM, with the attacker's public key $Mal.e$, and with the monitor M_B requesting from the logger all certificates as well as the Signed Tree Hash (STH). At 6am, the logger responds to the monitor with the current log (set of certificates) Log_6 and with the corresponding STH (STH_6), which is a signature over the digest (tree-head) of Log_6 and over the current time (6am). When Alice connects to the rogue website and receives C_M from the attacker Mal, then she requests from the logger the STH and PoI for C_M . The logger has to correctly provide an STH, denoted STH_7 , which allows a valid PoI for C_M - otherwise, Alice would immediately detect that the logger is faulty. Suppose now, at 8am, Rogue logger tries to hide issuing of rogue certificate C_M , by not reporting about it when the monitor makes a query (at 8am), i.e., sending STH_8 which 'omits' C_M . Suppose Alice later shared STH_7 with the monitor, via gossip; and the monitor asked the logger for Proof-of-Consistency (PoC) from STH_7 to STH_8 . Since the STH_7 digest includes C_M , which is not included in the input to the digest in STH_8 , then the logger cannot produce the required PoC, and the monitor detects that the logger was faulty.

The AnG-CT gossip mechanism: defense against the split-world attack. The Audit-and-Gossip Certificate Transparency (AnG-CT), protects against the split-world attack, by using a *gossip* protocol, as illustrated in Figure 8.23. There are many different gossip protocols, but AnG-CT only requires the very basic gossip functionality: sharing of information among two or more entities. Specifically, AnG-CT uses gossip to share every STH received by a monitor, auditor or relying party, with all other entities.

When an entity receives an STH it did not receive before, it requests the logger to provide the *Proof-of-Consistency* (*PoC*), showing the consistency between this new STH and previously received STHs. Specifically, if we received any STH with timestamp later than the newly received STH, then we ask for PoC from the new STH to the one with next timestamp; otherwise, we ask for STH from the STH received earlier with latest timestamp, to the newly received STH (with even later timestamp).

If the logger is honest, then its STHs should always be consistent: the log whose digest (tree-head) is in the later STH, must contain all the certificates from the log whose digest (tree-head) is in the earlier STH. On the other hand, if this relation does not hold, then the logger cannot produce a valid Proof-of-Consistency. Namely, the entity making the query detects that the logger is corrupt - by receiving invalid response or no response. This is illustrated in Figure 8.23.

What is missing in AnG-CT? The audit and gossip mechanisms ensure that a rogue logger cannot fully avoid detection, once a relying party performs audit. However, AnG-CT is still potentially vulnerable to some attacks:

Non-responding loggers: *no Proof-of-Misbehavior (PoM)*. Two conflicting STHs, e.g., STH_7 and STH_8 , are not sufficient as a *PoM*, when each contains a different timestamp, as in Figure 8.23. To check them, the monitor should send a request for Proof-of-Consistency (PoC) to the logger. Of course, the rogue logger cannot send a valid PoC, since the two STHs are *not* consistent. However, this lack of a PoC response, does not provide a *proof* that can be validated by all entities. Note that the rogue logger may, instead, simply not send the STH when it receives the request for the STH corresponding to the rogue SCT. Since it appears unlikely that all relying parties will perform audit on every certificate received, the rogue logger may even be lucky and not be audited at all on the rogue certificate; but even if it is - it can simply not send any response. What can the relying party do ??

No revocation transparency: *the zombie certificate attack*. Both the HL-CT and the AnG-CT designs do not ensure *revocation transparency*, i.e., ensuring that once a revocation occurred, it will become public information. One important reason for revocation transparency, is to prevent a rogue CA from revoking a certificate at request of the subject (website), due to exposure of the private key; but then intentionally providing an incorrect OCSP response

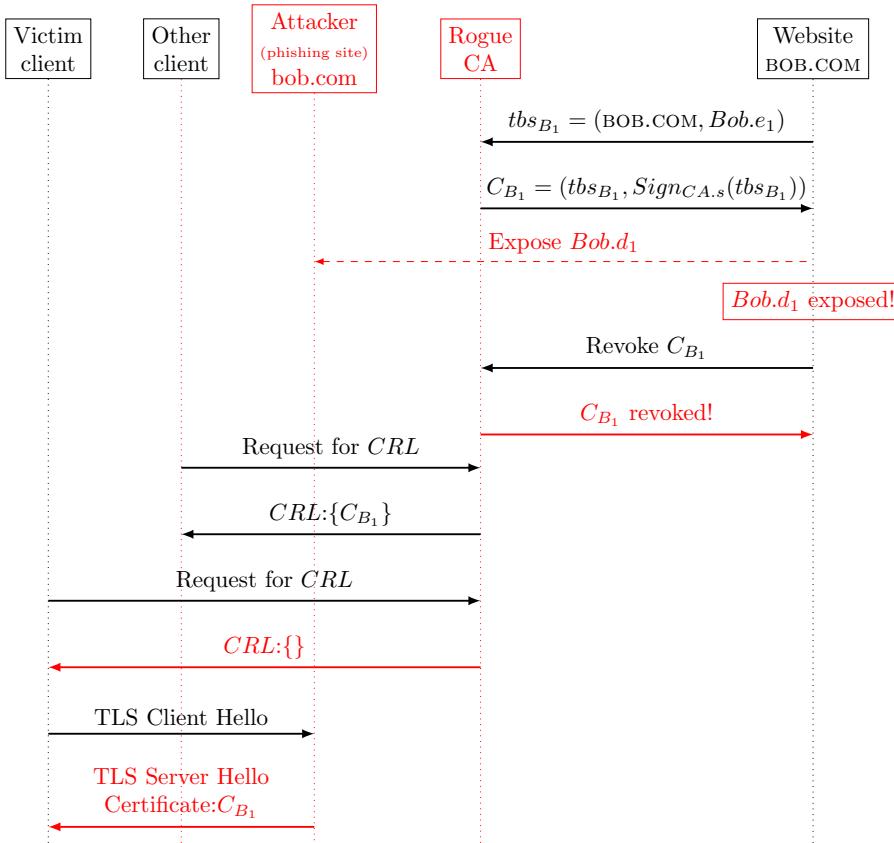


Figure 8.24: Zombie-certificate attack, illustrated for use of *CRL*. Website BOB.COM detects exposure of key *Bob.d₁* and asks the rogue CA to revoke *C_{B₁}*. The CA claims to revoke *C_{B₁}*, and may indeed include it in the *CRL* sent to some non-victim clients, however, does not include *C_{B₁}* in the *CRL* sent to the victim, who then is tricked by the attacker's phishing website who impersonates *bob.com*, presenting *C_{B₁}* and using the exposed key *Bob.d₁*. Note that the attack works, with very minor changes, also with OCSP, and with versions of CT that do not provide *revocation status transparency*.

to a specific (victim) relying party, indicating that the certificate is still valid - to facilitate abuse of the certificate (and exposed private key); we refer to this as the *zombie certificate attack*. See Exercise 8.22 and Figure 8.24.

Exposure of domain name. In AnG-CT, the relying parties should perform *audit* on certificates they receive; this exposes the domain names of the visited websites, i.e., harms the privacy of the relying party (typically, end user).

We next present the NTTP-Security Certificate Transparency (NS-CT) design, which addresses these deficiencies of AnG-CT.

8.6.4 The NTTP-Security Certificate Transparency (NS-CT) design

In this subsection we focus on the NS-CT design, which is a simplification of the further-improved scheme presented in [213]. The term *NTTP* stands for *No Trusted Third Party*; this term was used by the CT designers for the property that security is guaranteed even if any of the participating parties are corrupt.

To ensure NTTP, NS-CT changes the logging process: when the logger receives a certificate for logging, it does not respond immediately with an *SCT*. Instead, the logger waits for the next *MMD* period, and then responds with an *STH* and *PoI*. Namely, whenever a relying party receives an NS-CT certificate, it comes with its *STH* and *PoI*.

This allows NTTP-Security Certificate Transparency (NS-CT) to ensure NTTP, in the following sense. Suppose a relying party receives a certificate that appears valid (following NS-CT validation) at time t . Then one of the following holds:

- Every monitor already received, before t , this certificate (transparency), or
- Before $t + 2MMD$, every monitor will have a Proof-of-Misbehavior (PoM) showing that the logger is corrupted.

The no false PoM property. A trivial way to ensure NTTP is to issue a proof of misbehavior for every logger - even honest loggers; but this is clearly incorrect. We now introduce the *no false PoM* property, which requires that PoM should be valid only against rogue entities. Namely, a scheme ensures the *no false PoM* property if there is never a valid PoM against an honest entity.

Reliable, instantaneous communication simplification. For simplicity, our presentation of NS-CT adopts the following simplification; the complete design, in [213], avoids this simplification. The simplification is that communication is *reliable* and *instantaneous*. With reliable, bounded-delay communication, we can distinguish between the situation where a logger does not respond due to communication failures, and a rogue logger who intentionally does not respond, to avoid publishing certificates or from ‘*incriminating itself*’. Namely, every message sent is received (within small, bounded delay). For our purposes, we will *ignore* this delay, i.e., assume instant communication.

It is fairly easy to modify NS-CT to allow for realistic delays, but this adds some details that we prefer to ignore, so we just ignore the delays.

Assumption: most monitors are honest. Our second simplifying assumption is that most monitors are honest. Namely, the number of monitors is at least $2t + 1$, out of which, at most t may be corrupted; the rest are honest.

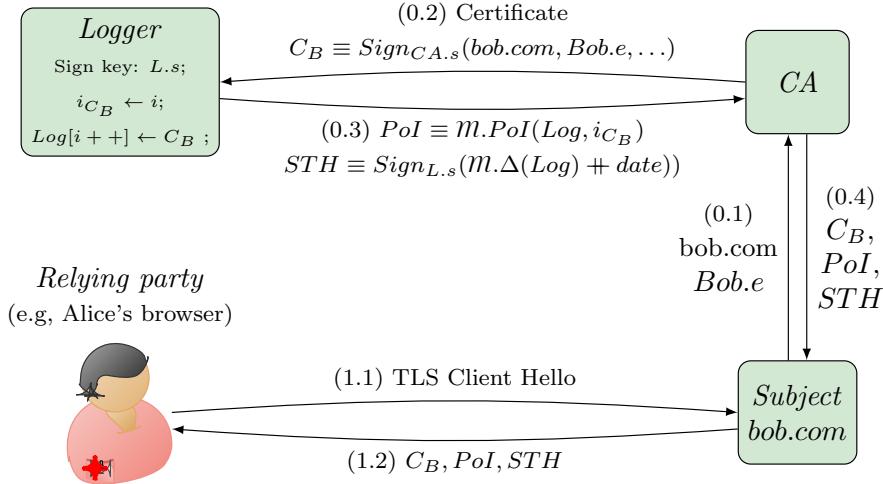


Figure 8.25: NTTP-Security Certificate Transparency (NS-CT) issue process, with a (possibly corrupt) logger, and typical usage in TLS server authentication. The logger responds after it issues the new *STH* (within at most $MMD = 24$ hours). The response contains both the *STH* (*Signed Tree Head*), as well as a *PoI* (*Proof-of-Inclusion*) of the certificate within the log (whose digest is signed in the *STH*). The *PoI* and *STH*, together, allow validation that the certificate C_B appears in the *Log* (based on its contents at end-of-day). The logger uses variable i as the counter of entries in the log; certificate C_B counter value is saved in i_{C_B} . Alice validates signatures (on C_B , *STH*), date, and the *PoI*. The $M.PoI$ and $M.\delta$ functions are of the Merkle Tree scheme, see Section 3.9. This figure does not include additional details, which are (only) relevant for revocation status transparency.

Other NS-CT simplifications. We present NS-CT with the two additional simplifications:

1. All monitors observe all loggers.
2. Loggers issue a new *STH* exactly once every 24 hours. I.e., MMD is 24 hours.

The NS-CT issue process. The NS-CT certificate issuing process is illustrated in Figure 8.25 (steps 0.1 to 0.4), together with a simplification of the TLS handshake between a website with an NS-CT certificate and a browser (relying party), shown as steps 1.1 and 1.2.

This issuing process is quite similar to the one of HL-CT and AnG-CT, shown in Figure 8.20. The difference is that in NS-CT, the logger responds with a new STH, which includes the newly issued certificate, and a PoI, which allows everyone to validate that the certificate is included in the STH. Loggers issue new STHs once every $MMD = 24$ hours; therefore, the logger respond - and the entire issue process - may take up to 24 hours (12 hours on average).

The NS-CT monitor process - detecting rogue certificates. Figure 8.26 shows the basic NS-CT monitoring process, where the logger informs monitors of newly issued certificates, allowing monitors (and their customers) to detect rogue and suspect certificates. This process is quite similar to the monitoring process in HL-CT and AnG-CT, shown in Figure 8.21. The main differences are that in NS-CT, the logger responds with an *STH*, which is a signature over the time-stamped digest of the log including the new certificate, and with the *PoI*, proving the inclusion of the certificate in the log; and this STH is also sent to the monitors. However, the operation of the monitor is essentially the same as in HL-CT: as soon as it identifies the suspect certificate C_M , it informs its customer, in this case - the website BOB.COM.

Audit in NS-CT: preventing the *omitted-certificate attack*. Recall the omitted-certificate attack of Figure 8.22, where a rogue logger colludes with a rogue CA and rogue website, to mislead relying users by the use of a rogue certificate, while not reporting this certificate to the monitors. To foil this attack, NS-CT uses an *audit protocol*, where a relying party reports on the received STH to a monitor. Note that since STHs are issued on a daily basis, every monitor should receive every STH; in NS-CT, monitors *maintain all the received STHs*. Therefore, if the STH received from the relying party is not identical to a previously received STH, the monitor immediately detects that the logger is corrupt.

Furthermore, since all STHs are signed by the logger, this combination of two conflicting STHs serves as a *Proof-of-Misbehavior (PoM)* of the logger. The Monitor shares this PoM with the relying party and with all other monitors and other entities. This process is illustrated in Figure 8.28.

NS-CT audit does not expose visited website. Recall that one drawback of the audit mechanism in AnG-CT was that it required the relying party to send the certificates of the visited website C_M , thereby exposing, typically to the logger, the identity of the visited website - much like the privacy exposure of a relying-party which uses an OCSP query (subsection 8.4.2). However, in NS-CT, the audit request only includes the *STH*, not the specific certificate or *SCT*. Therefore, the privacy exposure is much reduced - the relying party only exposes that it visited some website whose certificate is included in this STH, and normally, there would be many many websites whose certificates are in the same STH.

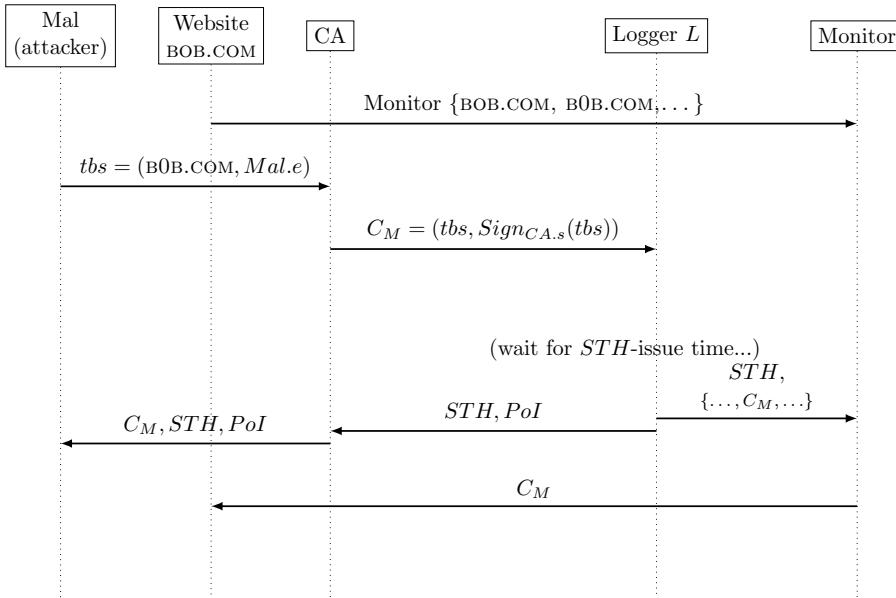


Figure 8.26: NTTP-Security Certificate Transparency (NS-CT) detection by monitor of a ‘misleading certificate’ issued for domain B0B.com. The monitor is asked to supervise BOB.COM, B0B.COM and other ‘potentially-misleading’ domains, by the owner of domain BOB.COM. Attacker (Mal) asks a CA to issue certificate for domain B0B.COM; the certificate, C_M , is issued, but when the logger updates the monitor, the rogue certificate is detected by the monitor and reported to the owner of the domain (BOB.COM), who would probably demand that the CA revoke the certificate.

The split-world attack on NS-CT and its mitigation with gossip. The NS-CT protocol has to protect against a *split-world attack*, quite similar to the one discussed for AnG-CT in Figure 8.23. In the case of NS-CT, the attacker must try to identify the *monitor* M_A used by the victim relying party, say Alice, and the monitor M_B used by the victim subject, e.g., the website BOB.COM.

Recall that in the split world attack, the attacker, whom we denote by Mal, has to control not only a rogue website for the domain BOB.COM (or a rogue related domain name, as in subsection 8.1.1), but also a rogue CA, who will issue the rogue certificate C_M , and a rogue logger, who will issue the corresponding STH and PoI, as required to make C_M a valid NS-CT certificate. The rogue logger ‘omits’ C_M from the set of certificates it sends to M_B and from the corresponding STH, denoted STH_B . However, in order to pass the audit initiated by Alice, the logger must include M_B in the set of certificates it sends to M_A and in the corresponding STH, denoted STH_A .

This makes it quite easy for the two monitors, M_A and M_B , to detect the conflict between STH_A and STH_B ; the combination of the two conflicting STHs becomes, therefore, a Proof-of-Misbehavior (PoM) which can be easily

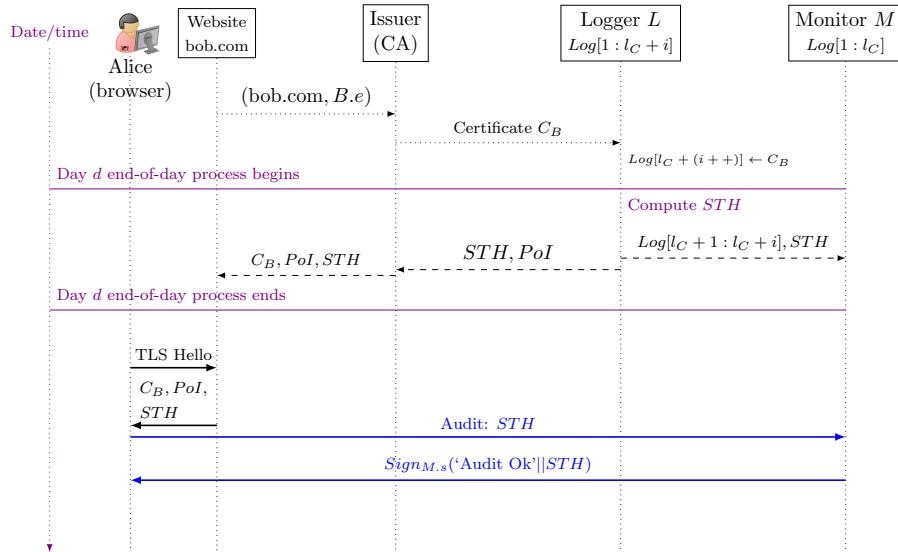


Figure 8.27: NTTP-Security Certificate Transparency (NS-CT), no-faults scenario.

verified by anybody, requiring only the public signature-validation key of the logger.

This scenario is illustrated in Figure 8.29.

Proof-of-Misbehavior for non-responding logger. A logger may try to avoid providing ‘incriminating evidence’, by simply not sending any STH to some (or all) monitors. The gossip mechanism foils this attack quite easily. Monitors perform gossip immediately upon receiving an STH. If no STH is received from some logger after the required time, directly or via gossip (from another monitor), then the monitor detects this logger must be corrupt. Upon such detection of a corrupt logger, the monitor uses gossip to broadcast a signed *detection message*, alerting other monitors to the fact that the logger is corrupt. Actually, all monitors should be aware of this fact, since if one of them received STH, then using gossip, all should have received the STH. The importance of sharing these signed detections is that a set of $t + 1$ such signed detections, becomes a *Proof-of-Misbehavior (PoM)* of this logger, since we assumed at most t rogue entities.

8.7 Additional Exercises

Exercise 8.10. Some X.509 extensions are quite elaborate, and may contain many elements of different types; for example, the SubjectAltName (SAN) extension. A relying party may receive a certificate with a known extension, but containing an element of unknown type, or incorrectly formatted. X.509 does

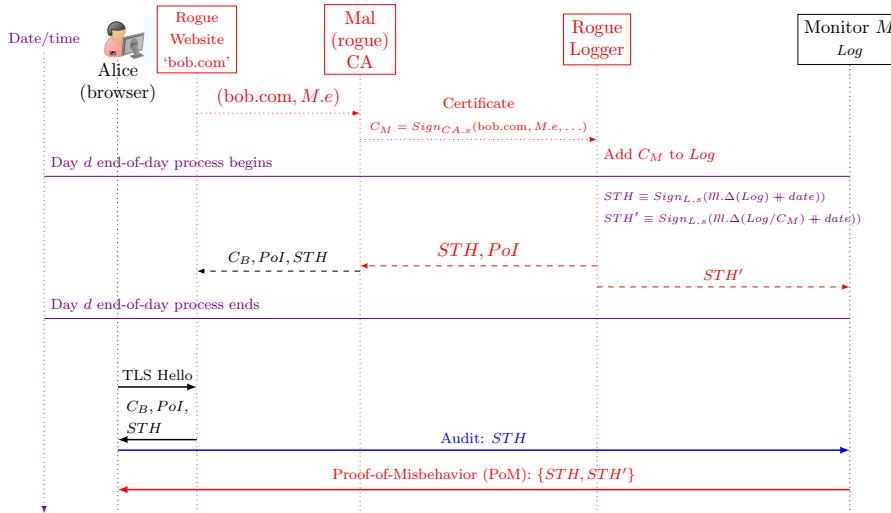


Figure 8.28: NTTP-Security Certificate Transparency (NS-CT) defending against the *omitted-certificate attack*, by providing a *Proof-of-Misbehavior (PoM)* of a rogue logger. The rogue logger sends to the rogue CA (and via it, to the rogue website) a rogue certificate C_M , which contains the domain name of a legitimate domain BOB.COM, or a misleading domain such as B0B.COM; of course, the logger also has to provide a corresponding *Signed Tree Head*, which we denote by STH , and a *Proof-of-Inclusion*, which we denote by PoI . To avoid detection of this rogue certificate C_M by a monitor M , the rogue logger excludes C_M from the computation of the Signed Tree Head STH' which it sends to the monitor M (we don't show sending new certificates to the monitor, but C_M would also be sent, of course). This indeed prevents the monitor from immediately detecting the rogue certificate. However, when a relying party (browser) performs audit of the STH it received with the monitor, the monitor detects the conflict between STH and STH' , both signed and dated (with the same date; the monitor maintains the STH of every day). Together, this pair of two conflicting STHs becomes a *Proof-of-Misbehavior (PoM)* of the rogue logger.

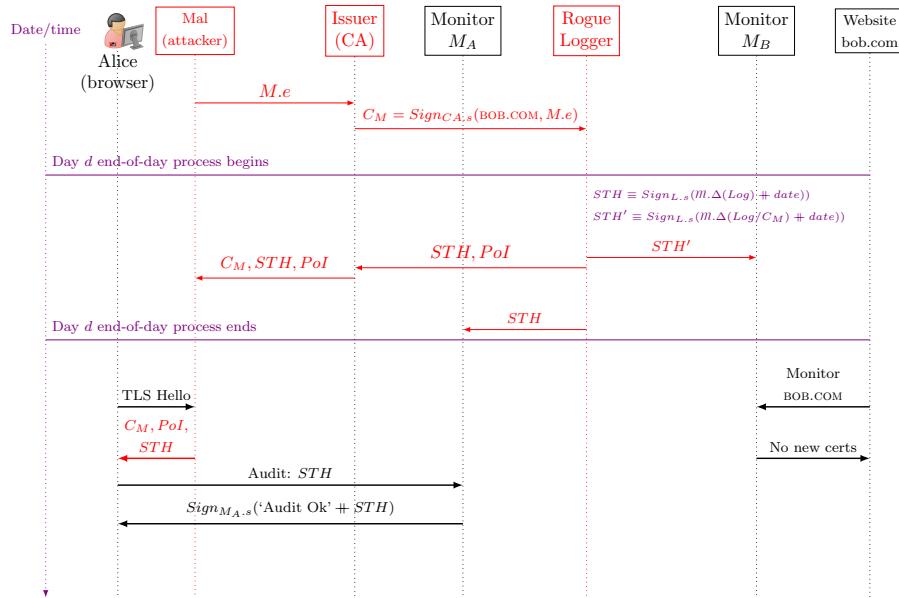


Figure 8.29: NTTP-Security Certificate Transparency (NS-CT): split-world attack by a rogue logger, against NS-CT if (incorrectly) deployed without the gossip mechanism. Logger ‘knows’ that Alice user monitor M_A , while website (bob.com) uses monitor M_B . The attack is foiled by the gossip protocol, which detects the mismatch between the STH received by the two monitors, as we show in Figure 8.30.

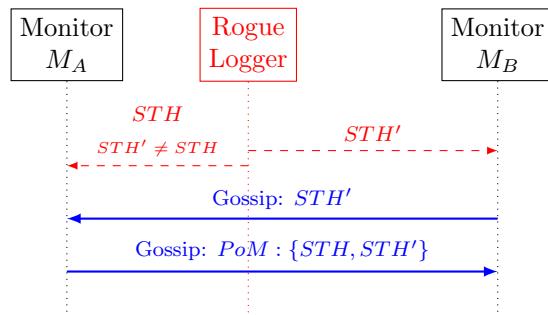


Figure 8.30: Inter-monitor *gossip* detects split-world attack by a rogue logger. Monitors share STHs, and immediately detect that the logger sent two different STHs for the same day. Note that STHs are signed by the logger, so the combination of the two ‘conflicting’ STHs is a *Proof-of-Misbehavior* (PoM), proving that the logger is corrupt.

not specify the correct behavior in such case, and implementations may consider the entire extension as unrecognized; if the extension is marked critical, this results in rejecting the entire certificate, i.e., considering it invalid.

Specify a format for an extension, which will allow defining critical and non-critical element types. Critical elements must be recognized by the relying party or the entire extension is deemed unrecognized, and non-critical elements may be safely ignored by the relying party if not recognized, while still handling the other elements in this extension.

Is this mechanism of potential use within an X.509v3 critical extension? Within a non-critical extension?

Exercise 8.11. Locate the certificate information in your browser, and:

1. Count, or estimate, the number of root CAs.
2. Count, or estimate, the number of intermediate CAs.
3. Can you give two examples of root CAs that you believe belong to organizations or companies that you know and trust? Explain.
4. Can you give two examples of root CAs that you believe belong to organizations or companies that you do not know or do not trust? Explain.
5. Can you give two examples of intermediate CAs that you believe belong to organizations or companies that you know and trust? Explain.
6. Can you give two examples of intermediate CAs that you believe belong to organizations or companies that you do not know or do not trust? Explain.
7. For one of the root CAs, view its details, and identify the extensions. For extensions covered in this chapter, explain their fields. For other extensions, identify what they are used for.
8. For one of the intermediate CAs, view its details, and identify the extensions. For extensions covered in this chapter, explain their fields. For other extensions, identify what they are used for.

Use screen-shots to document your results.

Exercise 8.12. A website eve.com receives a TLS certificate from a CA CA.org. What should prevent eve.com from using this certificate to impersonate as the website for domain bob.com? Present a mistake that CA.org could make in the certificate, which would allow this attack.

Exercise 8.13. It is sometimes desirable for certificates to provide evidence of sensitive information, like gender, age or address. Such information may be crucial to some services, e.g., to limit access to a certain chat room only to individuals of specific gender. It is proposed to include such information in a special extension, and to protect the privacy of the information by hashing it; for example, the extension may contain $h(\text{DoB})$ where DoB stands for date

of birth. When the subject wishes to prove her age, she provides the certificate together with the value DoB , allowing relying party to validate the age.

1. Should such extensions be marked critical? Why?
2. Show that the design above fails to preserve privacy, by describing the algorithm an attacker would use to find out the sensitive information (e.g., DoB), without it being sent by the subject (e.g., Client). Use pseudo-code or flow-chart. Explain why your algorithm is reasonably efficient.
3. Present an improved design which would protect privacy properly. Your design should consist of the contents of the extension (instead of the insecure $h(DoB)$), and any additional information which should be sent inside or outside the certificate to provide evidence of age.
4. Prove, or present clear argument, for security of your construction, under the random oracle model.
5. Present a counterexample showing that it is not sufficient to assume that h is a collision-resistant hash function, for the construction to be secure.

For a real-life example of such leakage, in the VeriSign CZAG extension, see [270].

Exercise 8.14 (Comparison of periodical revocation mechanisms). Assume a set of up to 64 million certificates, and certificates identified by serial numbers from 1 to 64 million, encoded, where necessary, by the minimal number of bits sufficient for these 64-million distinct serial numbers.

1. What will be the size (in bytes) of the CRL, assuming 1% of the certificates were revoked? Ignore the size of elements of the CRL except the list of serial numbers of revoked certificates, and assume (typical although sub-optimal) 96-bit representation for each serial number. Repeat, if 10% of the certificates were revoked.
2. Repeat, for delta-CRL, assuming 1000 certificates revoked during the last day.
3. Repeat both items, assuming efficient representation for serial numbers.
4. Repeat first item, for CRV instead of CRL, without compression.
5. Repeat second item, for ΔCRV instead of Delta-CRLs, without compression.
6. Estimate the size of the compressed CRV and ΔCRV from the last two items, using the simplified compression method described above.
7. Create random CRVs and $\Delta CRVs$, and then compress them using standard compression utility, and compare the length of the compressed versions.

Exercise 8.15 (SCSV certificate extension). *The TLS Fallback Signaling Cipher Suite Value (SCSV) [237], discussed in subsection 7.5.6, is designed to mitigate the Protocol version downgrade-dance attack; see subsection 7.5.6 and Exercise 7.12.*

1. Consider a MitM attacker who impersonates as web-server victim.com, to cause the client to downgrade to an insecure version of TLS; in this version, the attacker is able to find out the master key during the handshake. To foil SCSV, the attacker ignores it, i.e., behaves as if victim.com does not support SCSV. Show a sequence diagram showing how this attacker is able to establish the connection with the client in spite of SCSV.
2. Present an alternative defense to SVSV, which also prevents protocol downgrade attacks using a signaling mechanism, but a different signaling mechanism which uses an X.509 certificate extension.
3. Should your X.509 extension be always marked critical? Always non-critical? Sometimes (when?)? Justify.
4. Suppose a domain adopts your design, and upgrades its certificate accordingly. Is it required to revoke the previous certificate? Justify; if it is required - present an attack if this isn't done (with sequence diagram).
5. This X.509 extension signaling mechanism has the disadvantage that it requires a new certificate; it may take many domains considerable time to upgrade their certificates; even the CA may take time to support this extension. Explain how the new extension may co-exist with the current SCSV signaling mechanism, providing the current level of SCSV defense to domains which did not yet adopt the new X.509 extension. Explain this using sequence diagram for the interaction of a client with such 'legacy-SCSV' website.

Exercise 8.16. As shown in [111], a large number of browser-trusted CAs is operated by organizations such as universities, non-CA companies or other organizations, e.g., religious organizations. For example, assume that one of these is nu.ne, a university, which obtained this signing certificate as an intermediate CA of a trusted root CA, say root.ca. Note that nu.ne would be a (non-existent) Nigerian university, as indicated by its use of the Nigerian country code TLD (ccTLD) ne. The nu.ne university obtained this CA certificate so it may certify its different departments and project websites. Assume that the private key of nu.ne is compromised by a MitM attacker.

1. Would this allow the attacker to intercept the traffic between the user and (1) the university, (2) other Nigerian sites, (3) other universities, (4) additional sites (which)? Assume standard use of X.509 (only).
2. What would be a typical process for detection of the exposure of the private key of nu.ne? Can you bound the time or number of fake-certificates issued until exposure?

3. *Describe and explain any changes in your answers to the previous items, due to adoption of Certificate Transparency by relevant entities.*
4. *What should be the mitigation, once the exposure is detected?*

Exercise 8.17 (Web-of-Trust). *In this exercise we consider Web-of-Trust PKI, first proposed and often associated with the PGP [134] e-mail and file encryption/signing software. In a web of trust, each user can certify the (public key, name) mappings for people she knows, acting as a CA. To establish secure communication, two users exchange certificates they obtained for their keys, and possibly also certificates of the signers. Each user u maintains a directed graph (V_u, E_u) whose nodes V_u are (publickey, name) pairs from the set of all certificates known to u , and where there is an edge from one node, say $(pk_A, Alice)$ to another, say (pk_B, Bob) , if u has a certificate over (pk_B, Bob) using key pk_A ; the graph has a special ‘trust anchor’ entry for u herself, denoted (pk_u, u) . Each user u decides on a maximal certificate path length $L(u)$; we say that u trusts (pk_B, Bob) if there is a path of length at most $L(u)$ from (pk_u, u) to (pk_B, Bob) in (V_u, E_u) .*

Specify an efficient algorithm for determining for u to determine if she should trust (pk_B, Bob) . What is the complexity of this algorithm? Note: you may use well-known algorithm(s), in which case, just specify their names and reference, there is no need to copy them in your answer.

Exercise 8.18 (OCSP-stapling against powerful attacker). *Consider the following powerful attacker model: the attacker has Man-in-the-Middle (MitM) capabilities, and also is able to obtain a fraudulent certificate to a website, e.g., www.bob.com. Suppose, further, that the fraudulent certificate is promptly discovered and revoked.*

1. *Show a sequence diagram showing that this attacker can continue to impersonate as www.bob.com, in spite of the revocation of the certificate and of the use of OCSP-stapling by the website, when the must-staple certificate extension [155] is not used.*
2. *Explain how use of the must-staple certificate extension [155] would prevent this attack.*
3. *Propose a simple extension to OCSP-stapling, OCSP-stapling-pinning, that may also help against this threat, if the must-staple extension is not supported. Explain how your extension would foil the attack you presented. Hint: the required extension is related to the ‘key pinning’ mechanism discussed in subsection 8.5.2.*
4. *Discuss the applicability of the concerns of key pinning, discussed in subsection 8.5.2, to the extension.*
5. *Consider now an even more powerful attacker, who also has MitM capabilities, but also controls a root certificate authority (trusted by the browser).*

Show a sequence diagram showing that this attacker can continue to impersonate as www.bob.com, in spite of the use of the OCSP-stapling-pinning extension as you presented above.

6. Propose an improvement to the OCSP-stapling-pinning extension, that may help also against this threat.

Exercise 8.19 (Keyed hash and certificates). Many cryptographic protocols and standards, e.g., X.509, PKIX and CT, rely on the use of the ‘Hash-then-Sign’ paradigm, based on the assumption that the hash function in use, $h(\cdot)$, is a (keyless) collision-resistant hash function (CRHF).

1. Explain why this assumption is problematic, specifically, prove that there cannot exist any collision-resistant hash function. (If you fail to prove, at least present clear, convincing argument.)
2. One way to ‘fix’ such protocols is by using, instead, a keyed CRHF. Explain why your argument for non-existence of a keyless CRHF does not also imply non-existence of keyed-CRHF.
3. Design how to issue X.509 certificates using a keyed-CRHF, such the certificates can be properly validated by both (1) relying parties that support the keyed-CRHF mechanism and (2) ‘legacy’ relying parties that are not modified and still support only the existing certificate-validation mechanism (using a standard keyless CRHF).
4. If your design in the previous item makes use of any new or existing X.509 extension(s), explain and justify whether the extension(s) should be marked as critical, as non-critical, or if for some applications an extension should be marked critical and for others marked non-critical.

Exercise 8.20 (OCSP Hash-chain Variant). To avoid computational burden on the OCSP responder, it is proposed to use an alternative mechanism to OCSP, which will usually avoid the use of signatures as long as the certificate isn’t revoked, yet ensure secure revocation information. Specifically, add an extension, say called ‘OCSP-chain’, to the X.509 public-key certificate. The OCSP-chain extension will contain an n -bit binary string $x_0 \in \{0, 1\}^n$. A simple, efficient algorithm uses x_0 to validate a ‘daily validation token’ (i, x_i) to be sent by the CA; the CA sends the token (i, x_i) only if the certificate was not revoked i days after it was issued. Your solution may use a cryptographic hash function h , and ‘security under the random oracle model’ suffices, i.e., modeling as if h is a random function.

1. Validation of token (i, x_i) is done by checking that $x_0 = \underline{\hspace{1cm}}$.
2. Assume that the certificate’s maximal validity period is 1000 days. The CA computes/selects x_i , for $0 \leq i < 1000$, by $x_i \leftarrow \underline{\hspace{1cm}}$, and selects x_{1000} by $x_{1000} \leftarrow \underline{\hspace{1cm}}$.

3. Should the OCSP-chain extension be always marked critical? Always non-critical? Sometimes critical and sometimes not? Justify.
4. Assume that the probability of a random certificate to be revoked on a particular day is less than 10^{-5} . A further optimization reduces the number of hash-chain computations by the CA, by ‘grouping’ 100 certificates in a common ‘hash-chain group’. As long as none of these certificates is revoked, they all use the same ‘hash-chain group token’; only when one of them is revoked, a per-certificate token will be sent. This extension requires an extended version of the OCSP-chain extension, i.e., the extension should not simply contain an n -bit binary string $x_0 \in \{0, 1\}^n$ as before. Instead, the extension should contain: _____; and the ‘daily validation token’ will change from (i, x_i) to _____ (before any of the certificates in the group is revoked) and to _____ (after one or more of the certificates in the group is revoked).

Exercise 8.21 (Malicious CA). Let MAC be a malicious CA, trusted (e.g., as a root CA) by browsers.

1. Assume bob.com generates a signing public-private key pair $(B.s, B.v)$, and has $B.v$ certified by MAC. Present sequence diagram showing how MAC can eavesdrop on communication between the site <https://bob.com> and a client, Alice, in spite of their use of TLS 1.3. Your solution should be a very simple attack. Assume the use of only the defenses explicitly mentioned.
2. Assume that Alice uses CT and does not communicate with websites that do not provide a properly signed SCT. Assume a single, trustworthy logger, L , and a single, trustworthy monitor, M . Present a sequence diagram showing how this setup may prevent or deter MAC from performing the attack.
3. Assume that the private key of bob.com is exposed by MAC; however, the exposure is detected, and Bob revokes its certificate. Furthermore, both Alice and bob.com deploy OCSP-stapling, and Bob’s public key certificate includes the must-staple extension. Show, with sequence diagram, how MAC may still eavesdrop on communication between Alice and Bob.

Exercise 8.22 (CT revocation and the ‘Zombie certificate attack’.). The (legitimate) website bob.com receives a CT certificate for its public key $B.e$ from the CA RCA, with SCT signed by logger L . After a year, bob.com detects that its private key was exposed and abused by attackers; it immediately requests RCA to revoke C_B and issue a certificate using the newly-generated public key $B'.e$, and receives the corresponding certificate $C'_B = \text{Sign}_{RCA.s}(\text{bob.com}, B'.e)$. Assume that Alice’s browser is using OCSP, applying soft-fail, except for certificates with the Must-Staple extension (where it uses hard-fail). In the following items, your answers should include a message sequence diagram for each scenario,

indicating TLS and OCSP messages, certificates and SCTs. For the first few items of this question, assume the use of HL-CT.

1. Alice instructs her browser to visit bob.com; however, an attacker, Mal, is able to interfere with the address resolution process, directing the browser to the IP address of Mal, say 6.6.6.6, instead of the IP address of the legitimate bob.com website, say 1.2.3.4. The attacker does not have other traffic-manipulation capabilities, in particular, is not a MitM. Show what happens when Alice's browser tries to establish TLS 1.2 connection with bob.com, assuming that the browser uses OCSP. Denote Mal's public key by $M.e$.
2. Repeat, when the attacker has MitM capabilities (between all parties).
3. Repeat, assuming now (but not before!) that C_B and C'_B include the 'must staple' extension.
4. Repeat, when the attacker controls also RCA.
5. What, if any, would change in your answer when using HL-CT?
6. What, if any, would change in your answer when using NS-CT?

Chapter 9

Human-centered Cryptography

In the previous chapters, we often referred to the communicating entities by common names - most commonly, we used the ‘classical’ names Alice and Bob. In few places, we mentioned the distinction between the human users and their computing and communicating devices - the most important example being in subsection 8.1.1, where we discussed *misleading certificates and domain names*, i.e., certificates for domain names which mislead human users in different ways, to gain unwarranted trust. This is just one aspect of the large and very important area of *human aspects of cybersecurity* - probably the most important and least understood part of cybersecurity. Kevin Mitnick, one of the most famous hackers and the author of several books about hacking, most notably his memoir [234], put it eloquently in his testimony to the U.S. Senate [233]:

The human side of computer security is easily exploited and constantly overlooked. Companies spend millions of dollars on firewalls, encryption, and secure access devices and it’s money wasted because none of these measures address the weakest link in the security chain: the people who use, administer, operate and account for computer systems.

Indeed, it is fascinating how much attention and research is given to the more technical aspects of cybersecurity, in particular cryptography, compared to the human aspects; although clearly, the humans are often the ‘weak link in the chain’.

In this chapter, we explore a bit more of the important and insufficiently studied area of *human-centered security*, i.e., designing systems to ensure security, taking into account the humans using, operating and managing the systems. Our focus is on *human-centered cryptography*, i.e., issues which are related to applied cryptography, the main topic of this volume, and to its use by humans.

Human-Centred Security. The challenge of human-centered security, is to ensure security for systems which are used, administered and operated by humans. Ensuring security for systems used by humans is challenging, sometimes

in surprising ways. Some challenges are obvious: we cannot expect a user to remember an 128-bit key, or to make complex calculations as used by typical cryptographic functions. However, other challenges may be less obvious, and are often overlooked by designers. For example, we discussed in subsection 8.1.1, users are often *misled* by rogue certificates, since they do not notice the use of a visually-similar domain name, or the significance of some variations on the domain name. An even more disturbing reality is that users often fail to notice the lack of security indicators, such as a ‘padlock’ and the use of *https* rather than *http*, and even ignore warnings and proceed, e.g., to web-sites with revoked or invalid certificates, or to install unsigned software or software signed by unknown entities. Indeed, a natural response is: *we can’t protect users who do not follow security instructions, can we?*

However, I disagree, along with many security practitioners; we may not be able to protect *all* users *perfectly*, but we should be able to design our systems to *protect quite well most users*. Or, to quote the well-known security expert Steve Bellovin:

When most people don’t use security tools correctly, the problem is with the tools, not with the people.

From Bellovin’s quote, we derive the following principle for evaluating security mechanisms involving human users, which we call *Bellovin’s principle*.

Principle 18 (Bellovin’s principle: secure means used securely). *A system is considered secure, only if it operates correctly, even under attack, and when used by typical users in typical usage scenarios.*

Usable-security design refers to design of security mechanisms which *offer maximal security* when used by typical users, i.e., its goal is to optimize the security evaluation based on Bellovin’s principle. Usable-security design focuses on the design of the interface and interactions between the human user and computer systems and devices; we use the term *ceremony* to refer to a process involving interaction between users and systems, to distinguish between it and a *protocol*, which we mostly use to refer to interactions between different automated systems.

Usable-security design faces significant challenges, including:

Secure usage is unnatural. Human behavior is the complex result of eons of evolution under dramatically different conditions than usage of information systems. The evolution has resulted in detection of specific threats such as from predators, completely unrelated to the appearance cyber-threats such as malware or phishing website, or even to security alerts. On the contrary, many basic human traits, which served humanity well during evolution, may result in vulnerability to attacks when using computer systems, esp. when security mechanisms are not designed with human behavior in account.

People are strange. [RIP Jim Morison] Human behavior is complex and not fully understood, and there is limited study of human interactions with computerized security interfaces. Even in areas of human psychology which were much studied, there are no precise rules; people react in different and inconsistent ways to the same scenarios.

Secure-usability is hard to measure. Programmers and engineers mostly focus on well-defined specifications and measurable metrics of success. Both of these rarely exist for usable-security design; indeed, it is quite challenging to correctly measure the impact of different designs on the security of a system when used by typical users. Some of the difficulties are due to the fact that human behavior is often modified over time and depends on subtle factors, including awareness of being observed - as when performing experiments to evaluate and compare different usable-security designs. The problem is exacerbated by the fact that psychology is rarely part of the education of programmers and engineers.

Usability beats security. Finally, usable-security design impacts the user-experience - and may often result in conflicts with (non-security) usability and even marketing considerations. In such conflicts, the (non-security) user-experience considerations usually have precedence; see Note 8.3.

Structure of this chapter. In this chapter, we discuss two main topics related to human-centered cryptography, each covered by a section. The first section deals with the basic problem of *authenticating humans*, including both authentication of a user, using passwords and beyond, and authenticating that the current user *is* human - the all-too-familiar CAPTCHA problem. Then, in Section 9.2, we discuss the challenge of *social-engineering and phishing attacks*, and *usable-security defenses*. We conclude with discussion of some of the challenges of research and development of security mechanisms involving human users.

9.1 Login Ceremonies

One of the basic problems in cybersecurity is *authenticating the user* to a computing system or service, e.g., a web-site, i.e., the *authentication ceremony*. Authentication ceremonies are based on some authenticating capability which is available to the authorized user, but not to an imposter. This is similar to the entity-authentication mechanisms of Chapter 5, which are based on a secret key shared between the parties but unknown to the attacker.

Ideally, we should authenticate the user *both* upon *login*, i.e., start of work, *as well as* later, during the work-session - similarly to the combination of handshake and record protocols in chapter 5. However, continuous authentication would annoy users, i.e., degrade the user-experience, and is therefore rarely acceptable - if it requires awareness or involvement from the user. Many systems address

this challenge by automatically logging-out the user after a period of inactivity; even this is often disabled by users.

Some devices, e.g., mobile phones, complement the login mechanism with *continuous user authentication* mechanisms, which typically do not require any user awareness, to avoid annoying the user. Continuous user authentication may be achieved by monitoring the presence of the user, or monitoring for a biometric characteristic of the user, which provides an additional level of authentication. Such continuous authentication biometrics include typing or movement patterns, e.g., *gait* [132]. See further discussion of biometrics below.

However, we focus is on the *login ceremony*, i.e., the interaction between the system and the user upon start of a worksession. The most familiar login ceremony is based on user-id and password; indeed, most systems still use password-based login-ceremonies. However, passwords have well-known, serious drawbacks, both for usability and for security. As a result, there are many other login ceremony solutions, used instead of or together with passwords.

The login ceremonies are usually broken into categories based on the type of capability used by the user to authenticate - and assumed to be unavailable to the attacker; we refer to these capabilities as *Factors of Authentication*. The factors of authentication include:

Something you know: this are methods of authentication, which are based on the user using information which is known to the user, but unlikely to be known to or guessed by an attacker. The most obvious and widely used example are *passwords*. While password schemes are mostly textual, there are different designs for *graphical passwords* [49, 165]. One challenge with passwords is that users often forget them; this motivates the use of passwords in conjunction with a *fallback authentication* method; ‘something you know’ fallback authentication mechanisms include personal *questions-and-answers (Q&As)* [180, 265] and recognition of chosen or familiar images [49, 165]. There are multiple attack vectors on passwords, see subsection 9.1.1 we and Figure 9.1, and multiple proposals and designs to improve the security of password-based login, see subsection 9.1.2.

Something you have: in this ‘factor of authentication’, the authentication is provided by the combination of the user and of a trusted device or object. ‘Something you have’ login ceremonies became very popular, once addiction to mobile phones became common; we discuss some of these mechanisms in subsection 9.1.5.

Something you are - biometrics: *biometrics* are measurable properties which differ among individuals, and hence may identify a particular user. Diverse properties are used as biometrics - fingerprints, facial recognition, voice recognition, and more. See subsection 9.1.6.

Somebody you know: this type of authentication ceremony authenticates a user, typically as a fall-back authenticator when more convenient authenticators fail, using a predefined set of semi-trusted *helpers* of the user. This

is based on the assumption that these ‘friends’ can authenticate the user, using some human authentication mechanism such as direct recognition. This authentication factor was proposed in [67, 290]. See Exercise 9.1.

Somewhere you are: some level of authentication may be obtained from the *location* of the user. This is mostly relevant in controlled-access environments such as offices, and connection to a local-area-network is sometimes used as such indicator. This authentication factor is usually used only as an addition to other, more secure forms of authentication.

9.1.1 Password login ceremonies and attack-vectors

Passwords are a natural ‘something you know’ method of authentication, used from ancient times, and still widely used. A major advantage of passwords is the ease of deployment; there is no dependency or requirement of any device, and users are familiar with the use of passwords. In fact, users are usually even familiar with the basic password security rules, such as:

- Use a long, hard-to-guess password involving different types of characters and avoiding personally-associated data.
- Do not maintain written (or other) records of the password.
- Do not reuse the same password to login into different systems.
- Change the passwords frequently.
- Do not share the password with anyone.

However, it is well known that these rules are mostly broken; e.g., see [234, 331]. Users tend to choose short passwords of mostly or only lower-case letters, write down passwords, reuse them in many systems, change them rarely and share them (with family, friends and colleagues). The main reason is probably convenience, demonstrating the *UX>Security* precedence rule (Note 8.3 and Principle 16).

Considering their wide use, it is not surprising that there are many attacks that expose and abuse passwords - often, by exploiting insecure operation by the user. In Figure 9.1 we illustrate the main attack vectors on password-based login, which include:

Eavesdropping and shoulder-surfing: passwords may be exposed to an eavesdropper, if transmitted in plaintext. Passwords may be sent in plaintext from the client’s device (often browser) to the system (web-server); this is usually avoided by using TLS to protect the password in transit, see Chapter 7. It can be harder to defend against exposure of a password as the user enters it, e.g., by *shoulder surfing*: an attacker directly observing the password as the user enters it - in person or using camera. Such unintended transmission and exposure of information as a result of communication or computation is called a *side channel*; side-channels

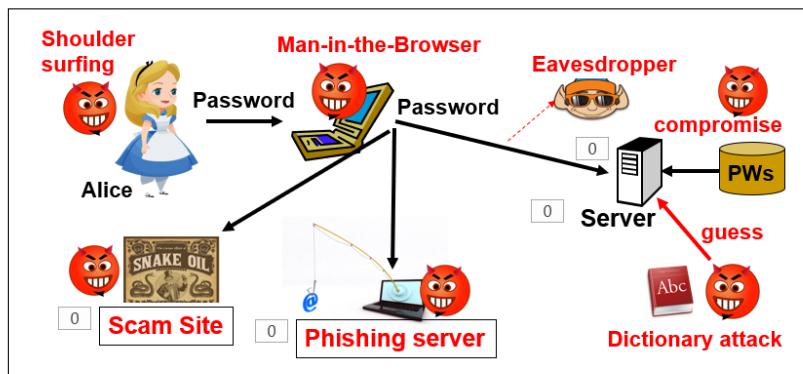


Figure 9.1: Attack vectors on password login ceremony.

are also used for cryptanalysis, e.g., CCA attacks (see subsection 2.8.6). There are other side-channels allowing complete or partial exposure of passwords, including learning partial information about the password from the acoustic sounds made by typing on the keyboard [44], *smudge attacks* which detect residues of use of a touchscreen [15], screen-capture while charging [228] and more. Note that side-channel attacks often expose only partial information, which is then combined with dictionary attacks.

Dictionary attacks: users often fail to choose sufficiently hard-to-guess passwords (see above). Attackers may try to find passwords by creating a list of frequently used passwords; one input to these lists may be the dictionary of words in the language, hence, such list of common passwords is often referred to as a *password dictionary*. (However, password dictionaries would contain many passwords which are not words, and not contain many words in standard dictionaries.) In a ‘classical’ *dictionary attack*, also referred to as *online dictionary attack*, the attacker simply tries to login using each password in the dictionary; this attack is often detected, by counting password-guess attacks, and then foiled, by limiting the number and/or rate of password entry attempts. However, many password system are vulnerable to a variant called *offline dictionary attacks*, where the attacker checks guesses for the password against ciphertexts or other functions of the password, obtained by eavesdropping or system compromise; see subsection 9.1.2.

Social engineering: another common attack is to manipulate users into exposing their passwords to the attacker; the general term for attacks that manipulate users is *social engineering* attacks. Most commonly, users are tricked into entering their password into a *spoofed login dialog*, which they are misled into believing it a trusted login dialog, often as part of a *phishing* attack; see Section 9.2. Note, however, that since users often reuse the same password across multiple sites, it may suffice for an

attacker to receive password for that attacker's own website, and reuse that to login into other systems.

Exposure of user password ‘notes’: passwords are supposed to be only remembered, and not written down; a written password may be exposed, indeed, it becomes a weak ‘something you have’ mechanisms. However, many users do write passwords down [234, 331], often due to concerns about recalling the password when required to login.

Man-in-the-Browser and key-loggers: another common way to steal passwords is directly from the user’s browser or computing device. This is mostly done by ‘hacking’ the browser into collecting the passwords, or by running a *key-logger malware* that collects information typed by the user.

Password file compromise: Finally, passwords are often obtained due to a compromise of the service using them. Usually, attackers simply get hold of an entire passwords database or file, which they can then abuse to impersonate to the compromised service - and to other services where the user used the same password.

Note that most attack vectors exploit the failure of users to follow the password security rules above. However, recall Bellovin’s principle (Principle 18); when most users fail to use the system securely, the problem is with the system and not with the users. Or, as the title of the relevant paper by Adams and Sasse, *users are not the enemy* [4].

Our focus in this book is on cryptography; therefore, in the next subsections, we discuss some of the cryptographic aspects of password security, and in the next section we focus on phishing attacks and spoofed websites - one of the main attack vectors against passwords. However, notice that there are many other important aspects to the security of passwords, and a lot of relevant and interesting research. Some specific interesting aspects, and few example references, include *graphical passwords* [49, 50], password policies and their impacts on users [175], questions-based authentication [289] and many more.

9.1.2 Post-compromise password security: Hashing, Salting and More

Compromise of customer’s data is a nightmare haunting many Chief Security Officers; such events have happened all to often in the recent years. There are many potential damages due to such compromise, such as privacy exposure for the customers; however, let us focus on the risk to password-based authentication.

Quite often, servers maintain

9.1.3 Password-Authenticated Key Exchange (PAKE)

9.1.4 Password Managers

9.1.5 Something you have login ceremonies

Another widely-used authentication factor is based on ‘something you have’. There is a large variety of ‘something you have’ login ceremonies, with different cryptographic designs:

Authenticating device: a device with cryptographic and communication capabilities, provides strong, cryptographic authentication of the user, typically, using a secret key (shared between the device and the system/service). This may be a dedicated device or an application on the user’s phone.

One-time-password device: this device also authenticates the user and then uses a pre-shared key to authenticate the user. However, instead of then authenticating the user against the system/service, this type of device produces a code which the *user* provides to the system/service; this code is typically referred to as a *one-time password*. The code is usually a function of the current time or a challenge received from the system/service.

One-time-passwords: this is a ‘low-tech’ variant of the one-time-password device, where the user simply maintains a list of one-use (one-time) passwords, each used for one login. The list is often simply printed on paper.

Mobile-phone based authentication: this popular user-authentication method authenticates users, using their mobile phones. This relies on the security of the connection between the mobile phone and the cellular network. Usually, the system/service sends an SMS to the mobile, with a code that the user copies and provides (usually, into a webpage). A similar solution is uses a regular phone call.

Authenticating using cookie from browser: this is the common way for users to authenticate to websites: the browser maintains a cookie, which is a string received from the website, and send automatically to the site whenever the browser sends a request to the site. This may be the most widely used method, due to its convenience to the users. We discuss cookies in [161].

9.1.6 Biometrics-based login ceremonies**9.1.7 Two-Factor Authentication****9.2 Phishing attacks and defenses****9.3 Challenges of Usable-Security Research and Development****9.3.1 Secure Usability: Can Johnny Encrypt?****9.3.2 Human Public Key Validation and Security of End-to-End Secure Messaging****9.4 Lab and Additional Exercises**

Lab 5 (Password cracking). *In this lab, we will crack passwords. (to be added soon, basically ready, just need minor adaptations)*

As for the other labs in this textbook, we will provide Python scripts for generating and grading this lab (`LabGen.py` and `LabGrade.py`). If not yet posted online, professors may contact the author to receive the scripts. The lab-generation script generates random challenges for each student (or team), as well as solutions which will be used by the grading script. We recommend to make the scripts available to the students, as example of how to use the cryptographic functions. It is easy and permitted to modify these scripts to use other languages/libraries or to modify and customize them as desired.

Exercise 9.1 (Somebody-you-know login ceremony). *In this question we investigate the somebody-you-know login ceremony [67, 290]. Let Alice be a user who needs to login into a server, using the help from a pre-defined peer Bob. Assume that Bob is using a computing device, and, using this device, shares a key k_B with the server.*

1. *Design a somebody-you-know login ceremony, and present it using a schedule diagram. Your solution may use communication between all parties, and avoid any clock-synchronization assumptions.*
2. *Repeat, but this time avoiding communication between Bob and the server, instead, relying on synchronized clocks.*
3. *Repeat, for the case where the server only has the public key of Bob, not a pre-shared key.*

Hint: compare to the Key Distribution Center (KDC) protocols of Section 5.5.

Exercise 9.2. *Many websites invoke third-party payment services, such as PayPal or ‘verified by Visa’. These services reduce the risk of exposure of client’s credentials such as credit-card number, by having the seller’s site open*

a new ‘pop-up’ window, at the payment provider’s site, say PayPal; and then having the users enter their credentials at PayPal’s site.

1. *Assume that a user is purchasing at the attacker’s site. Explain how that site may be able to trick the user into providing their credentials to the attacker. Assume typical user, and exploit typical human vulnerabilities. Present the most effective attack you can.*
2. *Identify the human vulnerabilities exploited by your construction.*
3. *Propose up to three things that may help to reduce the chance of such attack.*

Exercise 9.3 (Homographic attacks). *To support non-Latin languages, domain names may include non-Latin characters, using unicode encoding. Some browsers display these non-Latin characters as part of the URL, while others display them only in the webpage itself, and, if they appear in the URL, display their punycode encoding (encoding of unicode as few ascii characters).*

1. *Discuss security and/or usability of these two approaches; where there is vulnerability, give an example scenario.*
2. *Some browsers display only using single font, i.e., never displaying domain names which mix Latin characters with unicode characters, or use ponycode encoding in case of mixed fonts. What is the motivation? If this is open to abuse, give example.*
3. *Another proposal is that whenever displaying non-Latin characters, to add a special warning symbol at the end of the domain name, and if the user clicks on it, provide detailed warning. Identify any secure-usability principles and human vulnerabilities which are related to this proposal.*

Solution to first part: There is a usability advantage in allowing non-latin domain names to be displayed using the ‘correct’ font in the URL line: this allows such domain names to appear ‘correct’ to users familiar with the relevant (non-Latin) language. In fact, when such users use a browser that does NOT display non-Latin characters in the URL, then they may be practically unable to understand the URL; this may even open them to phishing attacks as they may get used to ignore the URL and not notice when the domain name is incorrect.

However, browsers displaying URLs with non-Latin characters may facilitate homographic attacks on websites in domain names consisting of Latin characters - the vast majority of websites, definitely popular ones. Specifically, an attacker may be able to buy a domain name which includes or consists of non-Latin characters and is available for sale, although it is visually very similar to a domain name used by some ‘victim’ website, due to using non-Latin characters which are visually similar (some are visually almost identical!) to some latin characters in the victim domain name. For example, the latin character P has a visually-similar Cyrillic character (also looking as P); hence the domain name

APPLE.COM may be written using the Cyrillic P but appear visually identical to the ‘real’ APPLE.COM domain name.

Exercise 9.4 (Anti-phishing browser). *A bank wants to design a special browser for its employees, which will reduce the risk of them falling to phishing attacks. It considers the following changes from regular browsers. For each, specify if it would have significant, small, negligible or no positive impact on security, and justify, based on secure usability principles.*

1. Only allow surfing to SSL/TLS protected (<https://>) sites.
2. Do not open websites as results of clicking on URLs received in emails.
3. If user clicks on URL in email, browser displays warning and asks user to confirm that the URL is correct before requesting that page.
4. On the first time, per day, that the user surfs to a protected site, popup a window with the certificate details and ask the user to confirm the details, before requesting and displaying the site.

Exercise 9.5. *In the Windows operating system, whenever the user installs new software, a pop-up screen displays details and asks the user to approve installation - or to abort. Many programs are signed by a vendor, with a certificate for that vendor from a trusted CA; in this case the pop-up screen displays the (certified) name of the vendor and the (signed) name of the program. Other programs are not signed, or the vendor is not certified; in these cases the pop-up screen displays the names given by the program for itself and for the vendor, but with clear statement that this was not validated.*

1. Identify secure usability principles violated by this design, and explain how attacker can exploit human vulnerabilities to get malicious programs installed.
2. An organization wishes to prevent installation of malware, so it publishes to its employees a list of permitted software vendors, so that employees would verify their names before installing. Present criticism.
3. Propose an alternative method that an operating system could offer, that would allow organization a more secure way to ensure that only programs from permitted vendors would be run.

Exercise 9.6. *Some browsers display only using single font, i.e., never displaying domain names which mix Latin characters with Unicode characters, or use Punycode encoding in case of mixed fonts. Punycode is encoding of each Unicode character, as one or more ASCII characters.*

1. Which abuse this defense is designed to prevent? Identify, with example.
2. Other browsers always display non-Latin characters as Punycode. Identify advantages and/or disadvantages, related to security and usability.

Exercise 9.7. *Operating systems and servers often avoid storing the user's passwords, and yet are able to verify the passwords when provided by the user in login process.*

1. *Explain a common, and simple yet (relatively) secure way in which servers can validate the password without having it stored.*
2. *Motivate this behavior; what is the underlying design principle? Give example of a realistic risk, when such method is not used (and servers keep the passwords).*

Chapter 10

Conclusions and a taste of few advanced topics

10.1 Secret sharing and its Applications

10.2 Side-channels

10.3 Elliptic Curves Cryptography

10.4 Quantum and post-quantum cryptography: by Walter Krawec

The field of computer and information security is a highly fascinating one with numerous sub-domains. One area of interest intersects with the rapidly advancing field of quantum computing leading to *quantum cryptography* and *post-quantum cryptography*, two very different areas of research with a common goal: to maintain security even with adversaries capable of executing quantum algorithms (e.g., after the development of quantum computers).

Quantum computers utilize *quantum bits (qubits)* to process information. A qubit is the basic unit of quantum information; it is a superposition of two states, i.e., can be in both states simultaneously. Indeed, it is impossible to measure directly the qubit; a measurement returns a random bit, where the probability of the returned bit depends on the value of the qubit; at the same time, the value of the qubit changes to the returned bit value.

Due to several unique properties of qubits as compared to classical bits, there are many problems which quantum computers can solve efficiently (i.e., in polynomial time) which are thought to be hard on a classical computer. In particular, Shor presented an efficient quantum algorithm to factor large numbers or to take a discrete logarithm, thus effectively breaking many public key cryptographic systems [294]; Shor's algorithm can also be used to break ECC based systems. While quantum computers with a sufficient number of qubits needed to actually break real-world systems (such as RSA) are not

currently available (one would need a quantum computer with many thousands or even millions of qubit processing power), considering how rapidly they have advanced lately, it is important to design and study systems that can survive such a creation. Additionally, such systems provide fascinating new theoretical problems to study!

Quantum cryptography and post-quantum cryptography both aim to build systems secure against quantum-enabled adversaries, i.e., adversaries who are able to execute quantum algorithms in addition to classical algorithms. Quantum cryptography [72, 261] does this by actually harnessing quantum information (e.g., qubits) to send information securely (even against quantum-enabled adversaries). Post-quantum cryptography [45, 246] achieves this by switching to new mathematical assumptions which are assumed to be hard even for quantum computers to solve. Both fields have their own advantages and disadvantages:

Quantum cryptography requires new equipment and, while their security does not depend on unproven computational assumptions, current day devices do have several security vulnerabilities which need to be carefully guarded.

Post-quantum cryptography uses classical information processing, and thus does not require investing in a new communication infrastructure or hardware (it may operate over today's Internet and current computing technologies). However, its security depends on unproven computational assumptions.

Both fields hold many fascinating open research problems, though. Also, both can be combined into hybrid systems.

As mentioned, quantum computers utilize qubits. Qubits differ in many ways from classical bits. For instance, qubits can take the state “0” or “1”, like classical bits; when these are used as a state of a qubit, we denote them as $|0\rangle$ and $|1\rangle$, respectively. However, qubits can also take any arbitrary *superposition* state, combining 0 and 1; such an arbitrary superposition is denoted by $\alpha|0\rangle + \beta|1\rangle$ where the α and β are *complex numbers* called *probability amplitudes*. We require that $|\alpha|^2 + |\beta|^2 = 1$ for reasons which will be clear shortly.

If one has a collection of n -qubits, then these n qubits may be placed in a superposition consisting of an exponential number of terms using only a polynomial number of operations, called *quantum gates*. In particular, using only n quantum gates, the states of these n qubits may take on a superposition of all possible n -bit strings (i.e., it will consist of a superposition of 2^n different values) denoted $\frac{1}{\sqrt{2^n}} \sum_{x \in \{0,1\}^n} |x\rangle$.

Another difference is that, while reading a classical bit leads to a deterministic outcome, reading a qubit (or *measuring* it) leads to a potentially random outcome and is also potentially destructive. The probability of observing a particular outcome depends on the probability amplitude. For example, if you were to measure the qubit state $\alpha|0\rangle + \beta|1\rangle$, you would “see” (or *observe*) $|0\rangle$ with probability $|\alpha|^2$; otherwise, you would observe outcome $|1\rangle$ with probability

$|\beta|^2$. Furthermore, the state *collapses* to the observed value. So if the original qubit was in the state $\alpha|0\rangle + \beta|1\rangle$, and if someone measures it and observes $|0\rangle$, the state collapses to $|0\rangle$, that is it becomes $|0\rangle$ and the original state is destroyed. This is important for cryptography, as an adversary who attempts to read data from a quantum state may cause a collapse, if the measurement is not done correctly.

Finally, qubits cannot be copied deterministically (the so-called *no-cloning* theorem of quantum mechanics). From a security stand-point, this requires all adversaries to be active and they cannot copy quantum data to attack offline at a later point in time.

These, of course, are not the only differences but they serve to highlight some of the differences in quantum information processing as opposed to classical information. Both quantum algorithms (used, for instance, to attack classical cryptographic systems like RSA) and quantum cryptography (used to protect against future developments in quantum technology) carefully utilize all of the above properties. For a general introduction to quantum computing, the reader is referred to [248, 327].

10.4.1 Quantum cryptanalysis and post-quantum cryptography

While quantum cryptography utilizes quantum information to counter the security threat posed by quantum computers, post-quantum cryptography uses classical information processing to achieve the same end by switching to new assumptions that are considered hard for quantum computers to solve. While there are several quantum algorithms, perhaps the most relevant to cryptographic applications are Shor's Algorithm [294] and Grover's Search Algorithm [150].

Quantum algorithms work differently from classical ones in that they are able to take advantage of several properties unique to qubit systems. At a very basic level, a quantum algorithm works as follows: first the user prepares the input state to the algorithm. For example, if the input is a bit string $x \in \{0, 1\}^n$, then n qubits are prepared in the state $|x\rangle$. Additionally, some auxiliary space may be initialized to $|0\rangle$. From this, a sequence of quantum gates are applied - these gates are basic operations acting on one or two qubits (sometimes more) and act as the quantum analogue of classical logic gates. Some gates simply “flip” a $|0\rangle$ to a $|1\rangle$; some alter the *phase* of a qubit (e.g., changing a $|1\rangle$ to a $-|1\rangle$, that is changing the phase of the probability amplitude); others cause the qubit to evolve into a superposition (e.g., changing a $|0\rangle$ to a $\frac{1}{\sqrt{2}}|0\rangle + \frac{1}{\sqrt{2}}|1\rangle$). Other actions are of course possible too. Gates may also act conditionally, allowing the algorithm to change the state of certain qubits only if others are in a particular state. When the sequence of gates needed by the algorithm are completed, the n qubits are finally measured leading to the “answer” to the problem. Additional post-processing may be required at this point to derive the actual answer to the problem being solved. Furthermore, quantum algorithms

are probabilistic and may fail (with some bounded probability) and so the users may have to run the algorithm again several times to get the correct answer.

Shor’s algorithm provides an exponential speedup over the best known classical solution to the factoring problem. Interestingly, Shor’s algorithm will not directly determine the factors of a number $N = pq$. Instead, the quantum algorithm will output the period of the function $f(x) = a^x \bmod N$ for a random number a with $\gcd(a, N) = 1$. That is, the quantum portion of Shor’s algorithm actually finds the order of a in the multiplicative group \mathbb{Z}_N^* . A classical algorithm is then run, using number theoretic results, to actually find the non-trivial factors of N given the order of $f(x)$. Shor’s algorithm can also be adapted to compute discrete logs. It can also be used to break ECC based public key systems.

Besides Shor’s algorithm, Grover’s algorithm is another quantum procedure that may be used to attack cryptographic systems, though it is perhaps not as “devastating” in this application as Shor’s. Grover’s search algorithm allows a quantum computer to search over an unsorted search space of size N for a particular element x^* satisfying some search criteria in time $O(\sqrt{N})$, a quadratic speedup over a classical solution. Hence, unlike Shor’s algorithm, this does not lead to a “break” of a system; instead it does require one to double the key-lengths of any cryptographic system in order to attain the same level of security as against an adversary using a non-quantum computer. For example, Grover’s algorithm can be used to find the secret key used to encrypt a known plaintext. Here, the search space is the set of all keys $\{0, 1\}^n$ for some encryption algorithm. Given a ciphertext c of a known message m , a quantum computer could determine the key used k in time $O(2^{n/2})$. Thus the need to double key sizes to be secure against Grover’s search.

Post-quantum cryptography involves the design and analysis of cryptographic protocols that are secure against quantum algorithms such as these. For symmetric key encryption, AES is considered a post-quantum secure scheme. For public key systems, there are several candidate problems that are being used to develop post-quantum secure schemes. One example is lattice-based cryptography on which several schemes are based such as NTRU. The quest, however, is still ongoing for new post-quantum cryptographic standards. NIST currently has an ongoing competition to standardize post-quantum cryptographic systems with the third round of the competition having started in July 2020 [239].

Despite the obvious threat posed to classical cryptographic algorithms by quantum computers, and their rapid advancement lately, they still have not developed to a point powerful enough to break real-world systems such as RSA. For instance, the number of qubits needed to successfully factor an RSA composite $n = pq$, is several times the number of bits of n ; the number may even be measured in the millions, depending on how “noisy” or imperfect those qubits are. However, once such a system is available, it is estimated, in [135], that the time required to break an RSA key would only be 8 hours. This estimate is using plausible assumptions on quantum hardware.

10.4.2 Quantum Cryptography

The underlying theme of Quantum Cryptography is to utilize quantum states (e.g., qubits) to perform security tasks, in a way which would be secure (even) against quantum computers. Quantum cryptography was, for a while, synonymous with *quantum key distribution* (QKD) though, today, it encompasses many more fascinating cryptographic primitives. Still, QKD is the most celebrated result, and so we focus on it here.

With QKD, the goal is to permit two users, Alice and Bob, to establish a shared secret key. Here, security will be against a *computationally unbounded adversary* - in particular, the only assumption one needs to make on the power of the adversary is that they obey the same laws of physics that Alice and Bob do. Note that secure key distribution against computationally unbounded adversaries is an impossible task using classical communication alone.

By now there are several QKD protocols each with various advantages and disadvantages. The very first such protocol is the BB84 QKD protocol named after the two creators Bennett and Brassard in 1984 [43]. Other protocols include the B92 [42], SARG04 [3], LM05 [222], and Extended-B92 [221] just to list a few. While there are many protocols, all of them take advantage of the fact that qubits, unlike classical bits, can be prepared and sent in a multitude of manners. Furthermore, due to the no-cloning theorem of quantum mechanics, any attempt to attack quantum information must be active as an adversary is not capable of copying the transmitted quantum information (this impossibility result is a rule of physics and not a computational or technological assumption - quantum physics rules out the possibility of creating perfect copies of quantum data). By encoding a classical bit string (the key that Alice wants to send to Bob), in a variety of randomly chosen manners, any attempt at eavesdropping will necessarily disturb the information which may be detected by Bob. Not only this, but for a correctly designed QKD protocol, there is a direct correlation between the amount of disturbance in the quantum channel (which can be measured by the users) and the maximal amount of information an adversary may hold on the transmitted information. This is in stark contrast to classical information!

In quantum computing and communication, besides measuring a qubit, one may also apply quantum gates to the system. One vital gate is the *Hadamard* gate which takes a qubit $|i\rangle$ (for $i = 0$ or 1) and changes it to a $\frac{1}{\sqrt{2}}|0\rangle + (-1)^i \frac{1}{\sqrt{2}}|1\rangle$. The Hadamard gate also reverses itself, thus by applying it twice, one returns to the original state $|i\rangle$. From this, BB84 works essentially as follows. Alice will choose a random key-bit k and prepare the qubit $|k\rangle$. She then randomly chooses whether to apply the Hadamard gate to this qubit - or not. She keeps secret both choices (her key bit and her choice to use the Hadamard gate), and sends the resulting qubit to Bob. Now, when Bob receives the qubit, he alerts Alice to this fact using the classical authenticated channel. At this point Alice reveals to Bob her choice of whether to use the Hadamard gate or not. If she did, Bob applies the gate himself, otherwise he does nothing. At this point, the qubit he has should be of the form $|k\rangle$ and so Bob will measure

and learn the key bit.

But what happens if there is a MitM adversary? Let's assume the adversary captures the qubit when it travels from Alice to Bob. She cannot make a copy due to the no cloning theorem. Also, if she tries to extract information from the qubit in the form of a measurement, she may destroy the state as she does not know if Alice used the Hadamard gate or not. If Eve measures right away and Alice did use the Hadamard, she will get a random outcome and the original state collapses. Similarly, if Eve applies the Hadamard (guessing that Alice did so), but Alice did not use the Hadamard, again, the measurement afterwards would be random. In other words, any attack by Eve has a chance of destroying the qubit and, furthermore, potentially gives Eve random information. Due to state collapse, this interference can be detected by Bob.

Security proofs for QKD protocols generally involve the following process: first determine what observations users can make on the quantum channel (for instance what is the probability of an error in the quantum signal). Second, use this information to upper-bound an adversary's information on the raw key transmission. From this, classical error correction and a cryptographic process of *privacy amplification* are performed to derive a final secret key. One important statistic of any QKD protocol is its *key rate*, namely the ratio of secret key bits to number of qubits sent. A second, and related, statistic is the *noise tolerance* of the protocol - that is the noise threshold for which the key-rate becomes zero as an adversary has potentially too much information. For BB84, the key-rate of the protocol, at least in the ideal case, takes a relatively simple form of $r = 1 - 2h(Q)$ where Q is the observed error rate (namely, Q is the probability of an error in the quantum signal) [295] and $h(x)$ is the binary Shannon entropy function, namely $h(x) = -x \log_2 x - (1-x) \log_2(1-x)$. One may readily check that when $Q = 0$ (there is no noise), the key-rate is $r = 1$ meaning the users get one secret key bit for every qubit sent. When $Q = 11\%$, the rate r drops to zero (thus BB84's noise tolerance is 11%). Between these two extremes, the rate is strictly positive but decreasing as Q increases.

In practice, QKD, and quantum communication in general, can be achieved in a variety of ways. Typically photons are used to encode and transmit quantum information but even here there are several degrees of freedom within the photon that may be used by Alice and Bob to encode their key information. Typically polarization, phase, and timing are considered. These photons may be transmitted over fiber channels or through free-space (including the use of satellites for QKD) [261].

While QKD security in the ideal setting is information theoretic and does not require any computational assumptions on the adversary, one weakness of current QKD progress is in the numerous opportunities for side-channel attacks on the actual devices due to the imprecision of current-day devices. This can include attacks involving Alice's source (which may not create one qubit each time but perhaps multiple copies by accident); or Bob's measurement device (which has several attack opportunities). However, there are technological countermeasures to these issues; there are also alternative protocols which can be proven secure using an even stronger security model such as *measurement*

device independence [51, 295], where the adversary is assumed to fully control the measurement devices thus mitigating any side-channel attack against them.

As mentioned, however, QKD is far from the only application of quantum cryptography. Numerous exciting possibilities exist when using qubits to secure information. For instance, certifiable deletion is a possibility [71]. Here, one may give a ciphertext to someone (e.g., a cloud server) and, at any future point, ask that user to delete the cipher text. A certificate of deletion can be provided proving that the ciperhertext was destroyed. Since the ciphertext is a quantum state, it cannot be cloned and so this certificate proves that all copies of the ciphertext have been deleted. Obviously an impossible task in the classical world as multiple copies of a classical ciphertext may be made. Another interesting possibility is delegated computing [70, 108]; here a user who needs only basic quantum abilities, can perform a complex quantum computation on an input state using a more powerful quantum server yet the server will remain blind to the computation being performed and the input. For even more applications, see the survey [72].

10.5 Privacy and anonymity

give simple PIR example ?

10.6 Theory of cryptography

Appendix A

Background

A.1 Background: Computational Complexity

Most cryptographic schemes assume restrictions on the computational abilities of the adversary¹; this rules out attacks which are usually infeasible in practice, such as *exhaustive search* - trying out all possible keys.

How, however, can we give a well-defined meaning to a restriction on the adversary's computational abilities? This can be quite tricky. However, the theory of *computational complexity* provides an elegant solution. In this introductory textbook, we only need to understand some very basic notions and properties from the theory of complexity, which we try to summarize in this subsection. Our discussion is quite restricted to what we consider essential, and a bit simplified, although it basically conforms with the corresponding precise definitions. Hopefully, this will suffice to allow readers who are not familiar with computational complexity to follow this textbook. It may also motivate some readers to take a course or read one of the many excellent textbooks on computational complexity, e.g., [88, 142], which will provide a more complete and precise coverage. Our discussion also uses a few standard notations, summarized in subsection 1.3.5 and Table 1.1.

Comparing the run-time of algorithms. The main measure we use for computational resources required by a given algorithm, is the maximal run-time of the algorithm, when run over a specific computer (machine). The theory of complexity defines precise, simplified *computation model*, which corresponds to the capabilities of the machine (computer) on which the algorithm will be run. The run-time is the number of operations (steps) required by the algorithm, when using the specified computation model.

The number of operations of an algorithm, obviously depends on the inputs. However, it is useful to bound the run-time of the algorithm as a function of a

¹There are also some definitions and constructions of *unconditionally secure* cryptographic schemes. We cover two important unconditionally-secure schemes: *one time pad* (*OTP*) encryption (Section 2.4) and *Secret Sharing* (Section 10.1).

value which characterizes the ‘difficulty’ of the inputs. One simple measure is the *length*, in bits, of the inputs to the algorithm. In this book, we focus on this measure. Namely, we say that the run-time of an algorithm \mathcal{A} is bounded by function $T_{\mathcal{A}}$, if the number of steps of the algorithm for any input x can never exceed $T_{\mathcal{A}}(|x|)$.

Time and space complexities, and the big-O notation. We usually compare algorithms in terms of their *time and space complexities*, which are simplified bounds on the runtime and storage requirements, respectively.

The simplified bounds use the big-O notation for functions, which is a way to bound the asymptotic behavior of the functions, and ignore any multiplicative factors. Specifically, given two functions f and g over the integers, we say that f is in $O(g)$, if:

$$(\exists c > 0) \lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} < c \quad (\text{A.1})$$

The notation $O(g(n))$ is called the *big-O notation*. We denote this by $f(n) = O(g(n))$.

The big-O notation ignores differences between functions, if the asymptotic impact of the difference is at most a constant. Namely, if $f(n) = O(g(n))$, then it also has time complexity $O(g'(n))$ for every function g' such that $g(n) = O(g'(n))$.

The big-O notation allows simpler bounds, and we try to use the simplest possible big-O notation. For example, if $f(n)$ is a polynomial, $f(n) = c_0 + c_1 \cdot n + \dots + c_l \cdot n^l$, then we would use $f(n) = O(n^l)$ as the natural, and simple, big-O notation. By doing this, it becomes easier to compare the asymptotic behavior of functions.

Let $T_{\mathcal{A}}(n)$ denote an upper bound on the run time of algorithm \mathcal{A} , where n is the number of bits in the input to \mathcal{A} . We say that \mathcal{A} has *time-complexity* $O(f(n))$ if $T_{\mathcal{A}}(n) = O(f(n))$, i.e., if the run-time of \mathcal{A} is in $O(f(n))$. For example, we say that the time complexity of algorithm \mathcal{A} is at most *linear*, if $T_{\mathcal{A}}(n) = O(n)$; and that the time complexity is (bounded by a) *polynomial* if $T_{\mathcal{A}}(n) = O(n^c)$ for some constant c .

In this way, the big-O notation is convenient to compare the run-time of algorithms. Similarly to the time complexity, we sometimes refer to the *space complexity* of an algorithm, which is a similar measure, also using the big-O notation, for the maximal amount of storage required by the algorithm.

Efficient algorithms and functions. Security mechanisms, and in particular cryptographic schemes, should be efficient yet ensure security against adversaries with reasonable capabilities, in particular, computational capabilities. To properly study security, we need a precise definition for these terms (‘efficient’ and ‘reasonable computational capabilities’).

Most research in cryptography uses, for both adversary and schemes, the same definition for efficient algorithms, taken from complexity theory: *probabilistic polynomial time (PPT)* algorithms, also referred to as *polytime* algorithms.

Let us define this important notion, and also *PPT functions*, which are functions computable by PPT algorithms. Since we refer to PPT algorithms also as efficient algorithms, we similarly may also refer to such functions as efficient functions.

Definition A.1 (Efficient (PPT) algorithms and functions). *Let A be an algorithm. We say that A is an efficient algorithm, or write $A \in \text{PPT}$, if there is some polynomial p such that when A is given any input x , then A terminates after performing at most $p(|x|)$ operations, i.e., after running time which is bounded the polynomial p applied to the length of the input x to the algorithm A .*

We say that function f is efficiently-computable, and write $f \in \text{PPT}$, if there is an efficient algorithm A that computes f .

To rephrase the definition using time complexity and the big-O notation, we say that algorithm \mathcal{A} is *efficient* if its time complexity is polynomial, i.e., $O(n^c)$ for some constant c .

Notice that we allow efficient algorithms to be *randomized* (probabilistic). This is in contrast to *deterministic* algorithms, whose output and run-time are determined completely by their inputs.

The security parameter 1^l . Note that the running time is bounded by a polynomial in the *length of the input*. For many algorithms, this is natural, since the computation required depends on the size of their inputs; for example, a B-tree allows search in time $O(\log(n))$, where n is the number of items in the tree (input to the algorithm) [88]. However, in many cryptographic algorithms, the most significant factor impacting the computation time isn't the size of the 'real' inputs, e.g., the message being signed or encrypted.

Instead, the main factor impacting computation time of cryptographic algorithms, and the adversary, is a special input, 1^l , to which we refer to as the *security parameter*. The security parameter 1^l is supposed to be related to the maximal amount of resources that the adversary may spend to attack the system; often, it is directly related to the length of the cryptographic keys used. The security parameter is usually specified as a *unary* string, i.e., to encode a security parameter whose value is an integer l , we provide the input 1^l , i.e., an l -bit string ('word'), all of its l bits having the value 1. This is since the running time is bounded by a polynomial in the *length* of the input, not the actual numeric value of the input².

The computational model. Definition A.1 does not specify the *computation model*, i.e., what operations are allowed to the algorithm. In other words, we did not specify the computer on which the program runs: can the algorithm multiply numbers as a single operation, or only add numbers? What is the size

²Notice that it only takes about $\log_2(l) + 1$ bits to encode the numeric value l . For example, the encoding of 4 is 100, whose length is $3 = \log_2(4) + 1$; and similarly the encoding of 2^n is the string $1 + 0^n$, whose length is only $n + 1$ bits.

of the registers (basic storage units), e.g., can the algorithm add two 32 bit numbers as a single operation, or maybe two 64 bit numbers? A more precise definition would specify the exact computational model, typically, as a *Turing machine*. A Turing machine models a very simple computing device, albeit with unbounded storage; see textbooks on the theory of complexity, e.g., [88, 142].

However, we will not define Turing machines or any other computational model, since we believe that readers who are not interested in theory, will be able to understand our notions of efficiency even without precise understanding of any specific computational model. This belief is based on the (important) fact that *the class of efficient (polytime) algorithms is robust* to (reasonable) changes in the computational model. Namely, an efficient (polytime) algorithm for some problem under the Turing machine model, exists if, and only if, there exists an efficient algorithm for the same problem when running over different ‘real’ computers. This includes computers with arbitrary (yet fixed) register size, and with a wide variety of operations (essentially, every operation that can be computed efficiently). For details, see, e.g., [88, 142].

The class of efficient algorithms has other nice properties; let us give two examples. We again refer the reader to [88, 142] and other sources for details.

Fact A.1. *If A and B are efficient algorithms, then:*

1. *Consider algorithm C , which runs in a special machine, whose operations include running of A and B on inputs defined by C . Then if C is an efficient algorithm (when we consider running of A and B as a single operation), and A , B are efficient (when running over over a machine which only supports basic operations), then the same computations as done by C , may be done by an efficient algorithm C' which runs over a machine with only basic operations.*
2. *Their cascade, algorithm C , defined by $C(x) = A(B(x))$, is also efficient.*

The $NP \stackrel{?}{=} P$ question. Let us now introduce the most famous open question in the theory of computer science: $NP \stackrel{?}{=} P$. The question refers to two classes of algorithms - P and NP ; we also use P (or NP) to refer to the set of functions computable by an algorithm in P (or NP , respectively). Let us first introduce these two important *complexity classes*:

- The class P is similar to PPT ; it contains all *deterministic* polynomial time algorithms. Since every deterministic algorithm is also a randomized algorithm (simply never flipping coins), then $P \subseteq PPT$.
- The class NP , contains functions whose solutions can be *verified* in polynomial time. Namely, if $f \in NP$, then there is a predicate³ $v \in P$ such that $v(x, y) = \text{TRUE}$ if and only if $y = f(x)$. Note that every function in P is also in NP , i.e., $P \subseteq NP$.

³A predicate is simply a function whose output is always one of the two values `TRUE` and `FALSE`.

The name NP stands for *non-deterministic polynomial-time*; see [88, 142] for discussion of the concept of *non-deterministic computation*. As explained there, every algorithm \mathcal{A} in NP can be emulated by a deterministic algorithm \mathcal{B} . However, this generic emulation is inefficient: the run-time of \mathcal{B} may be *exponential* in the inputs. Of course, there are many problems for which we can find a non-deterministic polynomial-time algorithm \mathcal{A} , but we can also find an efficient (deterministic or randomized polynomial-time algorithm) \mathcal{B} . Is this always possible? That is *the* question, at least in the theory of computer science. This question is often referred to as $NP \stackrel{?}{=} P$.

The $NP \stackrel{?}{=} P$ question asks, if the two sets, P and NP , are identical. Since $P \subseteq NP$, the only question is whether also the reverse holds, i.e., $NP \subseteq P$; if this holds then $P = NP$. This problem has been thoroughly studied, and a solution, either showing that $P = NP$ or showing that $P \neq NP$ (hence $P \not\subseteq NP$), would have dramatic implications on several areas. In particular, many cryptographic schemes, including RSA, would be deemed insecure if it is found that $P = NP$, or ‘only’ that $PPT = NP$.

Let us give a specific example of a problem which is in NP , but not known to be in P . This is the *factoring* problem, which is the basis for the conjectured security of the RSA cryptosystem.

The factoring problem. The factoring problem is one of the oldest problems in algorithmic number theory, and is the basis for RSA and other cryptographic schemes. Basically, the factoring problem involves finding the prime divisors (factors) of a large integer. However, most numbers have small divisors - half of the numbers divide by two, third divide by three and so on.... This allows efficient *number sieve* algorithms to factor most numbers. Therefore, the factoring hard problem refers specifically to factoring of numbers which have only *large* prime factors.

For the RSA cryptosystem, in particular, we consider factoring of a number n computed as the product of two large random primes: $n = pq$. The factoring hard problem assumption is that given such n , there is no efficient algorithm to factor it back into p and q .

It is not *known* whether factoring is in P , i.e., is efficiently computable; if it is, then there is an efficient algorithm that finds the RSA private key and allows decryption of messages. However, factoring definitely is in NP ; hence, if $P = NP$, factoring is also in P , which would imply that RSA is insecure.

To see that factoring is in NP , we observe that there is an efficient algorithm to *verify* the factors of a large composite number such as $n = pq$. Namely, given p and q , there is an efficient algorithm that finds if $n = pq$. In fact, in this case, verification consists simply of multiplying p by q and comparing the result to n . In fact, even if given only one of the two, say p , verification is still easy: divide n by p and confirm that the result is an integer q with no residue.

A.2 Background: Number Theory and Group Theory

Number theory is often used in the design and analysis of cryptographic schemes. In this section, we introduce the minimal subset of number theory that is necessary for our study of applied cryptography. The subset of number theory that we need is mostly focused on *modular arithmetic*.

A.2.1 The modulo operation and modular arithmetic

Modular arithmetic is based on the *modulo* operation, denoted \bmod , which takes two inputs, an integer, say a , and a positive integer m which is called the *modulus*. The modulo operation is denoted $a \bmod m$ and returns the *residue*⁴ of division of the integer a by the modulus m . For example, $13 \bmod 4 = 1$, since $13 = 1 + 3 \cdot 4$. Note that for any given $a, m \in \mathbb{Z}$ such that $m > 0$, there is exactly one such residue of a modulo m , as the reader should be able to confirm. The following definition tries to make this more clear.

Definition A.2 (Residue, congruence, the modulo operation, \mathbb{Z}_m). *Let $m > 0$ be an integer ($m \in \mathbb{Z}$). We say that integers a and b are congruent modulo m if $a - b$ is divisible by m , i.e., there is some integer i such that $a - b = i \cdot m$. We denote the fact that a is congruent modulo m with b by the notation:*

$$a \equiv b \pmod{m}$$

We use \mathbb{Z}_m to denote the set of integers from zero to $m - 1$, i.e., $\{0, \dots, m - 1\}$. Every integer a is congruent to an integer $r \in \mathbb{Z}_m$, i.e., $0 \leq r < m$; we say that $r \geq 0$ is the residue of a modulo m . For any given $a, m \in \mathbb{Z}$, there is exactly one such residue of a modulo m ; we denote it by $a \bmod m$, and refer to \bmod as the modulo operation, and to m as the modulus.

Modular arithmetic is the computation of expressions involving arithmetic operations over integers, where the operations include modulo operations. The \bmod operation is applied after all ‘regular’ arithmetic operations such as addition and multiplication; i.e., $(a + b) \bmod m = [(a + b) \bmod m]$.

The reader should be able to confirm the following useful, basic properties of the modular operation, which hold for every integers $a, b, m \in \mathbb{Z}$ where $m > 0$:

$$(a + b) \bmod m = [(a \bmod m) + (b \bmod m)] \bmod m \quad (\text{A.2})$$

$$(a - b) \bmod m = [(a \bmod m) - (b \bmod m)] \bmod m \quad (\text{A.3})$$

$$a \cdot b \bmod m = [(a \bmod m) \cdot (b \bmod m)] \bmod m \quad (\text{A.4})$$

$$a^b \bmod m = (a \bmod m)^b \bmod m \quad (\text{A.5})$$

Example A.1. Let $A = a_n a_{n-1} \dots a_2 a_1 a_0$ be an $n + 1$ -digits number. Let us use Equations (A.2) and (A.4) to prove that A is divisible by 3 if and only if

⁴Some people use the term *remainder* in stead of *residue*.

the sum of its digits $\sum_{i=0}^n a_i$ is divisible by 3:

$$\begin{aligned}
 \sum_{i=0}^i (10^i a_i) \bmod 3 &= \sum_{i=0}^i (10^i a_i \bmod 3) \bmod 3 \\
 &= \sum_{i=0}^i [(10^i \bmod 3)(a_i \bmod 3)] \bmod 3 \\
 &= \sum_{i=0}^i [(1)(a_i \bmod 3)] \bmod 3 \\
 &= \sum_{i=0}^i (a_i \bmod 3) \bmod 3 \\
 &= \sum_{i=0}^i a_i \bmod 3
 \end{aligned}$$

The reader is encouraged to prove the similar rule for division by 9.

Exercise A.1. Use Eq. (A.5) to show that for any integers a, b holds $a^b \bmod (a - 1) = 1$.

Similar properties hold for any polynomial $p(x)$ with integer coefficients and input ($x \in \mathbb{Z}$), as well as for a polynomial $p(x_1, x_2, \dots)$ with integer coefficients and multiple integer parameters ($x_1, x_2, \dots \in \mathbb{Z}$):

$$[p(x)] \bmod m = [p(x \bmod m)] \bmod m \quad (\text{A.6})$$

$$[p(x_1, x_2, \dots)] \bmod m = p(x_1 \bmod m, x_2, \dots) \bmod m = \quad (\text{A.7})$$

$$= p(x_1 \bmod m, \dots) \bmod m \quad (\text{A.8})$$

These properties often make it much easier to compute the residue of a complex expression or calculation, as the following exercise shows.

Exercise A.2. Compute the following values:

1. $9 \bmod 7$
2. $45 \bmod 7$
3. $\frac{45}{9} \bmod 7$
4. $445 \cdot (81 \cdot 34^{13} + 83 \cdot 33^{345}) \bmod 4$

Solutions:

1. $9 \bmod 7 = 2$ since $9 = 2 + 1 \cdot 7$.
2. $45 \bmod 7 = 3$ since $45 = 3 + 6 \cdot 7$.
3. $\frac{45}{9} \bmod 7 = 5 \bmod 7 = 5$.

4. Denote $445 \cdot (81 \cdot 34^{13} + 83 \cdot 33^{345}) \pmod{4}$ by x . Then we find x as follows:

$$\begin{aligned} x &= 445 \cdot (81 \cdot 34^{13} + 83 \cdot 33^{345}) \pmod{4} \\ &= (445 \pmod{4}) \cdot ((81 \pmod{4}) \cdot (34 \pmod{4})^{13} + \\ &\quad + (83 \pmod{4}) \cdot (33 \pmod{4})^{345}) \pmod{4} \\ &= 1 \cdot (1 \cdot 2^{13} + 3 \cdot 1^{345}) \pmod{4} \\ &= (2 \cdot 4^6 + 3) \pmod{4} \\ &= 3 \pmod{4} = 3 \end{aligned}$$

The reader should be able to show that *congruence modulo m* is an *equivalence relation*, namely, that it satisfies the following properties:

Reflexivity: $a \equiv a \pmod{m}$.

Symmetry: $a \equiv b \pmod{m}$ if $b \equiv a \pmod{m}$.

Transitivity: if $a \equiv b \pmod{m}$ and $b \equiv c \pmod{m}$ then $a \equiv c \pmod{m}$.

A.2.2 Multiplicative inverses

Note that when we solved the third item of exercise A.2, we did not compute the modulo of the numerator and denominator and then divide - that would have resulted in $\frac{3}{2} \pmod{7}$, and $\frac{3}{2}$ is not even an integer, so it would have no modulus. Indeed, division was *not* included in equations A.2 to A.8; in fact, division of two integers, $\frac{a}{b}$, is not always an integer - hence, $(\frac{a}{b}) \pmod{m}$ may not even be defined. We picked an example - $\frac{45}{9} \pmod{7}$ - where $\frac{45}{9} = 5$, i.e., is an integer, so $\frac{45}{9} \pmod{7}$ is defined (and equal to 5).

Of course, in ‘regular’ algebra, we often use division to solve equations; so what can we do in modular arithmetic? The answer is that often - but not always - instead of dividing, we can *multiply by the multiplicative inverse*. This actually also usually works in ‘regular’ algebra over the reals: every non-zero real number $x \in \mathbb{R}/0$, i.e., $x \neq 0$, has a multiplicative inverse which we denote by x^{-1} or $\frac{1}{x}$, and for any y holds $\frac{y}{x} = y \cdot x^{-1}$. The following fact similarly defines a multiplicative inverse modulo m , and shows a sufficient and necessary condition for its existence.

Fact A.2. *Let $a \in \mathbb{Z}$ be an integer. We say that integer b is the multiplicative inverse modulo m of a , if $a \cdot b \equiv 1 \pmod{m}$; if it exists, we denote the multiplicative inverse by $b = a^{-1} \pmod{m}$ (or, when m is clear from context, simply a^{-1}).*

An integer a has multiplicative inverse modulo integer $m > 0$, if and only if a and m are coprime, namely, they do not have a common divisor (except 1).

Proof: see number-theory textbooks such as [171]. □

When we know the multiplicative inverse $d^{-1} \pmod n$ of the divisor d of a modular division $y = \frac{x}{d} \pmod n$, we can use it to transform the computation into multiplication:

$$y = \frac{x}{d} \pmod n = x \cdot d^{-1} \pmod n \quad (\text{A.9})$$

This can be easier to compute, since we can now reduce d (as well as x) modulo n . For example, see the following exercise.

Exercise A.3. *Compute:*

1. $12^{-1} \pmod 5$.
2. Find x such that $x \cdot 144 \pmod 5 = 2$.

Solution:

1. $12^{-1} \pmod 5 = (12 \pmod 5)^{-1} \pmod 5 = 2^{-1} \pmod 5$, so let's find the inverse of 2 modulo 5, which is probably a bit easier. Namely, we need to find integers a, b so that $a \cdot 2 + b \cdot 5 = 1$. One simple solution is $a = 3, b = -1$, since: $3 \cdot 2 + (-1) \cdot 5 = 1$. Hence, $12^{-1} \pmod 5 = 2^{-1} \pmod 5 = 3$. To check that we were not wrong, let us perform the modular multiplication, i.e.: $(12 \cdot 12^{-1}) \pmod 5 = (12 \cdot 3) \pmod 5 = 36 \pmod 5 = 1$, as required.
2. To find x , we need to ‘divide’ by 144 - that's what we would have done if the equation was over the real numbers (not modular). But since the computation is modulo 5, we must instead multiply by $144^{-1} \pmod 5$. As explained in proof of Fact A.2, it is sufficient to find $(144 \pmod 5)^{-1} \pmod 5 = 4^{-1} \pmod 5$. Actually, in the previous item we already found that $2^{-1} \pmod 5 = 3$, so we can use this to know that $4^{-1} \pmod 5 = 2^{-1} \cdot 2^{-1} \pmod 5 = 9 \pmod 5 = 4$. Hence, we need to solve $x \cdot 144 \cdot 144^{-1} \pmod 5 = 2 \cdot 144^{-1} \pmod 5$, i.e., $x = 2 \cdot 4 \pmod 5$, i.e., $x = 3$. Let us check the solution: $3 \cdot 144 \pmod 5 = 12 \pmod 5 = 2$, as required.

The computation of multiplicative inverses allows us to turn modular division into multiplication (Equation A.9, and then reduce the multiplicands. Can we similarly reduce *exponentiation*? Equation A.5 shows that we may reduce the *base* in exponentiation: $a^b \pmod m = (a \pmod m)^b \pmod m$. But what about reducing the exponent b ? That seems to be often even more important than reducing the base! In the next two subsections, we present Fermat and Euler's theorems, which allow reductions in the exponent.

Fermat and Euler's theorems also provide an efficient method to compute multiplicative inverses modulo a prime (Fermat) or a composite whose factorization is known (Euler). However, recall that the factoring problem is considered hard (Section A.1); when the factors are hard to compute, computing multiplicative inverses is also hard. In fact, this is crucial to the security of the well-known and important RSA public key cryptosystem, which we discuss in Chapter 6.

Specifically, in RSA, the private key is the multiplicative inverse of the public key, which means that if it is possible to efficiently compute multiplicative inverses, then RSA is insecure.

A.2.3 Fermat's and Euler's Theorems

Fermat's theorem, also known as Fermat's little theorem⁵, is a simple but very useful result; in particular, it allows to efficiently compute multiplicative inverses modulo a prime p , and to simplify computation of exponentiation modulo a prime, by reducing the exponent modulo $p - 1$. We later present Euler's theorem, which generalizes Fermat's little theorem to the case of a composite modulus, allowing efficient computation of multiplicative inverses and reducing the exponent, when the factoring of the modulus is known.

Theorem A.1 (Fermat's theorem). *For any positive integer a and prime p holds:*

$$a^{p-1} \equiv 1 \pmod{p} \quad (\text{A.10})$$

Proof: See, e.g., [171]. □

The following exercise makes a very useful observation which follows (easily) from Fermat's theorem.

Exercise A.4. *Show that for any positive integer q holds:*

$$\begin{aligned} a^q \pmod{p} &= a^{q \pmod{(p-1)}} \pmod{p} \\ &= (a \pmod{p})^q \pmod{p} \end{aligned} \quad (\text{A.11})$$

Note that since p is prime, it is definitely also coprime with any integer a such that $a \not\equiv 0 \pmod{p}$, i.e., a and p have no common divisor. In particular, this holds for every $a \in \mathbb{Z}_p^*$, where \mathbb{Z}_p^* is a fancy notation for the set $\{1, \dots, (p-1)\}$ (i.e., the same as \mathbb{Z}_p , but excluding zero).

Fact A.3. *For any prime p and any positive integer a such that $a \not\equiv 0 \pmod{p}$, in particular, for $a \in \mathbb{Z}_p^*$, holds $a^{p-1} \equiv 1 \pmod{p}$. Also, $a^{p-2} \pmod{p}$ is the multiplicative inverse modulo p of a .*

Proof: From Fact A.2, a has a multiplicative inverse modulo p , denoted a^{-1} (more precisely, $a^{-1} \pmod{p}$). Hence, the fact follows from Fermat's theorem. □

The following exercise uses Fermat's theorem and Fact A.3.

Exercise A.5. *Compute the following, without a calculator. Use Fermat's little theorem (Theorem A.1), if it is applicable; and use, if/where necessary, the modulo operation rules.*

⁵The name *Fermat's little theorem* helps to distinguish this theorem with few other important theorems associated with Fermat, in particular, the famous Fermat's last theorem. For brevity, we drop the 'little' and use the term Fermat's theorem.

n	1	2	3	4	5	6	7	8	9	10
$\phi(n)$	1	1	2	2	4	2	6	4	6	4
factors?	none	none	none	$2 \cdot 2$	none	$2 \cdot 3$	none	2^3	$3 \cdot 3$	$2 \cdot 5$

Table A.1: Euler's function $\phi(n)$, computed for small integers; see Equation A.12.

1. $13^{31} \pmod{31}$
2. $17^{734} \pmod{4}$
3. $19^{26} \pmod{17}$
4. $3^{-1} \pmod{11}$. Check your answer by confirming that $3 \cdot 3^{-1} \equiv 1 \pmod{7}$.

Euler's function and theorem Fermat's theorem allows us to reduce the exponent, when computing modulo a prime p . We can often reduce the exponent even when computing modulo a composite number n , using *Euler's Theorem* (Theorem A.3). Euler's theorem is a fundamental result from number theory, which is a generalization of Fermat's little theorem.

Euler function ϕ . Before we present Euler's theorem, we must introduce *Euler function* $\phi(n)$, also referred to as *Euler's totient*⁶ *function*. We define $\phi(1) = 1$, and for every integer $n > 1$, we define $\phi(n)$ as the number of positive integers which are less than n and *co-prime* to n . Two integers i, n are *co-prime* if they do not have any common divisor (except 1, of course, which divides all integers). Namely:

$$\phi(n) \equiv |\{i \in \mathbb{N} | i < n \wedge \gcd(i, n) = 1\}| \quad (\text{A.12})$$

Where $\gcd(i, n)$ is the *greatest common divisor* of i and n , i.e., the largest integer j s.t. $0 = i \pmod{j}$ and $0 = n \pmod{j}$.

Euler's function $\phi(n)$, computed for small integers, is shown in Table A.1.

The following lemma shows that if p is a prime then $\phi(p) = p - 1$. Furthermore, for a multiplication of two different primes p, q holds: $\phi(p \cdot q) = (p - 1) \cdot (q - 1)$. Notice how this holds for the respective values in Table A.1, e.g., $\phi(7) = 6$, $\phi(10) = 4$.

Lemma A.1. For any prime $p > 1$ holds: $\phi(p) = p - 1$. For prime $q > 1$, if $q \neq p$, then: $\phi(p \cdot q) = (p - 1)(q - 1)$.

Proof: A prime p is clearly co-prime to any positive integer smaller than it, hence $\phi(p) = p - 1$.

For primes p, q , the number of positive integers smaller than $p \cdot q$ is again $p \cdot q - 1$; but some of them are divided by either p or by q . Let $A_p \equiv \{p, 2p, \dots, (q-1)p\}$ denote the set of positive integers smaller than $p \cdot q$ divided by p , and

⁶The word ‘totient’ comes from ‘how many’ in Latin.

$A_q \equiv \{q, 2q, \dots, (p-1)q\}$ denote the set of positive integers smaller than $p \cdot q$ divided by q . Then:

$$\phi(p \cdot q) = (p \cdot q - 1) - |A_p \cup A_q| \quad (\text{A.13})$$

Let us compute the size of $|A_p \cup A_q|$. First note that no positive integer smaller than $p \cdot q$ can be divided by *both* p and q , since they are (different) primes. Hence $|A_p \cup A_q| = |A_p| + |A_q|$. Directly from the definition of each of these sets we see that $|A_p| = q - 1$ and $|A_q| = p - 1$.

By substituting in Equation A.13, we have:

$$\phi(p \cdot q) = (p \cdot q - 1) - (p - 1) - (q - 1) = p \cdot q - p - q + 1 = (p - 1)(q - 1)$$

□

The next lemma generalizes Lemma A.1, and states that the Euler function is *multiplicative* for co-prime inputs, as follows.

Lemma A.2 (Euler function multiplicative property). *If a and b are co-prime positive integers, then $\phi(a \cdot b) = \phi(a) \cdot \phi(b)$.*

Proof: Omitted.

Note that if a is co-prime to both b and c , than a is co-prime to bc . Hence, Lemma A.2 generalizes to a multiplication of multiple co-primes, e.g., if each pair among a, b and c is co-prime, then $\phi(a \cdot b \cdot c) = \phi(a) \cdot \phi(b) \cdot \phi(c)$.

We are getting close to being able to compute the Euler function for any integer, given its factors. However, the Lemmas so far do not allow us to compute the Euler function of an integer which is a *power of a prime*, i.e., p^i where p is a prime and i is a positive integer. For example, $9 = 3^2$ and $\phi(3) = 2$, but $\phi(9) = 6$. Let us now compute the Euler function for p^i .

Lemma A.3. *For any prime p and integer $l > 0$ holds $\phi(p^l) = p^l - p^{l-1}$.*

Proof: Left to the reader. *Hint:* use the same idea as in proof of Lemma A.1. □

Given a number n and a set of pairs $\{(p_i, l_i)\}$ such that $n = \prod_{i=1}^n (p_i^{l_i})$, where $\{p_i\}$ is a set of distinct primes (all different), and l_i is a set of positive integers (exponents of the different primes), we say that $\{(p_i, l_i)\}$ is a *factorization* of n . We next observe that using the above lemmas, we can compute the Euler function $\phi(n)$, given the factorization of n ; this generalizes the previous lemmas.

Lemma A.4. *Let $n = \prod_{i=1}^n (p_i^{l_i})$, where $\{p_i\}$ is a set of distinct primes (all different), and l_i is a set of positive integers (exponents of the different primes). Then:*

$$\phi(n) = \phi\left(\prod_{i=1}^n (p_i^{l_i})\right) = \prod_{i=1}^n (p_i^{l_i} - p_i^{l_i-1}) \quad (\text{A.14})$$

Proof: Every pair of powers of distinct primes $(p_i^{l_i}, p_j^{l_j})$ are co-primes; hence from Lemma A.2, generalized to a multiplication of multiple co-primes, it follows that $\phi(n) = \prod_{i=1}^n \phi(p_i^{l_i})$. The claim follows from Lemma A.3. □

We next present *the fundamental theorem of arithmetic*, also referred to as the *unique factorization theorem*, which says that *every* integer $n > 1$ has a *unique* factorization; hence, for every $n >$, we can apply Lemma A.4 to compute its Euler function $\phi(n)$. The first presentation and proof of this theorem are due to Euclid; we only present the theorem, not the proof.

Theorem A.2 (The fundamental theorem of arithmetic). *Every number $n > 1$ has a unique representation as a product of powers of distinct primes.*

Note that 1 was considered a prime, the theorem would have been incorrect, or should have been rephrased to exclude 1 factors.

The following exercise may improve your understanding of the Euler function and the different ways to compute it.

Exercise A.6. *Compute the following; use the lemmas and facts above, as necessary.*

1. $\phi(31)$
2. $\phi(93)$
3. $\phi(2^9)$
4. $\phi(12^5)$
5. $\phi(60^3)$

Solution of item 5: first notice that $60 = 3 \cdot 4 \cdot 5 = 3 \cdot 2^2 \cdot 5$.

To compute $\phi(60)$, we apply Lemma A.2:

$$\phi(60) = \phi(3 \cdot 2^2 \cdot 5) \quad (\text{A.15})$$

$$= \phi(3) \cdot \phi(2^2) \cdot \phi(5) \quad (\text{A.16})$$

$$= 2 \cdot 2 \cdot 4 = 16 \quad (\text{A.17})$$

To compute $\phi(60^3)$, we apply Lemma A.4:

$$\phi(60^7) = \phi(3^3 \cdot 2^6 \cdot 5^3) \quad (\text{A.18})$$

$$= (3^3 - 3^2) \cdot (2^6 - 2^5) \cdot \phi(5^3 - 5^2) \quad (\text{A.19})$$

$$= 18 \cdot 32 \cdot 100 = 57600 \quad (\text{A.20})$$

Euler's theorem. Finally, we can now present Euler's theorem:

Theorem A.3 (Euler's theorem). *For any co-prime integers m, n holds $m^{\phi(n)} = 1 \pmod{n}$. Furthermore, for any integer l holds:*

$$m^l \pmod{n} = m^{l \pmod{\phi(n)}} \pmod{n} \quad (\text{A.21})$$

With Euler's theorem, we now have several modular arithmetic tools. First, we have the ‘basic’ modular-reduction rules (Equations A.2 to A.8). Second, we have Fermat's little theorem and Euler's theorem. Finally, to apply Euler's theorem, we can use Lemma A.4 to compute the Euler's function $\phi(m)$ for an integer m , if we have the factorization of m .

The following exercise requires the use of these tools to compute several modular computations. The challenge, of course, is to identify and use the ‘right tool’ for each exercise.

Exercise A.7. *Prove the following, using the appropriate ‘tools’ among these we learned. There may be more than one way to prove.*

1. $13^{31} \bmod 31 = 13$
2. $17^{734} \bmod 4 = 1$
3. $27^{26} \bmod 10 = 9$
4. $35^{841} \bmod 12 = 11$

A.2.4 Group Theory, Cyclic Groups and Generators

In this subsection, we introduce basic notions from the domain of group theory, which are used widely in applied cryptography, and a bit in this textbook.

Let us first define a group. A group is a pair (G, \cdot) of a set G of elements, and an operation, which we denoted \cdot , which is defined on all pairs of elements from G , and whose outcome is always also in G , i.e., G is closed under the operation \cdot . In this book (and in many texts), we usually use the symbol \cdot for the operation of the group, like the typical notation for multiplication, i.e., $a \cdot b \in G$ for every $a, b \in G$; we may sometimes omit the dot and simply write ab , with the same meaning, or use the standard notation for exponentiation, e.g., $a^2 = a \cdot a$. However, it is also possible to use other symbols for the group operation, e.g., $*$ or $+$.

The requirements from group operations are simple, and familiar from both regular arithmetic and modular arithmetic; let us define a group, including these requirements.

Definition A.3. *A group is a set G of elements and an operation, denoted \cdot , satisfying the following requirements:*

Associativity: *for every $a, b, c \in G$ holds $(a \cdot b) \cdot c = a \cdot (b \cdot c)$.*

Identity element: *there exists a (unique) element in G , which we call the identity element and usually denote by $1 \in G$, such that for every element $a \in G$ holds: $a = a \cdot 1 = 1 \cdot a$.*

Inverse: *For each $a \in G$, there is an element $a^{-1} \in G$ such that $a \cdot a^{-1} = a^{-1} \cdot a = 1$, where 1 is the identity element. For each a , there is only one such element, which we call the inverse of a and denote a^{-1} . (From the*

identity element property, it follows that the identity element is always its own inverse.)

A commutative group (also called an Abelian group) is a group that also satisfies:

Commutativity: *for every $a, b \in G$ holds $a \cdot b = b \cdot a$.*

We focus on *finite commutative groups*. One such group is the finite *additive group* denoted Z_n (or \mathbb{Z}_n), which consists of the set $\mathbb{Z}_n = \{0, 1, \dots, (n-1)\}$, for integer $n > 1$, and where the group operation is addition modulo n . For this group, it is more natural to denote the group operation by $+$ rather than by \cdot ; note that we do not include $\mod n$, however, we obviously refer to $\mod n$ addition. Let us see that Z_n is indeed a group, by validating that it satisfies the requirements:

Associativity: for every $a, b, c \in Z_n$ holds $(a + b \mod n) + c \mod n = (a + b) + c \mod n = a + (b + c) \mod n = a + (b + c \mod n) \mod n$. We included the $\mod n$ notation, to make it easier to follow the argument.

Identity element: the identity element is zero; for every $i \in Z_n$ holds $i + 0 = 0 + i = i$.

Inverse: consider any $i \in Z_n$ except zero. Then $(n - i) \in Z_n$ is the inverse of i , since $i + (n - i) \mod n = n \mod n = 0$. Zero is the inverse of itself.

Commutativity: for every $a, b \in Z_n$ holds $a + b = b + a$.

We also use finite *multiplicative groups*, mostly, the $\mod p$ multiplicative group denoted Z_p^* (or \mathbb{Z}_p^*), which consists of the set $\mathbb{Z}_p^* = \{1, \dots, (n-1)\}$, for prime p , and where the group operation is multiplication modulo p . Let us see that Z_p^* is a commutative group:

Associativity: for every $a, b, c \in Z_p$ holds $(a \cdot b \mod p) \cdot c \mod p = (a \cdot b) \cdot c \mod p = a \cdot (b \cdot c) \mod p = a \cdot (b \cdot c \mod p) \mod p$. We included the $\mod p$ notation, to make it easier to follow the argument.

Identity element: the identity element is one; for every $i \in Z_p^*$ holds $i \cdot 1 = 1 \cdot i = i$.

Inverse: consider any $a \in Z_p^*$ except one. From Fact A.3, a has an inverse $(a^{p-2} \mod p)$.

Commutativity: for every $a, b \in Z_n^*$ holds $a \cdot b = b \cdot a$.

We use the *exponentiation notation* to denote repeated application of the group operation with the same operand, i.e., $a^1 = a$ and $a^i = a^{i-1} \cdot a$ for every positive integer i . Exponentiation generalizes naturally to zero or negative exponents, with a^0 defined to be the unit element of the group, denoted simply 1, and a^{-i} defined as the multiplicative inverse of a^i .

We now define the important concepts of *cyclic groups*, *generator* and *order*.

Definition A.4 (Cyclic group, generator and order). *A group G is cyclic, if there is an element $g \in G$ such that for every element $a \in G$, there is an integer i such that $a = g^i$. Such an element g is called a generator of G . The order of G is the integer $q > 0$ such that $g^q = 1$, where g is a generator of G and 1 is the unit element of G .*

Note that $G = \{g^1, \dots, g^q\} = \{1, g, g^2, \dots, g^{q-1}\}$, hence, the order q of a cyclic group G , is also the number of element in G . We also define the *order* of an element $a \in G$; this is the *smallest* possible integer $q > 0$ such that $a^q = 1$. In particular, the order of a is the same as the order of G if, and only if, a is a generator of g .

We can now define the discrete logarithm problem, which is another important number-theoretic problems considered computationally hard, and used as the basis for public-key cryptography, including the Diffie-Hellman key-exchange protocol (Chapter 6). We define the problem for an arbitrary cyclic group G ; a common instantiation is for the multiplicative mod p group \mathbb{Z}_p^* , where p is a prime.

Definition A.5 (Discrete logarithm problem). *Let G be a cyclic group, g be a generator for G . A discrete logarithm of an element $x \in G$ of G (with generator g) is an integer $a \in \mathbb{N}$ s.t. $x = g^a$. The discrete logarithm problem for G is to compute integer $a \in \mathbb{N}$ s.t. $x = g^a$, given a random $x \in G$ and a generator g for G .*

A.3 Background: Probability

Probabilistic analysis and algorithms are very important for computer science in general and for cryptography in particular. However, luckily, only the very basics are required for our study of applied cryptography. For more in-depth coverage, take a course and/or read one of the many excellent textbooks, e.g., [88, 149].

Probability deals with events which result in a value from some predefined set. For simplicity, we only consider a *finite* set of possible outcomes. The classical example is a coin-flip, which has two possible outcomes: Head or Tail. With a fair coin, either outcome is equally likely; and, even for a biased coin, we expect each coin-flip to be independent of other coin-flips.

Let $\{x_1, x_2, \dots, x_n\}$ denote the set of n outcomes of coin-flips, i.e., $(\forall i \leq n) x_i \in \{\text{Head}, \text{Tail}\}$; each of the x_i values is called a *random variable*, i.e., it is a variable whose value is set by a *random experiment*. Since we expect each coin-flip to be independent of the others, it makes sense to compute the *average* number of outcomes with a given value, e.g. Head. We refer to the average fraction of Head outcomes as the *probability* (likelihood) that the random outcome of a coin flip would be Head. Let $\Pr(x = \text{Head})$ denote the probability that an outcome x of a random coin toss is Head, and denote $\Pr(\text{Tail})$ similarly.

Then:

$$\begin{aligned}\Pr(x = \text{Head}) &= \lim_{n \rightarrow \infty} \frac{\sum_{i=1}^n \{1 \text{ if } x_i = \text{Head}, 0 \text{ else}\}}{n} \\ \Pr(x = \text{Tail}) &= \lim_{n \rightarrow \infty} \frac{\sum_{i=1}^n \{1 \text{ if } x_i = \text{Tail}, 0 \text{ else}\}}{n}\end{aligned}\quad (\text{A.22})$$

Clearly, $0 \leq \Pr(x = \text{Head}) \leq 1$, $0 \leq \Pr(x = \text{Tail}) \leq 1$ and $\Pr(x = \text{Head}) + \Pr(x = \text{Tail}) = 1$. If x is the random outcome of a *fair* coin flip, then: $\Pr(x = \text{Head}) = \Pr(x = \text{Tail}) = \frac{1}{2}$.

These concepts extend naturally to the any random experiment with a finite set of possible outcomes $Y = \{y_1, \dots, y_m\}$. Again, we use x to denote the outcome of a random experiment and $\{x_1, x_2, \dots, x_n\}$ to denote the outcomes of n experiments:

$$\begin{aligned}(\forall j : 1 \leq j \leq m) \Pr(x = y_j) &= \lim_{n \rightarrow \infty} \frac{\sum_{i=1}^n \{1 \text{ if } x_i = y_j, 0 \text{ else}\}}{n} \\ (\forall j : 1 \leq j \leq m) 0 \leq \Pr(x = y_j) &\leq 1 \\ \sum_{j=1}^m \Pr(x = y_j) &= 1\end{aligned}\quad (\text{A.23})$$

Note that probabilities are always between zero and one. Hence, it is often convenient to refer to probability as a percentage, which is between 0 and 100. In speaking language, we often omit the word ‘probability’ and only refer to the percentage, as in ‘I’m 100% sure I will break this system’.

We mostly deal with *uniform probabilities*, i.e., every value in $Y = \{y_1, \dots, y_m\}$ is equally likely. We use the notation $x \xleftarrow{\$} Y$ to denote that x is a random value chosen *uniformly* from the finite set Y .

For any predicate $\pi : Y \rightarrow \{\text{TRUE}, \text{FALSE}\}$, let $\Pr_{x \xleftarrow{\$} Y}(\pi(x))$ denote the probability that $\pi(x) = \text{TRUE}$ when x is chosen randomly from Y . For example, for the simple predicate $\pi_{y_j}(x) \equiv \{\text{TRUE if } x = y_j, \text{ FALSE otherwise}\}$, we have:

$$(\forall j : 1 \leq j \leq m) \Pr_{x \xleftarrow{\$} Y} \pi_{y_j} = \Pr_{x \xleftarrow{\$} Y} (x = y_j) = \frac{1}{m} = \frac{1}{|Y|} \quad (\text{A.24})$$

Note: we cannot choose uniformly from an infinite set. Lucky for us, in this textbook we only deal with finite sets; *it is impossible to uniformly select a value from an infinite set*. And we always look at uniform choice of random values.

Example A.2. Here are some simple probabilities, mostly calculated already. Try to understand the calculations, and then find the missing values.

- Let $\text{Die} = \{1, \dots, 6\}$ be the set of possible outcomes of a die roll. Then: $\Pr_{x \xleftarrow{\$} \text{Die}}(x = 5) = \frac{1}{6}$, $\Pr_{x \xleftarrow{\$} \text{Die}}(x > 3) = \frac{1}{2}$, $\Pr_{x \xleftarrow{\$} \text{Die}}(x \geq 3) =$

- $\Pr_{x \leftarrow \{00,01,10,11\}}(x = 00) = \frac{1}{4}$, $\Pr_{x,x' \leftarrow \{00,01,10,11\}}(x = x') =$

Note that in the second item, we refer to the probability of a predicate π which depends on two random variables, x' and x'' , chosen uniformly (and hence independently) from sets X' and X'' , i.e., $\Pr_{x' \leftarrow X', x'' \leftarrow X''}(\pi(x', x''))$. An important special case is when $\pi(x', x'') = \pi'(x') \wedge \pi''(x'')$, i.e., the predicate π is the conjunction of two predicates π' and π'' , which each depend only on one of the two inputs (x' and x'' , respectively). In this case, we have:

$$\Pr_{x' \leftarrow X', x'' \leftarrow X''}(\pi(x', x'')) = \Pr_{x' \leftarrow X'}(\pi'(x')) \cdot \Pr_{x'' \leftarrow X''}(\pi''(x''))$$

Namely:

$$\Pr_{x' \leftarrow X', x'' \leftarrow X''}(\pi'(x') \wedge \pi''(x'')) = \Pr_{x' \leftarrow X'}(\pi'(x')) \cdot \Pr_{x'' \leftarrow X''}(\pi''(x'')) \quad (\text{A.25})$$

Equation A.25 is often useful. In particular, often the same predicate π is applied to both variables x', x'' ; and often x' and x'' are chosen (uniformly) from the same set. This gives the following special cases:

$$\Pr_{x' \leftarrow X', x'' \leftarrow X''}(\pi(x') \wedge \pi(x'')) = \Pr_{x' \leftarrow X'}(\pi(x')) \cdot \Pr_{x'' \leftarrow X''}(\pi(x'')) \quad (\text{A.26})$$

$$\Pr_{x', x'' \leftarrow X}(\pi'(x') \wedge \pi''(x'')) = \Pr_{x' \leftarrow X}(\pi'(x')) \cdot \Pr_{x'' \leftarrow X}(\pi''(x'')) \quad (\text{A.27})$$

$$\Pr_{x', x'' \leftarrow X}(\pi(x') \wedge \pi(x'')) = \Pr_{x' \leftarrow X}(\pi(x')) \cdot \Pr_{x'' \leftarrow X}(\pi(x'')) \quad (\text{A.28})$$

Notice that Equation A.28 can be simplified, since $\Pr_{x' \leftarrow X}(\pi(x')) = \Pr_{x'' \leftarrow X}(\pi(x''))$, as we just use a different variable name in the two expressions (x'' and x'). Hence we have:

$$\Pr_{x', x'' \leftarrow X}(\pi(x') \wedge \pi(x'')) = \left(\Pr_{x \leftarrow X}(\pi(x)) \right)^2 \quad (\text{A.29})$$

Here is an exercise where you can make use of these equations.

Exercise A.8. The set $\{0,1\}^n$ contains the 2^n strings consisting of n -bits. Compute:

- $\Pr_{x_1 \leftarrow \{0,1\}, x_2 \leftarrow \{0,1\}}(x_1 || x_2 = 1^2) =$
- $\Pr_{x \leftarrow \{0,1\}^2}(x = 1^2) =$
- $\Pr_{x', x'' \leftarrow \{0,1\}^2}(x'' = x') =$
- $\Pr_{x \leftarrow \{0,1\}^n}(x = 1^n) =$

- $\Pr_{x', x'' \in \{0,1\}^2}(x'' = x') =$
- $\Pr_{x', x'' \in \{0,1\}^n}(x'' = x') =$
- $\Pr_{x' \in \{1, \dots, 10\}, x'' \in \{1, \dots, 12\}}((0 = x' \bmod 2) \wedge (0 = x'' \bmod 3)) =$

We are often interested in the outcome of a *randomized algorithm* \mathcal{A} , i.e., an algorithm that can perform a *bit flip* operation, i.e., a random choice from the set $\{0, 1\}$. We use the notation $\Pr(\pi(\mathcal{A}))$ to denote the probability that predicate π holds for the (randomized) output of \mathcal{A} , and the notation $y \leftarrow \mathcal{A}(x)$ to denote that y is assigned the random outcome of a uniformly-chosen run of \mathcal{A} with input x .

When we say a *uniformly-chosen run* we mean a run where the random bit-flips of algorithm \mathcal{A} are done fairly, i.e., each bit is chosen randomly (from $\{0, 1\}$).

Similarly, we often discuss the probability $\Pr_{x \in X}(\pi(\mathcal{A}(x)))$ that a predicate π holds, when applied over $\mathcal{A}(x)$, where x is chosen uniformly from the finite set X .

We often explicitly specify the predicate π as a small program, written in pseudocode. We will see examples later in this chapter; specifically, the pseudocode of Algorithm 1 is used as a predicate in Definition 1.6.

Note that $\Pr_{x \in X}(\pi(\mathcal{A}(x))) = \Pr_{y \in \mathcal{A}(x) | x \in X}(\pi(y))$. Also, note that if \mathcal{A} terminates after a number of steps, then its number of bit-flip operations is finite, and hence the set $Y \equiv \{y \in \mathcal{A}(x) | x \in X\}$ is finite. How lucky!

Example A.3. Let n, m be positive integers. Understand the first calculation and then solve the second one.

$$\begin{aligned} \Pr_{x \in \{1, \dots, 3 \cdot n\}}(\{\text{Return TRUE if } 0 = x \bmod 3, \text{ otherwise FALSE}\}) &= \frac{1}{n} \\ \Pr_{x \in \{1, \dots, m \cdot n\}}(\{\text{Return TRUE if } 1 = x \bmod m, \text{ otherwise FALSE}\}) &= \end{aligned}$$

A few additional basic facts about probabilities are worth noting; for details, proofs and more information, see, e.g., [88, 149].

Fact A.4 (Basic properties of probability). 1. Given predicate π , let $\bar{\pi}$ be the complementary predicate, i.e. $\pi(y) = \text{TRUE if and only if } \bar{\pi}(y) = \text{FALSE}$. Then for any algorithm \mathcal{A} and finite set X holds:

$$\Pr_{x \in X}(\bar{\pi}(\mathcal{A}(x))) = 1 - \Pr_{x \in X}(\pi(\mathcal{A}(x)))$$

2. The probability that predicate $\pi(y)$ does not hold for $y \leftarrow \mathcal{A}$, for n runs of \mathcal{A} , is $1 - (1 - \Pr(\pi(\mathcal{A})))^n$.

3. Let π, π' be two predicates and X, X' be two finite sets. Then:

$$\begin{aligned}
 & \Pr_{x \leftarrow X, x' \leftarrow X'} (\pi(x) \vee \pi'(x')) = \\
 &= \Pr_{x \leftarrow X} (\pi(x)) + \Pr_{x' \leftarrow X'} (\pi'(x')) - \Pr_{x, x' \leftarrow X} (\pi'(x') \wedge \pi''(x'')) \\
 &= \Pr_{x \leftarrow X} (\pi(x)) + \Pr_{x' \leftarrow X'} (\pi'(x')) - \Pr_{x' \leftarrow X} (\pi'(x')) \cdot \Pr_{x'' \leftarrow X} (\pi''(x''))
 \end{aligned} \tag{A.30}$$

Let us now give a more detailed exercise and its solution, as an example. Try to solve before reading the solution.

Exercise A.9. 1. Let $m_1 \neq m_2 \in \{0, 1\}^n$ be two randomly-chosen n bit strings, i.e., $m_1, m_2 \xleftarrow{\$} \{0, 1\}^n$.

- a) What is the probability that $m_1 = m_2$? If you cannot calculate this value, present the best bounds on it.
- b) What is the probability that every bit in m_1 is different from the corresponding bit of m_2 ? If you cannot calculate this value, present the best bounds on it.

2. Repeat, for three strings $m_1, m_2, m_3 \xleftarrow{\$} \{0, 1\}^n$:

- a) Compute or bound the probability that $m_1 = m_2 = m_3$.
 - b) Compute or bound the probability that every bit in m_1 is different from the corresponding bit of m_2 and from the corresponding bit of m_3 .
3. Repeat, when m_1 is chosen in arbitrary way - not necessarily randomly - before choosing randomly m_2 (and m_3).
4. Repeat, when m_1 is chosen in arbitrary way - not necessarily randomly - after choosing randomly m_2 (and m_3).

Solution:

1. a) Consider any m_1 . Then the probability that a random m_2 will have the same bit at any given position is half; the probability of this holding for all n bits is therefore 2^{-n} .
- b) This is exactly the same.
2. This is also exactly the same except for $\frac{1}{4}$ instead of half and 4^{-n} instead of 2^{-n} .
3. Also exactly the same.

4. For the case of two strings, we can intentionally choose a string to satisfy the requirement or not, so we only have the trivial bounds of 0 and 1 on the probability. For the case of three strings, we can still choose to *not* satisfy the requirement, so the probability may be as low as zero. However, we can only satisfy the requirement if $m_2 = m_3$ i.e. the maximal success probability is 2^{-n} .

Index

- $NP \stackrel{?}{=} P$, 588
(Optional) OCSP stapling, 520
pre-shared key, 452
cryptographic hash function, 112
 $2lM\mathcal{T}$, 201, 203–205
 $M\mathcal{T}$, 206, 207, 209
2PP, 274, 276–278, 280

recover security , 310

access control, 292
access-control, 477
accountability, 223
accumulator, 151, 159
accumulator scheme, 151
ACR, 165, 206
Adaptive Chosen Message Attack,
 27
Adaptive-CMA, 27
additional data, 417, 419
adoption challenge, 524
Advanced Encryption Standard, 95
advantage, 266, 273, 281
advantage function, 32
AEAD, 243, 244, 252, 267, 401, 402,
 405, 408, 417, 418, 422
AES, 54, 95, 327
Alan Turing, 72
AnG-CT, 537, 538, 540, 541, 543–
 549, 552, 553
anonymity, 210, 215, 358
anonymous voting, 358
ANSI X9.31, 143
any collision resistance, 165
ARL, 529

ASCII, 575
ASN.1, 488
asymmetric cryptography, 310, 311,
 319
asymmetric cryptosystem, 45
asymptotic security, 32, 75
AtE, 267, 401, 405, 407, 408
attack model, 5, 24, 26, 67
attack surface, 449, 450
attack vector, 449, 450
attacks, 7
audit, 549
audit protocol, 552
Audit-and-Gossip Certificate Transparency, 537, 540, 541, 544,
 548
Auth- h -DH, 346–348
Authenticate-then-Encrypt, 267, 401,
 405, 407
Authenticated Encryption, 243
authenticated encryption, 223, 243,
 405
Authenticated Encryption with Ad-
ditional Data, 418
Authenticated Encryption with As-
sociated Data, 244
authenticated encryption with asso-
ciated data, 243, 417
authenticated request/response pro-
tocols, 280
authenticated-encryption with asso-
ciated data, 252
authentication, 151, 223
authentication ceremony, 567
authenticator, 225

- authenticity, 4, 5
 Authorities Revocation List, 510
 Autokey, 55, 57
 availability, 4, 5, 223, 276
 backward compatibility, 443
 base-CRL, 510
 basic constraints, 498–500, 504
 BEAST attack, 126, 466
 Biased-Coin Extractor, 184
 bigram, 53, 54
 biometric, 568
 biometrics, 568
 birthday attack, 166, 167, 171, 181
 birthday paradox, 167
 Bitcoin, 212, 213
 bitwise randomness extractor, 185
 bitwise-randomness extractor, 186
 black-hat hacker, 40
 blacklists, 477
 Bleichenbacher, 368, 369, 372, 374, 375, 383, 450
 Bleichenbacher’s attack, 451
 block, 191
 block cipher, 54, 71, 98, 112, 232, 239, 404
 block ciphers, 46, 93
 block code, 122
 block-cipher, 77
 Blockchain, 202
 blockchain, 4, 41, 209
 Bombe, 60
 bounded key length stream cipher, 70
 BRE, 185, 186, 218
 BREACH, 403
 buffer overflow, 402
 CA, 378, 395, 421, 473, 474, 479
 CA certificates, 495
 CA/Browser Forum, 493
 CAA, 505, 531, 536
 Caesar cipher, 51
 cascade, 143, 588
 CBC, 126
 CBC-MAC, 114, 232–234
 CBC-MAC mode, 234
 CBIL, 235
 CCA, 62, 104, 108, 126, 372
 CCM, 114
 CCS, 403, 419
 CCSCA, *see* chosen-ciphertext side-channel attack
 ccTLD, 559
 CDH, 343, 352
 ceremony, 566
 certificate, 16, 395
 Certificate Authority, 3, 421, 473, 474
 certificate authority, 378, 395, 424, 479
 Certificate Authority Authorization, 505, 536
 certificate path, 497
 Certificate policy, 495
 certificate policy, 495
 Certificate Revocation List, 475, 508
 Certificate Revocations List, 505
 Certificate Status Request, 520
 Certificate Status Response, 519
 Certificate Transparency, 473, 474, 533, 535, 536, 538
 certification path, 492, 498
 Change Cipher Specification, 403, 419
 Checksum, 250
 chosen ciphertext attack, 104
 Chosen Message Attack, 27
 Chosen Message Attacker, 29
 Chosen Plaintext Attack, 65, 126
 chosen plaintext attack, 104, 265
 chosen-ciphertext attack, 62
 chosen-ciphertext side-channel attack, 58, 372–374
 chosen-message attack, 30
 Chosen-Plaintext Attack, 61
 chosen-plaintext attack, 61, 106
 chosen-prefix collision, 476
 chosen-prefix collisions, 160
 CIA, 223
 CIA triad, 4, 223
 CIAA, 223

- cipher, 9
cipher suite, 298, 402, 411
cipher suite negotiation, 298, 299
cipher-text only, 104
ciphertext, 48, 51, 59, 269
Ciphertext Only, 81
ciphertext only, 53
Ciphertext Only Attack, 57
Ciphertext-Only, 122
ciphertext-only, 59, 105, 265, 302
client authentication, 480
client certificate, 480
client-server, 292
ClientHello, 397
ClientFinished, 421
ClientHello, 420, 421
ClientKeyExchange, 421
cloud computing, 357
CMA, 27, 29, 30
co-prime, 595
collision, 153, 160
collision resistance, 17, 151, 152, 155, 174
Collision Resistant, 160
collision resistant, 154, 161
collision-resistance, 191
collision-resistant, 161, 164
collision-resistant hash, 41, 169
Collision-Resistant Hash Function, 170, 190
collision-resistant hash function, 155, 156, 172, 195
collisions, 101
Colossus, 37, 38
common name, 481–483
Compress-then-Encrypt Vulnerability, 252
compression function, 151, 152, 193
computational complexity, 18, 585
Computational DH, 343
computationally hard, 19
computer-aided cryptography, 7
concrete security, 32, 99
confidentiality, 4, 9, 223
congruence, 592
consensus, 213
conservative design, 17, 27, 29, 107, 414, 416
conservative design and usage, 36, 60
conservative design principle, 265
controlled blockchains, 210, 212
convolution code, 122
cookie, 62, 406, 414–416
cookies, 466
correctness, 95, 100
correctness requirement, 26
cost-benefit analysis, 8
counter-mode, 84
country code TLD, 559
CP, 495, 513
CPA, 61, 62, 65, 104, 106, 265
CPA-Oracle Attack, 58, 129, 406, 407, 409, 410, 412, 415, 416, 466
cracker, 40
CRC, 132, 217, 237
CRC-32, 133
CRHF, 151, 155–158, 160–164, 166–170, 172, 174, 175, 180, 190–195, 211, 218, 220, 221
CRIME, 403
critical, 489–491, 493, 524, 557, 558, 561, 562
criticality indicator, 490
CRL, 474, 475, 505–510, 513, 521, 525, 529, 549, 558
CRL Distribution Point, 509
CRL extension, 510
CRL issuer, 508
cRLDistributionPoints, 509
CRLsets, 529
cross site attack, 406
Cross Site Request Forgery, 416
cross-protocol, 463
cross-protocol attack, 465
cross-site attack model, 416
cross-site Denial-of-Service attack, 431
CRV, 529, 530, 558
cryptanalysis, 7
Crypto-agility, 398

- crypto-agility, 298, 299, 398, 399, 436
 cryptocurrency, 212, 215
 cryptographic backdoor, 394
 cryptographic building blocks, 239, 451
 cryptographic hash, 65
 cryptographic hash function, 152
 cryptographic trapdoor, 394
 cryptography, 9
 cryptology, 9
 cryptosystem, 9
 CSR, 519, 520
 CSRF, 416
 CSRF token, 416
 CSS, 12
 CT, 473, 474, 533, 535, 536
 CT-certificate, 536, 538, 539
 CTO, 53, 57–60, 62, 81, 104, 108, 109, 122, 129, 265, 302
 cyberpunk, 40
 cyberspace, 40
 cyclic group, 332
 Cyclic Redundancy Check, 237
 DANE, 536
 Data Encryption Standard, 37, 64, 95
 DDH, 344, 345, 353, 354, 356, 428
 DDH group, 345
 Decisional DH, 344, 345, 428
 decrypt-and-verify, 418
 Delta-CRL, 510
 Delta-CRV, 529, 530, 558
 deniability, 226
 Denial of Service, 153, 431, 453
 Denial-of-Service, 5, 153, 154, 472, 505, 507, 518, 535
 denial-of-service, 273
 DES, 37, 54, 64, 95
 detection, 3
 deterministic encryption, 368
 DH, 16, 310, 321, 328, 332
 DH key exchange, 319
 DH key exchange protocol, 351, 354
 DH PKC, 351–353
 DH protocol, 346
 DH public key cryptosystem, 351, 353
 DH-Ratchet, 346, 349
 dictionary attack, 570
 differential cryptanalysis, 95
 Diffie-Hellman, 16, 310, 321, 322, 328, 332, 341
 Diffie-Hellman Key Exchange, 37
 digest, 151, 159, 190, 191, 193, 545
 digest function, 197
 Digest scheme, 190
 Digest-Chain, 190
 digest-chain, 190, 191
 digital signature, 3, 320
 digital signature schemes, 378
 digital signatures, 42, 319
 digitized handwritten signatures, 16
 discrete logarithm, 325, 331, 332, 600
 discrete logarithm problem, 332
 Distinguished Name, 481, 482, 493, 508
 distinguished name, 482, 483
 Distinguished Names, 482
 DistributionPoint, 509, 510
 DN, 481, 482
 DNS, 396, 493, 496, 536
 dNSName, 493
 DNSsec, 536
 domain name, 477
 Domain Name System, 396, 493, 536
 domain name system, 496
 Domain Validation, 495, 496
 domain validation, 475, 532
 domain-validated, 533
 DoS, 5, 153, 154, 431, 453, 472, 505, 507, 535
 dot notation, 21
 Double Ratchet, 350
 downgrade, 393, 464
 downgrade attack, 297–299, 409, 420, 436, 438, 456
 downgrade attacks, 81
 downgrade dance, 469

- DTLS, 268, 401
Dual-EC Deterministic Random Bit Generator, 394
DV, 495, 496
dynamic pinning, 534
eavesdropping adversary, 266, 322, 323, 336
ECB, 126, 368
ECC, 109, 121, 250, 295
ECIES, 328
EDC, 121, 132, 237, 250
effective key length, 63, 66, 168
efficient, 23, 75
efficient algorithm, 587
efficiently-computable, 587
El-Gamal, 328, 332
El-Gamal PKC, 354
email address, 496
email validation, 496
Encode-then-Encrypt, 121, 122
encrypt-and-authenticate, 418
Encrypt-then-Authenticate, 252, 266, 269, 405, 408, 417
Encrypt-then-MAC, 417
Enigma, 36, 55, 60
entity authentication, 270–272
equivocating, 479
Error Correcting Code, 295
Error Correction Code, 250, 294
error correction code, 121
Error Detection Code, 3, 237, 250
error detection code, 121
error localization, 121
Error-Correcting Code, 109
error-detecting code, 132
error-detection code, 217
Eta, 249, 266, 269, 408, 417
Euler function, 365, 595
Euler’s Theorem, 365, 595
EV, 495, 496
evidence, 324
execution process, 260
exhaustive search, 18, 51, 59, 60, 63, 66, 328, 585
existential unforgeability, 30, 32
existential-unforgeability, 266
existentially unforgeable, 16, 33, 225, 266
existentially unforgeable signature, 169, 170, 172, 253
extend, 196
extend function, 196
extended CDH, 347
extended key usage, 490, 494
Extended Validation, 495, 496
extended-validation, 475
extract-then-expand, 186
factorial, 99
factoring, 325, 331, 589
Factors of Authentication, 568
fallback authentication, 568
Feedback Shift Register, 78
Feistel, 102
FHE, 357
FIL, 159
FIL-MAC, 228
Fixed Input Length, 159, 228
Flash Crowd, 518
Flat Merkle Tree, 201, 203, 204
Flat Merkle tree, 199
Flat Merkle Tree *PoC* is correct and secure., 205
Forward Secrecy, 307, 317
forward secrecy, 260, 306, 307, 309, 324
FREAK, 440
frequency analysis, 53, 55, 56
freshness, 232, 267, 284
FSR, 78
Fully Homomorphic Encryption, 357
gait, 568
game-based, 29
GCM, 114
General Monoalphabetic Substitution cipher, 52–54
Generalized-Caesar cipher, 51, 52, 55, 69
generic attack, 63
generic extractor, 184

- gossip, 546, 548, 556
 graphical passwords, 568, 571
 greatest common divisor, 595
 GSM, 12, 79, 109, 122, 138, 259, 292, 293
 GSM handshake protocol, 313
 hacker, 40
 Hamming code, 122
 Hamming distance, 122
 hard-fail, 512, 516, 562
 Hardware Security Module, 288, 306
 hash chain, 180, 181, 221
 hash function, 151, 239
 hash functions, 151
 hash-chain, 528
 Hash-then-Sign, 24, 25, 42, 43, 159, 169–172, 174–176, 183, 217, 329, 363, 379, 468, 476, 486, 488
 hashed, 25
 Hashed DH PKC, 351–353
 Hashed El-Gamal, 354
 Heartbleed Bug, 476
 Heartbleed bug, 509
 helper, 568
 HL-CT, 537–544, 548, 552, 563
 HMAC, 163, 218, 233, 237, 239, 240, 242, 462
 Honest-Logger Certificate Transparency, 537, 540–543
 HPKP, 534–536
 HSM, 288, 306
 HSTS, 446, 448
 Hts, 24, 169, 171, 175, 329, 379
 Hts, 174
 HTTP, 414
 HTTP public key pinning, 534
 HTTP Strict Transport Security, 448
 human-centered cryptography, 565
 human-centered security, 565
 hybrid encryption, 329, 364
 IAN, 493
 identity certificates, 477
 IDP, 510
 IDS, 3, 154
 IETF, 369
 IMSI, 292, 295
 IND-CCA, 118, 119, 127, 298
 IND-CCA1 secure, 491
 IND-CPA, 106–108, 112, 118, 126
 IND-CPA-PK, 109, 110
 independently pseudorandom, 95
 indistinguishability test, 70
 initialization vector, 143, 404
 integrity, 4, 123, 151, 223
 intermediate CA, 497
 International Mobile Subscriber Identifier, 292
 International Mobile Subscriber Identity, 295
 International Telecommunication Union, 480
 Intrusion Detection System, 3
 Intrusion-Detection Systems, 154
 Intrusion-Prevention Systems, 154
 invertible Pseudo-Random Permutation, 93
 IoT, 480
 IP address, 496
 IPS, 154
 IPsec, 267, 268
 ISO, 488
 issuer, 474
 Issuer Alternative Name, 493
 IssuerAltName, 493
 Issuing Distribution Point, 510
 ITU, 480, 488
 IV, 404
 KASUMI, 294
 KDC, 259, 290, 324, 573
 KDC protocol, 291
 KDC protocols, 290
 KDF, 183, 184, 187, 242, 346, 348, 349, 353, 462
 keepalive, 268
 Kerberos, 290
 Kerckhoffs' principle, 9
 Kerckhoffs's principle, 47

- Key Derivation Function, 184, 242, 346, 348, 462
key derivation function, 183
key derivation functions, 353
Key Distribution Center, 259, 290, 573
Key Distribution Center Protocol, 290
Key Distribution Protocol, 290
Key Exchange, 259, 260, 270, 274, 287–290, 292–297, 300–311, 349
key exchange, 319, 320, 323, 335
key generation, 25, 320
Key separation, 94
key separation, 302, 422, 465
key separation principle, 289, 463
key usage, 489–491, 494
key-exchange, 16, 310, 321, 324
key/logger, 571
Key-Only Attack, 26
key-usage, 464
Key Exchange, 397
keyed collision resistant, 163
keyed CRHF, 163, 165, 168, 239
keyed hash, 171
keyed-CRHF, 171
Keyed-HtS, 171, 172
keyless CRHF, 168, 170
keypair, 25
KeyUsage, 487
KISS principle, 449
KMA, 27
Known Message Attack, 27
Known Plaintext Attack, 53
Known plaintext attack, 51
known plaintext attack, 57, 265
known-plaintext, 59
Known-Plaintext Attack, 60
known-plaintext attack, 104
KPA, 51, 53, 55, 59, 60, 62, 104, 108, 109, 265
Legendre symbol, 334
length constraints, 530
letter-frequency attack, 59
Limited-use signatures, 34
linear cryptanalysis, 96
logger, 539
login, 567
login ceremony, 179, 568, 573
Lucky13, 408
MAC, 153, 190, 197, 223, 226, 228, 305, 319, 321, 346, 347, 379, 405
malware, 40, 41, 571
malware/virus scanners, 153
Man in the Middle, 335
Man-in-the-Browser, 571
Man-in-the-Middle, 17, 224, 226, 260, 261, 265, 299, 377, 406, 437, 515
master key, 287, 322, 346
matching, 273
Maximal Merge Delay, 545
MD-strengthening, 194
Merkle digest, 198
Merkle digest scheme, 4, 199, 201, 202, 545
Merkle Tree, 206, 207, 209
Merkle tree, 151, 159, 199, 544, 545
Merkle-Damgård, 151, 175, 176, 191
Message Authentication Code, 3, 5, 190, 223, 226, 319, 321, 346, 347
message recovery, 380
min-entropy, 184
mining, 213, 216
MitM, 17, 226, 260, 261, 265, 299, 303, 311, 324, 336, 342, 348, 377, 406, 437, 446, 496, 532, 536
MitM soft-fail, 519, 521–523
MMD, 545, 547, 550–552
mode of operation, 86, 112, 113
mode-of-operation, 243
Modular arithmetic, 590
modular arithmetic, 19, 590
modulo, 590
modulus, 590

- monoalphabetic substitution cipher, 52
- multiplicative homomorphic, 355, 356
- multiplicative inverse, 19
- Must-Staple, 521, 523, 524
- must-staple, 490, 560
- Mutual Entity Authentication, 270, 273–278, 289
- name constraint, 489, 500, 503, 504
- name constraints, 498, 530
- negligible, 32, 33
- nextUpdate, 509
- ngram, 53, 56
- ngrams, 56
- NIST, 95
- non-critical, 489, 490, 524, 561, 562
- non-repudiation, 4, 5, 15, 226, 377, 378
- nonce, 274, 277, 418
- NS-CT, 537–540, 543, 544, 549–556, 563
- NTTP, 537, 550
- NTTP-Security Certificate Transparency, 537, 538, 540, 549–551, 553–556
- OAEP, 362, 370–372, 375
- Object Identifier, 486
- object identifier, 487–489
- OCSP, 473, 505, 511, 512
- OCSP client, 511, 514
- OCSP Must-Staple, 519
- OCSP request, 511
- OCSP responder, 511, 512
- OCSP response, 511
- OCSP server, 514
- OCSP Stapling, 519
- OCSP stapling, 507, 518, 519
- OFB, 86
- OFB-mode, 84
- off-path, 17, 496
- offline dictionary attacks, 570
- OID, 486–489, 495
- old versions die slow, 436
- omitted-certificate attack, 546, 552, 555
- One Time Pad, 18, 25
- one time pad, 54, 67, 68, 70, 71, 76, 179, 585
- one time password, 179, 180
- One Way Function, 293
- one way function, 179, 180
- one-time pad, 56, 70, 370
- one-time password, 179, 572
- One-Time Signature, 34
- one-time signature scheme, 13
- one-time signatures, 169
- One-Time-Pad, 224
- One-Way Function, 178
- one-way function, 77, 169, 178
- one-way functions, 151
- OneCRL, 529
- Online Certificate Status Protocol, 511
- online dictionary attack, 570
- oracle, 89, 107
- oracle access, 23, 29, 106
- oracle notation, 26, 30
- Organization Validation, 495, 496
- organization validation, 475
- OTP, 18, 54, 67, 71, 72, 179, 224, 253, 585
- OTPw, 179
- Output Feedback, 86
- OV, 495, 496
- OWF, 151, 157, 169, 178, 179, 181, 218, 221, 293
- padded RSA, 366, 368
- padding attack, 404, 408, 409
- padding oracle, 129, 131, 407, 408
- Padding Oracle Attack, 129, 130
- Padding Oracle Attacks, 127
- Partially-Homomorphic Encryption, 358
- password, 568
- PBR, 113
- Per-Block Random, 113
- Per-goal keys, 94
- per-goal keys, 289

- Perfect Forward Secrecy, 307, 310, 312, 324, 346–348, 398, 429, 434
Perfect forward secrecy, 426
perfect forward secrecy, 260, 307, 430, 451, 459, 461, 535
Perfect Recover Security, 307, 311, 346, 348, 349
perfect recover security, 451
Perfect Recover Security , 310, 311
perfect recover security , 307, 311
Periodic Revocation Status, 529
permissioned, 212
permissionless, 212
permutation, 96
PFS, 260, 307, 310, 312, 324, 346–349, 398, 426, 429, 430, 434, 451, 454, 456, 459, 461, 535
PGP, 41, 560
PHE, 358
phishing, 533, 570
phishing email, 477
PKC, 16, 37, 109, 310, 319–321, 324, 326, 335, 473
PKC IND-CPA, 353
PKCS#1, 369, 375, 383
PKCS#5 padding, 128, 130, 131, 404, 408, 409
PKI, 5, 16, 378, 473, 478
PKI profile, 493
PKIX, 473, 485, 490–494, 498, 500, 504, 530, 533, 534, 536, 561
plaintext, 48–51, 59
plaintext-aware, 370
plaintext-awareness, 372
PoC, 201, 211, 545, 548
Pohlig-Hellman algorithm, 333
PoI, 151, 198–202, 204–208, 211, 525, 526, 539, 545, 546, 550–552
policy constraints, 498, 504
policy mapping, 504
polyalphabetic cipher, 55
polytime, 19, 586
PoM, 3, 548, 550, 552, 554–556
PoNI, 527, 528
Poodle, 404, 408, 409
Poodle padding attack, 445
Poodle version downgrade attack, 445
PoW, 160, 212, 213
PPT, 19, 23, 74, 75, 89, 173, 260, 265, 332, 333, 343, 586
Pre-Shared Key, 456
Pre-Shared Key modes, 459
Preimage resistance, 178
preimage resistant, 178
preimage-resistant hash, 178
preloaded, 448
premaster key, 426
PRF, 70, 71, 77, 82, 84–88, 96, 112, 141, 153, 185–187, 232, 242, 293, 295, 349, 387
PRG, 63, 70–72, 74, 75, 140, 143, 185, 186, 298, 310
PRG indistinguishability test, 72
principle of key separation, 231, 385
privacy, 358
private key, 320
Proactive security, 324
proactive security, 312
Probabilistic Polynomial Time, 23, 75
probabilistic polynomial time, 19, 586
probability, 600
Proof of Inclusion, 151
Proof of Misbehavior, 3
Proof of Non-Inclusion, 527, 528
proof of stake, 216
Proof-of-Consistency, 198, 201, 202, 210, 211, 545, 548
Proof-of-Inclusion, 198, 200, 206, 211, 525, 526, 539, 545, 555
Proof-of-Misbehavior, 548, 552, 554–556
Proof-of-Stake, 213
Proof-of-Work, 160, 212, 213
protocol, 566
provable security, 6
proxy re-encryption, 360

- PRP, 70, 71, 93, 96–98, 100, 232
 PRP/PRF switching lemma, 101, 112, 232
 PRS, 307, 310, 311, 346, 348, 349, 451
 Pseudo-Random Function, 387
 Pseudo-Random Generator, 63, 71, 72, 74, 75
 pseudo-random generator, 70–72, 74, 186, 310
 pseudo-random generators, 70
 Pseudo-Random Permutation, 96, 98, 232
 pseudo-random permutation, 70, 71, 96
 Pseudo-Random Permutations, 100
 pseudo-randomness, 70
 pseudorandom function, 70, 71, 77, 82, 84–88, 112, 186, 232, 242, 293, 295
 pseudorandom functions, 70, 83
 PSK, 452, 456
 PSS, 25
 public key, 3, 223, 225, 320
 public key certificate, 17, 324, 378, 474, 483
 public key certificates, 2, 176
 Public key cryptography, 324
 public key cryptography, 307, 319, 320, 335, 473
 Public Key Cryptology, 37
 public key cryptology, 310
 Public Key Cryptosystem, 319
 Public key cryptosystem, 320
 public key cryptosystem, 16, 45, 169, 320, 321
 public key cryptosystems, 109
 Public Key Infrastructure, 5, 473
 public key infrastructure, 16
 public key(s), 481
 public-key cryptography, 310
 public-key cryptosystem, 319, 321, 328, 335
 public-key encryption, 112
 public-key infrastructure, 378
 Punycode, 575
 quadratic residue, 90, 333, 344
 quantum computer, 326
 quantum computing, 182
 rainbow tables, 65
 random functions, 83, 96
 random functions., 82
 Random Oracle Methodology, 240
 Random Oracle Model, 6, 151, 188
 Random oracle model, 157
 random oracle model, 157, 163
 Random Oracle Model (ROM), 188
 random permutation, 96
 randomness extraction, 242
 randomness extractor, 158, 183, 346, 347, 353
 randomness-extractor hash, 353
 rational adversary, 3
 RC4, 79, 81, 82, 132–134
 RC4-dropN, 81
 rCCA, 360
 record protocol, 397
 Recover Security, 307, 348
 recover security, 260, 306, 307, 309
 Recover-Security Key Exchange, 309
 reduction, 352
 relying parties, 481
 relying party, 475, 489, 499
 renegotiation, 425, 452
 replay, 279
 replay-prevention, 232
 repudiation, 226
 Request-Response, 282
 request-response, 278, 282, 452
 Request-Response matching, 281
 residue, 590
 resiliency, 276
 resiliency to exposure, 324
 Resiliency to Exposures, 346
 Resiliency to key exposure, 306
 resiliency to key exposure, 310
 revocation transparency, 548
 revoked-certificates Merkle tree, 527
 RFC, 369
 robust combiner, 93, 237, 238, 398
 rogue certificate, 478, 496, 533

- Rogue certificates, 477
rogue certificates, 476
rogue website, 406
ROM, 6, 157, 163, 188, 240
root CA, 479, 497
root certificate authorities, 479
root program, 537
round trip time, 453
routing, 496
RR-matching, 281
RS-Ratchet, 349
RSA, 16, 67, 328, 331, 589
RSA assumption, 366, 367
RSA signatures, 380
RTT, 453
- S/Key, 180
S/MIME, 480
safe prime, 333, 339, 340, 346
safe prime group, 333
safe primes, 341
salt, 184, 187
Same-Origin-Policy, 477
SAN, 493, 554
scam, 533
scope, 510
SCSV, 445, 446, 454
SCT, 539–542, 544, 546, 550, 552, 562
second preimage resistant, 151
Second-Preimage Resistance, 172
second-preimage resistance, 163, 172
Second-Preimage Resistant, 163
second-preimage resistant, 173
Secret Sharing, 18, 585
secure in the standard model, 189
Secure Socket Layer, 391
securely matching, 273
security by obscurity, 35, 47
security parameter, 25, 31, 89, 106, 190, 191, 587
security requirements, 5
self-inverse function, 50
sender authentication, 232
sequence diagram, 21
sequence diagrams, 20
- Server Name Indication, 431
Server_Hello, 397
ServerFinished, 421
ServerHello, 421
session key, 322
session keys, 287
session resumption, 459
Session Ticket Encryption Key, 433, 434
session-ticket, 452
session/record, 263
session/record protocol, 287
SHA-1, 152
shared key, 223
shared key cryptosystem, 45
shift cipher, 51
shoulder surfing, 569
side channel, 127, 129, 569
side-channel, 372, 373, 518
Signaling Cipher Suite Value, 446
signature, 14, 25, 43
signature scheme, 5, 24, 25, 112, 223, 225, 321, 377
signature with appendix, 379
signature with message recovery, 379, 380
signcryption, 390, 494
signed malware, 477
Signed Transparency-tree Head, 539
Signed Tree Head, 545, 555
simulation-based, 29
smudge attacks, 570
SNA, 274–277, 313
SNA mutual-authentication protocol, 274
SNI, 431
Social engineering, 570
social engineering, 570
soft-fail, 512, 515–517, 521, 562
somebody-you-know, 573
SOP, 477
split world attack, 546
split-world, 553
split-world attack, 548, 553
SPR, 151, 157, 158, 163, 172–175, 178, 184, 218, 220, 221

- SSL, 128, 391, 473, 475, 477, 479, 532
 SSL-Stripping, 447, 448
 SSL-stripping, 447
 stapled-OCSP, 512
 static pinning, 534
 STEK, 434
 STH, 539, 545–548, 550, 552, 555
 stream cipher, 70, 71
 stream ciphers, 46
 SubjectAltName, 485, 493, 554
 subsequent certificates, 498, 504
 sufficient effective key length, 326
 Supported_Versions, 454
 symmetric cryptography, 290
 symmetric cryptosystem, 45, 327
 Systems Network Architecture, 274
- table look-up, 63
 table look-up attack, 65
 table lookup, 328
 Target Collision Resistant, 165, 168
 target collision resistant, 253
 Target Collision-Resistant, 172
 target collision-resistant, 165
 target-collision resistant, 171
 TCP, 267, 396
 TCP handshake, 397
 TCR, 165, 168, 171, 172, 206, 253
 TCR-HtS, 171, 172
 TDMA, 295
 text book RSA, 360
 textbook RSA, 357, 366, 368
 textbook-RSA, 361
 Threshold security, 324
 threshold security, 312
 TIME, 403
 time-memory tradeoff, 63, 65
 time/memory/data tradeoff, 65
 timing side channel, 127, 408
 TLDcc, 534
 TLS, 128, 263, 293, 307, 391, 473, 475, 477, 479, 520, 532
 TLS extension, 490, 519
 TLS feature, 490
- TLS record protocol, 263, 268, 287, 288
 TLS-feature, 521, 523
 TLS_FALLBACK_SCSV, 446
 to-be-signed, 487
 TOFU, 17, 377, 536
 top-level domain country codes, 534
 Tor, 41
 transaction fee, 216
 transcript, 261
 Transport-Layer Security, 391
 tree head, 545
 trust anchor, 497–499
 Trust On First Use, 536
 trust on first use, 17, 377
 trusted third party, 259, 290
 TTP, 259, 290
 Turing machine, 588
- unary, 89
 unconditionally secure, 18, 585
 Unicode, 575
 uniform distribution, 20
 universal one-way hash functions, 165
 universal re-encryption, 360
 Universal Resource Locator, 395
 URI, 509
 URL, 395
 user experience, 517
 user-experience, 515
 Using OCSP to validate a Certificate-Path, 513
 UX, 515, 517
 UX>Security, 517, 521, 523, 569
 UX>security, 515
- Variable Input Length, 228
 variable input length, 113
 Verify-Proof-of-Consistency, 202
 Vigenère cipher, 38, 47, 51, 55–57
 VIL, 113, 235, 239
 VIL-MAC, 228
 Virtual Private Network, 400
 visited network, 302
 Von Neumann extractor, 184

voting, 358
VPN, 400

weak collision resistance, 174
Web PKI, 473, 477, 480, 484, 532,
 534, 536
Web-of-Trust, 560
Web-of-trust, 497
website spoofing, 477
WebSocket, 414
WEP, 131, 237
white-hat hacker, 40
whitelists, 477
Wi-Fi Protected Access, 132
wildcard certificates, 493
wildcard character, 494
Wired Equivalency Privacy, 131
WPA, 132

X.500, 480, 482, 483
X.509, 473, 482
X.509v3, 473, 489
X9.17 padding, 404
X9.23 padding, 128–130, 408, 409

zombie certificate attack, 548, 549

Bibliography

- [1] J. Aas, March 2020.
- [2] I. Abraham, D. Malkhi, et al. The blockchain consensus layer and bft. *Bulletin of EATCS*, 3(123), 2017.
- [3] A. Acin, N. Gisin, and V. Scarani. Coherent-pulse implementations of quantum cryptography protocols resistant to photon-number-splitting attacks. *Phys. Rev. A*, 69:012309, Jan 2004.
- [4] A. Adams and M. A. Sasse. Users are not the enemy. *Communications of the ACM*, 42(12):40–46, 1999.
- [5] D. Adrian, K. Bhargavan, Z. Durumeric, P. Gaudry, M. Green, J. A. Halderman, N. Heninger, D. Springall, E. Thomé, L. Valenta, et al. Imperfect forward secrecy: How diffie-hellman fails in practice. *Communications of the ACM*, 62(1):106–114, 2018.
- [6] G. Agnew. Secure electronic transactions: Overview, capabilities, and current status, 2003. Chapter 10, ISBN 3-540-44007-0.
- [7] N. J. Al Fardan and K. G. Paterson. Lucky thirteen: Breaking the tls and dtls record protocols. In *2013 IEEE Symposium on Security and Privacy*, pages 526–540. IEEE, 2013.
- [8] W. Alexi, B. Chor, O. Goldreich, and C. P. Schnorr. RSA and Rabin functions: Certain parts are as hard as the whole. *SIAM Journal on Computing*, 17(2):194–209, 1988.
- [9] N. AlFardan, D. J. Bernstein, K. G. Paterson, B. Poettering, and J. C. Schuldt. On the security of rc4 in {TLS}. In *22nd {USENIX} Security Symposium ({USENIX} Security 13)*, pages 305–320, 2013.
- [10] ANSI. ANSI X9.23 financial institution encryption of wholesale financial messages, 1998.
- [11] R. Arends, R. Austein, M. Larson, D. Massey, and S. Rose. DNS Security Introduction and Requirements. RFC 4033 (Proposed Standard), Mar. 2005. Updated by RFCs 6014, 6840.

- [12] J.-P. Aumasson. *Serious Cryptography: A Practical Introduction to Modern Encryption*. No Stratch Press, 2017.
- [13] J.-P. Aumasson, S. Neves, Z. Wilcox-O’Hearn, and C. Winnerlein. Blake2: Simpler, smaller, fast as md5. In M. J. J. Jr., M. E. Locasto, P. Mohassel, and R. Safavi-Naini, editors, *Applied Cryptography and Network Security - 11th International Conference, ACNS 2013, Banff, AB, Canada, June 25-28, 2013. Proceedings*, volume 7954 of *Lecture Notes in Computer Science*, pages 119–135. Springer, 2013.
- [14] N. Aviram, S. Schinzel, J. Somorovsky, N. Heninger, M. Dankel, J. Steube, L. Valenta, D. Adrian, J. A. Halderman, V. Dukhovni, et al. {DROWN}: Breaking {TLS} using sslv2. In *25th {USENIX} Security Symposium ({USENIX} Security 16)*, pages 689–706, 2016.
- [15] A. J. Aviv, K. L. Gibson, E. Mossop, M. Blaze, and J. M. Smith. Smudge attacks on smartphone touch screens. In C. Miller and H. Shacham, editors, *4th USENIX Workshop on Offensive Technologies (WOOT’10)*. USENIX Association, 2010.
- [16] M. Badra. Pre-Shared Key Cipher Suites for TLS with SHA-256/384 and AES Galois Counter Mode. RFC 5487 (Proposed Standard), Mar. 2009.
- [17] M. Badra and I. Hajjeh. ECDHE.PSK Cipher Suites for Transport Layer Security (TLS). RFC 5489 (Informational), Mar. 2009.
- [18] M. Barbosa, G. Barthe, K. Bhargavan, B. Blanchet, C. Cremers, K. Liao, and B. Parno. Sok: Computer-aided cryptography. In *IEEE Symposium on Security and Privacy*, 2021.
- [19] G. V. Bard. A challenging but feasible blockwise-adaptive chosen-plaintext attack on SSL. In M. Malek, E. Fernández-Medina, and J. Hernando, editors, *Proceedings of SECRIPT*, pages 99–109. INSTICC Press, 2006.
- [20] E. Barkan, E. Biham, and N. Keller. Instant ciphertext-only cryptanalysis of GSM encrypted communication. *J. Cryptology*, 21(3):392–429, 2008.
- [21] E. Barker. Nist special publication 800-57 part 1 revision 4—recommendation for key management (part 1: General), 2016.
- [22] L. Bassham, W. Polk, and R. Housley. Algorithms and Identifiers for the Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile. RFC 3279 (Proposed Standard), Apr. 2002. Updated by RFCs 4055, 4491, 5480, 5758, 8692.
- [23] T. Be’ery and A. Shulman. A Perfect CRIME? Only TIME Will Only Tell. In *Blackhat Europe*, March 2013.

- [24] M. Bellare, R. Canetti, and H. Krawczyk. Keying hash functions for message authentication. In N. Koblitz, editor, *Advances in Cryptology—CRYPTO '96*, volume 1109 of *Lecture Notes in Computer Science*, pages 1–15. Springer-Verlag, 18–22 Aug. 1996.
- [25] M. Bellare, R. Canetti, and H. Krawczyk. Message authentication using hash functions: the HMAC construction. *CryptoBytes*, 2(1):12–15, Spring 1996.
- [26] M. Bellare, R. Canetti, and H. Krawczyk. HMAC: Keyed-hashing for message authentication. Internet Request for Comment RFC 2104, Internet Engineering Task Force, Feb. 1997.
- [27] M. Bellare, A. Desai, E. Jokipii, and P. Rogaway. A concrete security treatment of symmetric encryption. In *Proceedings 38th Annual Symposium on Foundations of Computer Science*, pages 394–403. IEEE, 1997.
- [28] M. Bellare, J. A. Garay, R. Hauser, A. Herzberg, H. Krawczyk, M. Steiner, G. Tsudik, E. V. Herreweghen, and M. Waidner. Design, implementation, and deployment of the iKP secure electronic payment system. *IEEE Journal of Selected Area in Communications*, 18(4):611–627, April 2000.
- [29] M. Bellare, J. A. Garay, R. Hauser, A. Herzberg, H. Krawczyk, M. Steiner, G. Tsudik, and M. Waidner. iKP: A family of secure electronic payment protocols. In *Proceedings of the First USENIX Workshop of Electronic Commerce*, pages 89–106, Berkeley, July 1995. USENIX Association.
- [30] M. Bellare, R. Guérin, and P. Rogaway. XOR MACs: New methods for message authentication using finite pseudorandom functions. In *Annual International Cryptology Conference*, pages 15–28. Springer, 1995.
- [31] M. Bellare, J. Kilian, and P. Rogaway. The security of the cipher block chaining message authentication code. *J. Comput. Syst. Sci.*, 61(3):362–399, 2000.
- [32] M. Bellare, T. Kohno, and C. Namprempre. Authenticated encryption in ssh: provably fixing the ssh binary packet protocol. In *Proceedings of the 9th ACM conference on Computer and Communications Security (CCS)*, pages 1–11, 2002.
- [33] M. Bellare, T. Krovetz, and P. Rogaway. Luby-rackoff backwards: Increasing security by making block ciphers non-invertible. In *Advances in Cryptology—Eurocrypt*, 1998.
- [34] M. Bellare and C. Namprempre. Authenticated encryption: Relations among notions and analysis of the generic composition paradigm. *Journal of Cryptology: the journal of the International Association for Cryptologic Research*, 21(4):469–491, Oct. 2008.

- [35] M. Bellare and P. Rogaway. Random oracles are practical: A paradigm for designing efficient protocols. In *Proceedings of the 1st ACM conference on Computer and communications security*, pages 62–73, 1993.
- [36] M. Bellare and P. Rogaway. Optimal asymmetric encryption. In *Workshop on the Theory and Application of Cryptographic Techniques*, pages 92–111. Springer, 1994.
- [37] M. Bellare and P. Rogaway. The exact security of digital signatures—how to sign with RSA and Rabin. In U. Maurer, editor, *Advances in Cryptology—EUROCRYPT 96*, volume 1070 of *Lecture Notes in Computer Science*, pages 399–416. Springer-Verlag, 12–16 May 1996.
- [38] M. Bellare and P. Rogaway. Collision-resistant hashing: Towards making UOWHFs practical. In *Annual International Cryptology Conference*, pages 470–484. Springer, 1997.
- [39] S. M. Bellovin. Frank Miller: Inventor of the One-Time Pad. *Cryptologia*, 35(3):203–222, 2011.
- [40] S. M. Bellovin. Vernam, Mauborgne, and Friedman: The One-Time Pad and the index of coincidence. In P. Y. A. Ryan, D. Naccache, and J.-J. Quisquater, editors, *The New Codebreakers*, volume 9100 of *Lecture Notes in Computer Science*, pages 40–66. Springer, 2016.
- [41] J. Benaloh, D. S. Butler Lampson, T. Spies, and B. Yee. The private communication technology (pct) protocol. Internet Draft, Microsoft Corporation, online at <https://datatracker.ietf.org/doc/html/draft-benaloh-pct-00>, Oct. 1995.
- [42] C. H. Bennett. Quantum cryptography using any two nonorthogonal states. *Phys. Rev. Lett.*, 68:3121–3124, May 1992.
- [43] C. H. Bennett and G. Brassard. Quantum cryptography: Public key distribution and coin tossing. In *Proceedings of IEEE International Conference on Computers, Systems and Signal Processing*, volume 175. New York, 1984.
- [44] Y. Berger, A. Wool, and A. Yeredor. Dictionary attacks using keyboard acoustic emanations. In *Proceedings of the 13th ACM conference on Computer and communications security*, pages 245–254, 2006.
- [45] D. J. Bernstein and T. Lange. Post-quantum cryptography. *Nature*, 549(7671):188–194, 2017.
- [46] D. J. Bernstein, T. Lange, and R. Niederhagen. Dual ec: A standardized back door. In *The New Codebreakers*, pages 256–281. Springer, 2016.

- [47] B. Beurdouche, K. Bhargavan, A. Delignat-Lavaud, C. Fournet, M. Kohlweiss, A. Pironti, P.-Y. Strub, and J. K. Zinzindohoue. A messy state of the union: Taming the composite state machines of tls. In *2015 IEEE Symposium on Security and Privacy*, pages 535–552. IEEE, 2015.
- [48] K. Bhargavan, A. D. Lavaud, C. Fournet, A. Pironti, and P. Y. Strub. Triple handshakes and cookie cutters: Breaking and fixing authentication over tls. In *2014 IEEE Symposium on Security and Privacy*, pages 98–113. IEEE, 2014.
- [49] R. Biddle, S. Chiasson, and P. C. Van Oorschot. Graphical passwords: Learning from the first twelve years. *ACM Computing Surveys (CSUR)*, 44(4):1–41, 2012.
- [50] R. Biddle, S. Chiasson, and P. C. Van Oorschot. Graphical passwords: Learning from the first twelve years. *ACM Computing Surveys (CSUR)*, 44(4):1–41, 2012.
- [51] E. Biham, B. Huttner, and T. Mor. Quantum cryptographic network based on quantum memories. *Physical Review A*, 54(4):2651, 1996.
- [52] C. Bing and J. SCHECTMAN. Inside the UAE’s secret hacking team of american mercenaries. Technical report, Reuters, January 2019.
- [53] R. Bird, I. Gopal, A. Herzberg, P. A. Janson, S. Kutten, R. Molva, and M. Yung. Systematic design of a family of attack-resistant authentication protocols. *IEEE Journal on Selected Areas in Communications*, 11(5):679–693, 1993.
- [54] H. Birge-Lee, Y. Sun, A. Edmundson, J. Rexford, and P. Mittal. Bamboozling certificate authorities with BGP. In *27th USENIX Security Symposium*, pages 833–849, 2018.
- [55] A. Biryukov and A. Shamir. Cryptanalytic time/memory/data tradeoffs for stream ciphers. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 1–13. Springer, 2000.
- [56] J. Black, P. Rogaway, and T. Shrimpton. Encryption-scheme security in the presence of key-dependent messages. In K. Nyberg and H. M. Heys, editors, *Selected Areas in Cryptography*, volume 2595 of *Lecture Notes in Computer Science*, pages 62–75. Springer, 2002.
- [57] S. Blake-Wilson, M. Nystrom, D. Hopwood, J. Mikkelsen, and T. Wright. Transport Layer Security (TLS) Extensions. RFC 3546 (Proposed Standard), June 2003. Obsoleted by RFC 4366.
- [58] S. Blake-Wilson, M. Nystrom, D. Hopwood, J. Mikkelsen, and T. Wright. Transport Layer Security (TLS) Extensions. RFC 4366 (Proposed Standard), Apr. 2006. Obsoleted by RFCs 5246, 6066, updated by RFC 5746.

- [59] M. Blaze, G. Bleumer, and M. Strauss. Divertible protocols and atomic proxy cryptography. In K. Nyberg, editor, *Advances in Cryptology - EUROCRYPT '98, International Conference on the Theory and Application of Cryptographic Techniques, Espoo, Finland, May 31 - June 4, 1998, Proceeding*, volume 1403 of *Lecture Notes in Computer Science*, pages 127–144. Springer, 1998.
- [60] M. Blaze, J. Feigenbaum, and A. D. Keromytis. Keynote: Trust management for public-key infrastructures. In *International Workshop on Security Protocols*, pages 59–63. Springer, 1998.
- [61] M. Blaze, J. Feigenbaum, and J. Lacy. Decentralized trust management. In *Proceedings 1996 IEEE Symposium on Security and Privacy*, pages 164–173. IEEE, 1996.
- [62] D. Bleichenbacher. Chosen ciphertext attacks against protocols based on the RSA encryption standard PKCS# 1. In *Advances in Cryptology - Crypto 1998*, volume 1462 of *Lecture Notes in Computer Science*, pages 1–12. Springer, 1998.
- [63] H. Böck, J. Somorovsky, and C. Young. Return of bleichenbacher’s oracle threat ({ROBOT}). In *27th {USENIX} Security Symposium ({USENIX} Security 18)*, pages 817–849, 2018.
- [64] H. Böck, A. Zauner, S. Devlin, J. Somorovsky, and P. Jovanovic. Nonce-disrespecting adversaries: Practical forgery attacks on {GCM} in {TLS}. In *10th {USENIX} Workshop on Offensive Technologies ({WOOT} 16)*, 2016.
- [65] D. Boneh. The decision diffie-hellman problem. In *International algorithmic number theory symposium*, pages 48–63. Springer, 1998.
- [66] N. Borisov, I. Goldberg, and D. Wagner. Intercepting mobile communications: The insecurity of 802.11. In *Proceedings of the Seventh Annual International Conference on Mobile Computing and Networking (MOBICOM-01)*, pages 180–188, New York, July 16–21 2001. ACM Press.
- [67] J. Brainard, A. Juels, R. L. Rivest, M. Szydlo, and M. Yung. Fourth-factor authentication: somebody you know. In *Proceedings of the 13th ACM conference on Computer and communications security*, pages 168–178, 2006.
- [68] M. Brandt, T. Dai, A. Klein, H. Shulman, and M. Waidner. Domain for mitm-resilient pki. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 2060–2076, 2018.
- [69] B. Brewster. Portfolio: Theory and practice. *The Yale Literary Magazine*, 47(5), Feb. 1882.

- [70] A. Broadbent, J. Fitzsimons, and E. Kashefi. Universal blind quantum computation. In *2009 50th Annual IEEE Symposium on Foundations of Computer Science*, pages 517–526. IEEE, 2009.
- [71] A. Broadbent and R. Islam. Quantum encryption with certified deletion. In *Theory of Cryptography Conference*, pages 92–122. Springer, 2020.
- [72] A. Broadbent and C. Schaffner. Quantum cryptography beyond quantum key distribution. *Designs, Codes and Cryptography*, 78(1):351–382, 2016.
- [73] BSI. Kryptographische verfahren: Empfehlungen und schlussell, February 2017.
- [74] R. Canetti, R. Gennaro, A. Herzberg, and D. Naor. Proactive security: Long-term protection against break-ins. *RSA Laboratories' CryptoBytes*, 3(1):1–8, 1997.
- [75] R. Canetti, H. Krawczyk, and J. Nielsen. Relaxing chosen-ciphertext security. In *Advances in Cryptology – Crypto 2003*, volume 2729, pages 565–582. Springer, 2003.
- [76] B. Canvel, A. Hiltgen, S. Vaudenay, and M. Vuagnoux. Password interception in a ssl/tls channel. In *CRYPTO - Annual International Cryptology Conference*, volume 2729 of *LNCS*, pages 583–599. Springer, 2003.
- [77] M. M. Carvalho, J. DeMott, R. Ford, and D. A. Wheeler. Heartbleed 101. *IEEE Secur. Priv.*, 12(4):63–67, 2014.
- [78] B. B. CCITT. Recommendations X. 509 and ISO 9594-8. *Information Processing Systems-OSI-The Directory Authentication Framework* (Geneva: CCITT), 1988.
- [79] M. CCITT. Specification of abstract syntax notation one (asn.1). *Open Systems Interconnection-Basic Reference Model*, 1988.
- [80] V. Cerf. ASCII format for network interchange. RFC 20 (Internet Standard), Oct. 1969.
- [81] D. Chadwick. *Understanding X. 500: the directory*. Chapman & Hall, Ltd., 1994.
- [82] S. Checkoway, M. Fredrikson, R. Niederhagen, A. Everspaugh, M. Green, T. Lange, T. Ristenpart, D. J. Bernstein, J. Maskiewicz, and H. Shacham. On the practical exploitability of dual EC in TLS implementations. In *Proceedings of the 23rd USENIX conference on Security Symposium*, pages 319–335. USENIX Association, 2014.
- [83] T. Chung, J. Lok, B. C. 0002, D. R. Choffnes, D. Levin, B. M. Maggs, A. Mislove, J. P. Rula, N. Sullivan, and C. Wilson. Is the web ready for ocsp must-staple? In *Internet Measurement Conference*, pages 105–118. ACM, 2018.

- [84] S. Cohney, M. D. Green, and N. Heninger. Practical state recovery attacks against legacy RNG implementations, October 2017. online at <https://duhkattack.com/paper.pdf>.
- [85] I. C. S. L. M. S. Committee. *IEEE 802.11: Wireless LAN Medium Access Control and Physical Layer Specifications*, Aug. 1999.
- [86] D. Cooper, S. Santesson, S. Farrell, S. Boeyen, R. Housley, and W. Polk. Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile. RFC 5280 (Proposed Standard), May 2008. Updated by RFCs 6818, 8398, 8399.
- [87] D. Coppersmith. Small solutions to polynomial equations, and low exponent rsa vulnerabilities. *Journal of cryptology*, 10(4):233–260, 1997.
- [88] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to algorithms*. MIT press, 2009.
- [89] S. A. Crosby and D. S. Wallach. Denial of service via algorithmic complexity attacks. In *Proceedings of the 12th USENIX Security Symposium*, pages 29–44. USENIX, aug 2003.
- [90] S. A. Crosby and D. S. Wallach. Efficient data structures for tamper-evident logging. In *USENIX Security Symposium*, pages 317–334, 2009.
- [91] J. Daemen and V. Rijmen. *The design of Rijndael: AES-the advanced encryption standard*. Springer Science & Business Media, 2013.
- [92] W. Dai. Crypto++ 6.0.0 benchmarks, 2018. version of 01-23-2018.
- [93] I. B. Damgård. Collision free hash functions and public key signature schemes. In *Workshop on the Theory and Application of Cryptographic Techniques*, pages 203–216. Springer, 1987.
- [94] I. B. Damgård. A design principle for hash functions. In G. Brassard, editor, *Advances in Cryptology—CRYPTO ’89*, volume 435 of *Lecture Notes in Computer Science*, pages 416–427. Springer-Verlag, 1990, 1989.
- [95] Q. Dang. Secure hash standard, August 2015.
- [96] A. Delignat-Lavaud, C. Fournet, M. Kohlweiss, J. Protzenko, A. Rastogi, N. Swamy, S. Zanella-Béguelin, K. Bhargavan, J. Pan, and J. K. Zinzindohoue. Implementing and proving the tls 1.3 record layer. In *2017 IEEE Symposium on Security and Privacy (SP)*, pages 463–482. IEEE, 2017.
- [97] Y. Desmedt. Threshold cryptosystems. In *International Workshop on the Theory and Application of Cryptographic Techniques*, pages 1–14. Springer, 1992.

- [98] T. Dierks and C. Allen. The TLS Protocol Version 1.0. RFC 2246 (Historic), Jan. 1999. Obsoleted by RFC 4346, updated by RFCs 3546, 5746, 6176, 7465, 7507, 7919.
- [99] T. Dierks and E. Rescorla. The Transport Layer Security (TLS) Protocol Version 1.1. RFC 4346 (Historic), Apr. 2006. Obsoleted by RFC 5246, updated by RFCs 4366, 4680, 4681, 5746, 6176, 7465, 7507, 7919.
- [100] T. Dierks and E. Rescorla. The Transport Layer Security (TLS) Protocol Version 1.2. RFC 5246 (Proposed Standard), Aug. 2008. Obsoleted by RFC 8446, updated by RFCs 5746, 5878, 6176, 7465, 7507, 7568, 7627, 7685, 7905, 7919, 8447, 9155.
- [101] W. Diffie and M. E. Hellman. New Directions in Cryptography. *IEEE Transactions on Information Theory*, IT-22(6):644–654, nov 1976.
- [102] R. Dingledine, N. Mathewson, and P. F. Syverson. Tor: The second-generation onion router. In M. Blaze, editor, *Proceedings of the 13th USENIX Security Symposium, August 9-13, 2004, San Diego, CA, USA*, pages 303–320. USENIX, 2004.
- [103] Y. Dodis, R. Gennaro, J. Håstad, H. Krawczyk, and T. Rabin. Randomness extraction and key derivation using the CBC, cascade and HMAC modes. In *Annual International Cryptology Conference*, pages 494–510. Springer, 2004.
- [104] D. Dolev, C. Dwork, and M. Naor. Nonmalleable cryptography. *SIAM review*, 45(4):727–784, 2003.
- [105] N. Doraswamy and D. Harkins. *IPSec: the new security standard for the Internet, intranets, and virtual private networks*. Prentice Hall Professional, 2003.
- [106] B. Dowling, F. Günther, U. Herath, and D. Stebila. Secure logging schemes and certificate transparency. In *European Symposium on Research in Computer Security*, pages 140–158. Springer, 2016.
- [107] O. Dubuisson. *ASN. 1 communication between heterogeneous systems*. Morgan Kaufmann, 2000.
- [108] V. Dunjko, J. F. Fitzsimons, C. Portmann, and R. Renner. Composable security of delegated quantum computation. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 406–425. Springer, 2014.
- [109] O. Dunkelman. *Techniques for cryptanalysis of block ciphers*. Thesis (ph.d.), Faculty of Computer Science, Technion — Israel Institute of Technology, Haifa, Israel, 2006.

- [110] T. Duong and J. Rizzo. Here come the XOR ninjas. Presented at Ecoparty and available at <http://www.hpc.ecs.soton.ac.uk/~dan/talks/bullrun/Beast.pdf>, 2011.
- [111] Z. Durumeric, J. Kasten, M. Bailey, and J. A. Halderman. Analysis of the https certificate ecosystem. In K. Papagiannaki, P. K. Gummadi, and C. Partridge, editors, *Proceedings of the 2013 Internet Measurement Conference, IMC 2013, Barcelona, Spain, October 23-25, 2013*, pages 291–304. ACM, 2013.
- [112] M. Dworkin. Recommendation for block cipher modes of operation: methods and techniques. Technical report, NIST (National Inst of Standards and Technology), 2001.
- [113] M. Dworkin. Recommendation for block cipher modes of operation: Galois/Counter Mode (GCM) and GMAC. Technical report, NIST (National Inst of Standards and Technology), November 2007.
- [114] M. J. Dworkin. Recommendation for block cipher modes of operation: The cmac mode for authentication. *Special Publication (NIST SP)-800-38B*, 2016.
- [115] J. Dyer. China Accused of Doling Out Counterfeit Digital Certificates in ‘Serious’ Web Security Breach. VICE News, April 2015.
- [116] S. Dziembowski and K. Pietrzak. Leakage-resilient cryptography. In *Foundations of Computer Science, 2008. FOCS’08. IEEE 49th Annual IEEE Symposium on*, pages 293–302. IEEE, 2008.
- [117] D. Eastlake 3rd. Transport Layer Security (TLS) Extensions: Extension Definitions. RFC 6066 (Proposed Standard), Jan. 2011. Updated by RFCs 8446, 8449.
- [118] P. Eronen (Ed.) and H. Tschofenig (Ed.). Pre-Shared Key Ciphersuites for Transport Layer Security (TLS). RFC 4279 (Proposed Standard), Dec. 2005. Updated by RFC 8996.
- [119] C. Evans, C. Palmer, and R. Sleevi. Public Key Pinning Extension for HTTP. RFC 7469 (Proposed Standard), Apr. 2015.
- [120] S. Even. *Graph algorithms*. Cambridge University Press, 2011.
- [121] L. Ewing. Linux 2.0 penguins. Example of using The GIMP graphics software, online; accessed 1-Sept-2017.
- [122] I. Fette and A. Melnikov. The WebSocket Protocol. RFC 6455 (Proposed Standard), Dec. 2011. Updated by RFCs 7936, 8307, 8441.
- [123] S. Fluhrer, I. Mantin, and A. Shamir. Weaknesses in the key scheduling algorithm of rc4. In *International Workshop on Selected Areas in Cryptography*, pages 1–24. Springer, 2001.

- [124] S. R. Fluhrer and D. A. McGrew. Statistical analysis of the alleged rc4 keystream generator. In *International Workshop on Fast Software Encryption*, pages 19–30. Springer, 2000.
- [125] R. Forno and W. Feinbloom. Inside risks: Pki: a question of trust and value. *Communications of the ACM*, 44(6):120, 2001.
- [126] T. C. forum. Baseline requirements for the issuance and management of publicly-trusted certificates. Technical Report 1.7.6, CABF, June 2021. available in: <https://cabforum.org/baseline-requirements-documents/>.
- [127] T. C. forum. Guidelines for the issuance and management of extended validation certificates. Technical Report 1.7.6, CABF, June 2021. available in: <https://cabforum.org/extended-validation/>.
- [128] T. C. forum. Object registry of the CA / browser forum, Viewed June 2021. available in: <https://cabforum.org/object-registry/>.
- [129] S. Frankel and S. Krishnan. IP Security (IPsec) and Internet Key Exchange (IKE) Document Roadmap. RFC 6071 (Informational), Feb. 2011.
- [130] A. Freier, P. Karlton, and P. Kocher. The Secure Sockets Layer (SSL) Protocol Version 3.0. RFC 6101 (Historic), Aug. 2011.
- [131] A. O. Frier, P. Karlton, and P. C. Kocher. The SSL protocol version 3.0. Online: <https://datatracker.ietf.org/doc/html/draft-ietf-tls-ssl-version3-00>, November 1996. Published as Internet Draft draft-ietf-tls-ssl-version3-00.
- [132] D. Gafurov, E. Snekkenes, and P. Bours. Spoof attacks on gait authentication system. *IEEE Trans. Inf. Forensics Secur.*, 2(3-2):491–502, 2007.
- [133] T. E. Gamal. A public key cryptosystem and a signature scheme based on discrete logarithms. *IEEE Trans. Inf. Theory*, 31(4):469–472, 1985.
- [134] S. Garfinkel and N. Makarevitch. *PGP: Pretty Good Privacy*. O'Reilly International Thomson, Paris, France, 1995.
- [135] C. Gidney and M. Ekerå. How to factor 2048 bit rsa integers in 8 hours using 20 million noisy qubits. *arXiv preprint arXiv:1905.09749*, 2019.
- [136] Y. Gilad, A. Herzberg, M. Sudkovich, and M. Goberman. Cdn-on-demand: An affordable ddos defense via untrusted clouds. In *Network and Distributed System Security Symposium (NDSS)*. The Internet Society, 2016.
- [137] D. Gillmor. Negotiated Finite Field Diffie-Hellman Ephemeral Parameters for Transport Layer Security (TLS). RFC 7919 (Proposed Standard), Aug. 2016.

- [138] D. Giry. Cryptographic key length recommendation, 2018. version of 01-23-2018.
- [139] Y. Gluck, N. Harris, and A. Prado. Breach: reviving the crime attack. *Black Hat USA*, 2013.
- [140] O. Goldreich. *Foundations of Cryptography*, volume Basic Tools. Cambridge University Press, 2001.
- [141] O. Goldreich. *Foundations of cryptography: volume 2, basic applications*. Cambridge university press, 2009.
- [142] O. Goldreich. *P, NP, and NP-Completeness: The Basics of Complexity Theory*. Cambridge University Press, 2010.
- [143] O. Goldreich, S. Goldwasser, and S. Micali. How to construct random functions. *J. ACM*, 33(4):792–807, 1986.
- [144] S. Goldwasser and S. Micali. Probabilistic encryption. *Journal of Computer and System Sciences*, 28(2):270–299, Apr. 1984.
- [145] S. Goldwasser, S. Micali, and R. L. Rivest. A digital signature scheme secure against adaptive chosen-message attacks. *SIAM Journal on computing*, 17(2):281–308, 1988.
- [146] P. Golle, M. Jakobsson, A. Juels, and P. Syverson. Universal re-encryption for mixnets. In *Cryptographers’ Track at the RSA Conference*, pages 163–178. Springer, 2004.
- [147] M. Goodwin. Revoking intermediate certificates: Introducing OneCRL. Mozilla Security Blog, <https://blog.mozilla.org/security/2015/03/03/revoking-intermediate-certificates-introducing-onecrl/>, Mar. 2015.
- [148] Google. Ev ui moving to page info, 2019. available in: <https://chromium.googlesource.com/chromium/src/+/HEAD/docs/security/ev-to-page-info.md>.
- [149] C. M. Grinstead and J. L. Snell. *Introduction to probability*. American Mathematical Soc., 2012.
- [150] L. K. Grover. A fast quantum mechanical algorithm for database search. In *Proceedings of the twenty-eighth annual ACM symposium on Theory of computing*, pages 212–219, 1996.
- [151] B. Gruber. How certificate transparency works. In <https://www.certificate-transparency.org/how-ct-works>, August 2013.
- [152] C. G. Günther. An identity-based key-exchange protocol. In *Workshop on the Theory and Application of of Cryptographic Techniques*, pages 29–37. Springer, 1989.

- [153] P. Gutmann. Encrypt-then-MAC for Transport Layer Security (TLS) and Datagram Transport Layer Security (DTLS). RFC 7366 (Proposed Standard), Sept. 2014.
- [154] C. Hall, D. Wagner, J. Kelsey, and B. Schneier. Building PRFs from PRPs. In *Annual International Cryptology Conference*, pages 370–389. Springer, 1998.
- [155] P. Hallam-Baker. X.509v3 Transport Layer Security (TLS) Feature Extension. RFC 7633 (Proposed Standard), Oct. 2015.
- [156] P. Hallam-Baker, R. Stradling, and J. Hoffman-Andrews. DNS Certification Authority Authorization (CAA) Resource Record. RFC 8659 (Proposed Standard), Nov. 2019.
- [157] N. Haller. The S/KEY One-Time Password System. RFC 1760 (Informational), Feb. 1995.
- [158] D. Hankerson, A. J. Menezes, and S. Vanstone. *Guide to elliptic curve cryptography*. Springer Science & Business Media, 2006.
- [159] M. E. Hellman. A cryptanalytic time-memory trade-off. *IEEE Trans. Information Theory*, 26(4):401–406, 1980.
- [160] A. Herzberg. Folklore, practice and theory of robust combiners. *Journal of Computer Security*, 17(2):159–189, 2009.
- [161] A. Herzberg. *Foundations of Cybersecurity, volume 2: An Internet-focused Introduction to Network Security*. Draft version available at https://www.researchgate.net/publication/331906855_Foundations_of_Cybersecurity_early_draft_of_part_II_Network_Security, 2021.
- [162] A. Herzberg. *Foundations Internet Security*. Draft version available at https://www.researchgate.net/publication/331906855_Foundations_of_Cybersecurity_early_draft_of_part_II_Network_Security, 2022.
- [163] A. Herzberg and A. Jbara. Security and Identification Indicators for Browsers Against Spoofing and Phishing Attacks. *ACM Trans. Internet Techn.*, 8(4):16:1–16:36, Oct. 2008.
- [164] A. Herzberg, H. Leibowitz, E. Syta, and S. Wrótniak. MoSS: Modular Security Specifications framework. In *CRYPTO' 2021*, pages 33–63, 2021. <https://eprint.iacr.org/2020/1040>.
- [165] A. Herzberg and R. Margulies. My authentication album: Adaptive images-based login mechanism. In *IFIP International Information Security Conference*, pages 315–326. Springer, 2012.
- [166] A. Herzberg and Y. Mass. Relying party credentials framework. *Electronic Commerce Research*, 4(1-2):23–39, 2004.

- [167] A. Herzberg, Y. Mass, J. Mihaeli, D. Naor, and Y. Ravid. Access control meets public key infrastructure, or: Assigning roles to strangers. In *Proceeding 2000 IEEE Symposium on Security and Privacy. S&P 2000*, pages 2–14. IEEE, 2000.
- [168] K. E. Hickman and T. Elgamal. The SSL protocol (version 2). Archived copy: <http://drive.google.com/file/d/1v0T0WazE667Y-vwfGws0GMow98DX3i7B/view?usp=sharing>, June 1995. Published as Internet Draft draft-hickman-netscape-ssl-01.txt.
- [169] J. Hodges, C. Jackson, and A. Barth. HTTP Strict Transport Security (HSTS). RFC 6797 (Proposed Standard), Nov. 2012.
- [170] P. Hoffman and J. Schlyter. The DNS-Based Authentication of Named Entities (DANE) Transport Layer Security (TLS) Protocol: TSLA. RFC 6698 (Proposed Standard), Aug. 2012. Updated by RFCs 7218, 7671, 8749.
- [171] J. Hoffstein, J. Pipher, J. H. Silverman, and J. H. Silverman. *An introduction to mathematical cryptography*. Springer, 2008.
- [172] R. Housley. Cryptographic Message Syntax (CMS). RFC 5652 (Internet Standard), Sept. 2009. Updated by RFC 8933.
- [173] R. Housley, W. Polk, W. Ford, and D. Solo. Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile. RFC 3280 (Proposed Standard), Apr. 2002. Obsoleted by RFC 5280, updated by RFCs 4325, 4630.
- [174] T. A. Howes, T. Howes, M. Smith, and G. S. Good. *Understanding and deploying LDAP directory services*. Addison-Wesley Professional, 2003.
- [175] P. Inglesant and M. A. Sasse. The true cost of unusable password policies: password use in the wild. In E. D. Mynatt, D. Schoner, G. Fitzpatrick, S. E. Hudson, W. K. Edwards, and T. Rodden, editors, *Proceedings of the 28th International Conference on Human Factors in Computing Systems, CHI 2010, Atlanta, Georgia, USA, April 10-15, 2010*, pages 383–392. ACM, 2010.
- [176] International Telecommunication Union. ITU-T X.509 recommendation version 3: Information technology - open systems interconnection - the directory: Authentication framework, June 1997.
- [177] International Telecommunication Union. Recommendation ITU-T X.509, OSI – The Directory: Public-Key and Attribute Certificate Frameworks, 2016.
- [178] I. S. O. (ISO). Data cryptographic techniques: Data integrity mechanism using a cryptographic check function employing a block cipher algorithm. ISO/IEC 9797, 1989.

- [179] T. Jager, J. Schwenk, and J. Somorovsky. On the security of tls 1.3 and quic against weaknesses in pkcs# 1 v1. 5 encryption. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 1185–1196, 2015.
- [180] M. Jakobsson, E. Stolterman, S. Wetzel, and L. Yang. Love and authentication. In *CHI '08: Proceeding of the twenty-sixth annual SIGCHI conference on Human factors in computing systems*, pages 197–200, New York, NY, USA, 2008. ACM.
- [181] J. Jean. TikZ for Cryptographers. <https://www.iacr.org/authors/tikz/>, 2016. CBC figures by Diana Maimut.
- [182] J. Jonsson and B. Kaliski. Public-Key Cryptography Standards (PKCS) #1: RSA Cryptography Specifications Version 2.1. RFC 3447 (Informational), Feb. 2003. Obsoleted by RFC 8017.
- [183] A. Joux. *Algorithmic cryptanalysis*. CRC Press, 2009.
- [184] D. Kahn. *The Codebreakers: The comprehensive history of secret communication from ancient times to the internet*. Simon and Schuster, 1996.
- [185] B. Kaliski. PKCS #1: RSA Encryption Version 1.5. RFC 2313 (Informational), Mar. 1998. Obsoleted by RFC 2437.
- [186] B. Kaliski and J. Staddon. PKCS #1: RSA Cryptography Specifications Version 2.0. RFC 2437 (Informational), Oct. 1998. Obsoleted by RFC 3447.
- [187] B. Kaliski Jr. A layman’s guide to a subset of ASN. 1, BER, and DER. RSA Laboratories technical note, November 1993.
- [188] J. Kelsey. Compression and information leakage of plaintext. In *International Workshop on Fast Software Encryption*, pages 263–276. Springer, 2002.
- [189] J. Kelsey, B. Schneier, D. Wagner, and C. Hall. Cryptanalytic attacks on pseudorandom number generators. In S. Vaudenay, editor, *Fast Software Encryption: 5th International Workshop*, volume 1372 of *Lecture Notes in Computer Science*, pages 168–188, Paris, France, 23–25 Mar. 1998. Springer-Verlag.
- [190] A. Kerckhoffs. La cryptographie militaire. *Journal des Sciences Militaires*, IX, 1883.
- [191] A. Kerckhoffs. La cryptographie militaire. *Journal des Sciences Militaires*, IX, 1883.
- [192] S. Kille. A String Representation of Distinguished Names. RFC 1779 (Historic), Mar. 1995. Obsoleted by RFCs 2253, 3494.

- [193] C. A. Kirtchev. A cyberpunk manifesto. *Cyberpunk Review*. Disponible en <http://www.cyberpunkreview.com/wiki/index.php>, 1997.
- [194] A. Klein. Attacks on the RC4 stream cipher. *Designs, Codes and Cryptography*, 48(3):269–286, 2008.
- [195] L. R. Knudsen and M. Robshaw. *The Block Cipher Companion*. Information Security and Cryptography. Springer, 2011.
- [196] N. Koblitz and A. Menezes. Critical perspectives on provable security: Fifteen years of "another look" papers. *Adv. Math. Commun*, 13(4):517–558, 2019.
- [197] L. M. Kohnfelder. *Towards a practical public-key cryptosystem*. Bachelor's thesis, Massachusetts Institute of Technology, 1978.
- [198] P. Koopman. 32-bit cyclic redundancy codes for internet applications. In *Proceedings 2002 International Conference on Dependable Systems and Networks (DSN 2002)*, pages 459–472, (Bethesda, MD) Washington, DC, USA, June 2002. IEEE Computer Society.
- [199] H. Krawczyk. The order of encryption and authentication for protecting communications (or: how secure is SSL?). In J. Kilian, editor, *Advances in Cryptology – CRYPTO '2001*, volume 2139 of *Lecture Notes in Computer Science*, pages 310–331. International Association for Cryptologic Research, Springer-Verlag, Berlin Germany, 2001.
- [200] H. Krawczyk. Cryptographic extraction and key derivation: The hkdf scheme. In T. Rabin, editor, *Advances in Cryptology - Proceedings of CRYPTO*, volume 6223 of *Lecture Notes in Computer Science*, pages 631–648. Springer, Aug. 2010. available from <https://ia.cr/2010/264>.
- [201] H. Krawczyk and P. Eronen. HMAC-based Extract-and-Expand Key Derivation Function (HKDF). RFC 5869 (Informational), May 2010.
- [202] J. F. Kurose and K. W. Ross. *Computer networking: a top-down approach*. Addison Wesley, 4th edition, 2009.
- [203] L. Lamport. Password authentication with insecure communication. *Communications of the ACM*, 24(11):770–772, 1981.
- [204] A. Langley. Enhancing digital certificate security. *Google Security Blog*, January 2013.
- [205] A. Langley. Certificate transparency: Comparison with other technologies. In <http://http://www.certificate-transparency.org/comparison>, July 2014.
- [206] A. Langley. Maintaining digital certificate security. *Google Security Blog*, January 2014.

- [207] A. Langley, M. Hamburg, and S. Turner. Elliptic Curves for Security. RFC 7748 (Informational), Jan. 2016.
- [208] J. Larisch, D. R. Choffnes, D. Levin, B. M. Maggs, A. Mislove, and C. Wilson. Crlite: A scalable system for pushing all tls revocations to all browsers. In *IEEE Symposium on Security and Privacy*, pages 539–556. IEEE Computer Society, 2017.
- [209] B. Laurie. Certificate transparency. *Communications of the ACM*, 57(10):40–46, 2014.
- [210] B. Laurie, A. Langley, and E. Kasper. Certificate Transparency. RFC 6962 (Experimental), June 2013. Obsoleted by RFC 9162.
- [211] B. Laurie, A. Langley, E. Kasper, E. Messeri, and R. Stradling. Certificate transparency version 2.0. *IETF TRANS (Public Notary Transparency) WG, Internet-Draft*, November 2019.
- [212] B. Laurie, G. Sisson, R. Arends, and D. Blacka. DNS Security (DNSSEC) Hashed Authenticated Denial of Existence. RFC 5155 (Proposed Standard), Mar. 2008. Updated by RFCs 6840, 6944, 9077, 9157.
- [213] H. Leibowitz, H. Ghalwash, E. Syta, and A. Herzberg. Ctng: Secure certificate and revocation transparency. Cryptology ePrint Archive, Report 2021/818, 2021. <https://ia.cr/2021/818>.
- [214] H. Leibowitz, A. Herzberg, and E. Syta. Provably model-secure pki schemes. Cryptology ePrint Archive, Report 2019/807, 2019. <https://eprint.iacr.org/2019/807>.
- [215] A. K. Lenstra and E. R. Verheul. Selecting cryptographic key sizes. *Journal of Cryptology*, 14(4):255–293, 2001.
- [216] G. Leurent and T. Peyrin. Sha-1 is a shambles: First chosen-prefix collision on sha-1 and application to the {PGP} web of trust. In *29th {USENIX} Security Symposium ({USENIX} Security 20)*, pages 1839–1856, 2020.
- [217] S. Levy. *Crypto: how the code rebels beat the government, saving privacy in the digital age*. Viking, 2001.
- [218] J. Leyden. Ca issues no-questions asked mozilla cert. *The Register*, December 2008.
- [219] N. Li, J. C. Mitchell, and W. H. Winsborough. Design of a role-based trust-management framework. In *Proceedings 2002 IEEE Symposium on Security and Privacy*, pages 114–130. IEEE, 2002.
- [220] M. Luby and C. Rackoff. How to construct pseudorandom permutations from pseudorandom functions. *SIAM Journal on Computing*, 17(2):373–386, Apr. 1988.

- [221] M. Lucamarini, G. Di Giuseppe, and K. Tamaki. Robust unconditionally secure quantum key distribution with two nonorthogonal and uninformative states. *Physical Review A*, 80(3):032327, 2009.
- [222] M. Lucamarini and S. Mancini. Secure deterministic communication without entanglement. *Physical review letters*, 94(14):140501, 2005.
- [223] J. Manger. A chosen ciphertext attack on rsa optimal asymmetric encryption padding (oaep) as standardized in pkcs# 1 v2. 0. In *Annual international cryptology conference*, pages 230–238. Springer, 2001.
- [224] I. Mantin and A. Shamir. A practical attack on broadcast RC4. In *International Workshop on Fast Software Encryption*, pages 152–164. Springer, 2001.
- [225] M. Marlinspike. More tricks for defeating ssl in practice. *Black Hat USA*, 2009.
- [226] J. Mason, K. Watkins, J. Eisner, and A. Stubblefield. A natural language approach to automated cryptanalysis of two-time pads. In *Proceedings of the 13th ACM conference on Computer and communications security*, pages 235–244. ACM, 2006.
- [227] A. J. Menezes, P. C. van Oorschot, and S. A. Vanston, editors. *Handbook of Applied Cryptography*. CRC Press, 1996.
- [228] W. Meng, L. W. Hao, M. S. Ramanujam, and S. P. T. Krishnan. Charging me and i know your secrets!: Towards juice filming attacks on smartphones. In J. Z. 0001 and D. Jones, editors, *Proceedings of the 1st ACM Workshop on Cyber-Physical System Security, CPSS 2015, Singapore*, pages 89–98. ACM, 2015.
- [229] J. Menn. Mysterious company with government ties plays key internet role. *The Washington Post*, November 2022.
- [230] R. C. Merkle. A certified digital signature. In G. Brassard, editor, *Advances in Cryptology—CRYPTO ’89*, volume 435 of *Lecture Notes in Computer Science*, pages 218–238. Springer-Verlag, 1990, Aug. 1989.
- [231] R. C. Merkle. One way hash functions and DES. In *Advances in Cryptology—CRYPTO ’89*, pages 428–446, Berlin - Heidelberg - New York, Aug. 1990. Springer.
- [232] C. Meyer and J. Schwenk. Lessons learned from previous SSL/TLS attacks - a brief chronology of attacks and weaknesses. *IACR Cryptology ePrint Archive*, 2013:49, 2013.
- [233] K. Mitnick. The testimony of an ex-hacker. appeared in Fronline, available online at <https://www.pbs.org/wgbh/pages/frontline/shows/hackers/whoare/testimony.html>, March 2000. Testimony to the U.S. Senate Committee on Governmental Affairs.

- [234] K. Mitnick and W. Simon. *Ghost in the Wires: My Adventures as the World's Most Wanted Hacker*. Back Bay Books, 2012.
- [235] A. Mittelbach and M. Fischlin. *The Theory of Hash Functions and Random Oracles: An Approach to Modern Cryptography*. Springer Verlag, February 2021.
- [236] B. Moeller. Security of CBC ciphersuites in SSL/TLS: Problems and countermeasures, May 2004.
- [237] B. Moeller and A. Langley. TLS Fallback Signaling Cipher Suite Value (SCSV) for Preventing Protocol Downgrade Attacks. RFC 7507 (Proposed Standard), Apr. 2015. Obsoleted by RFC 8996.
- [238] B. Möller, T. Duong, and K. Kotowicz. This POODLE bites: Exploiting the SSL 3.0 fallback, September 2014. Security advisory.
- [239] D. Moody, G. Alagic, D. C. Apon, D. A. Cooper, Q. H. Dang, J. M. Kelsey, Y.-K. Liu, C. A. Miller, R. C. Peralta, R. A. Perlner, et al. Status report on the second round of the nist post-quantum cryptography standardization process. Technical Report NISTIR 8309, NIST, 2020.
- [240] K. Moriarty (Ed.), B. Kaliski, J. Jonsson, and A. Rusch. PKCS #1: RSA Cryptography Specifications Version 2.2. RFC 8017 (Informational), Nov. 2016.
- [241] Mozilla. Removed ca certificate list. <https://ccadb-public.secure.force.com/mozilla/RemovedCACertificateReport>.
- [242] Mozilla. The MCS incident and its consequences for CNNIC. <https://blog.mozilla.org/security/files/2015/04/CNNIC-MCS.pdf>, April 2015.
- [243] M. Naor and M. Yung. Universal one-way hash functions and their cryptographic applications. In *Proceedings of the twenty-first annual ACM symposium on Theory of computing*, pages 33–43, 1989.
- [244] National Bureau of Standards. *Data Encryption Standard*. U. S. Department of Commerce, Washington, DC, USA, Jan. 1977.
- [245] National Institute of Standards and Technology. *FIPS PUB 186-2: Digital Signature Standard (DSS)*. National Institute for Standards and Technology, pub-NIST:adr, Jan. 2000.
- [246] H. Nejatollahi, N. Dutt, S. Ray, F. Regazzoni, I. Banerjee, and R. Cammarota. Post-quantum lattice-based cryptography implementations: A survey. *ACM Computing Surveys (CSUR)*, 51(6):1–41, 2019.
- [247] B. C. Neuman and T. Ts'o. Kerberos: An authentication service for computer networks. *IEEE Communications magazine*, 32(9):33–38, 1994.

- [248] M. A. Nielsen and I. Chuang. Quantum computation and quantum information, 2002.
- [249] J. Nightingale. Comodo certificate issue – follow up. *Mozilla Security Blog*, March 2011.
- [250] P. Norvig. English letter frequency counts: Mayzner revisited. Online at <https://norvig.com/mayzner.html>, 2013.
- [251] P. Oechslin. Making a faster cryptanalytic time-memory trade-off. In *Annual International Cryptology Conference*, pages 617–630. Springer, 2003.
- [252] O. Omolola, R. Roberts, M. I. Ashiq, T. Chung, D. Levin, and A. Mislove. Measurement and analysis of automated certificate reissuance. In *PAM*, pages 161–174, 2021.
- [253] OpenSSL. OpenSSL x509. OpenSSL documentation.
- [254] R. Oppliger. *SSL and TLS: Theory and Practice*. Artech House, 2016.
- [255] D. O’Brien, R. Sleevi, and A. Whalley. Chrome’s plan to distrust symantec certificates. *Google Security Blog*, September 2017.
- [256] C. Paar and J. Pelzl. *Understanding Cryptography - A Textbook for Students and Practitioners*. Springer, 2010.
- [257] D. Paez. How “pwned” went from hacker slang to the internet’s favorite taunt, March 2020.
- [258] S. Pasini and S. Vaudenay. Hash-and-sign with weak hashing made secure. In *Australasian Conference on Information Security and Privacy*, pages 338–354. Springer, 2007.
- [259] C. Peikert. A decade of lattice cryptography. *Foundations and Trends in Theoretical Computer Science*, 10(4):283–424, 2016.
- [260] Y. Pettersen. The Transport Layer Security (TLS) Multiple Certificate Status Request Extension. RFC 6961 (Proposed Standard), June 2013. Obsoleted by RFC 8446.
- [261] S. Pirandola, U. L. Andersen, L. Banchi, M. Berta, D. Bunandar, R. Colbeck, D. Englund, T. Gehring, C. Lupo, C. Ottaviani, et al. Advances in quantum cryptography. *Advances in Optics and Photonics*, 12(4):1012–1236, 2020.
- [262] S. Pohlig and M. Hellman. An improved algorithm for computing logarithms over $gf(p)$ and its cryptographic significance (corresp.). *IEEE Transactions on Information Theory*, 24(1):106–110, 1978.

- [263] J. Postel. Transmission Control Protocol. RFC 793 (Internet Standard), Sept. 1981. Updated by RFCs 1122, 3168, 6093, 6528.
- [264] J. Postel. Domain Name System Structure and Delegation. RFC 1591 (Informational), Mar. 1994.
- [265] A. Rabkin. Personal knowledge questions for fallback authentication: Security questions in the era of facebook. In *Proceedings of the 4th symposium on Usable privacy and security*, pages 13–23, 2008.
- [266] S. Radack. Secure hash standard: Updated specifications approved and issued as federal information processing standard (fips) 180-4. Technical report, National Institute of Standards and Technology, 2012.
- [267] R. S. Raman, L. Evdokimov, E. Wurstrow, J. A. Halderman, and R. Ensafi. Investigating large scale https interception in kazakhstan. In *Proceedings of the ACM Internet Measurement Conference*, pages 125–132, 2020.
- [268] B. Ramsdell and S. Turner. Secure/Multipurpose Internet Mail Extensions (S/MIME) Version 3.2 Message Specification. RFC 5751 (Proposed Standard), Jan. 2010. Obsoleted by RFC 8551.
- [269] F. Y. Rashid. Mozilla asked to revoke Trustwave CA for allowing SSL eavesdropping. *eWeek*, February 2012.
- [270] S. G. Renfro. Verisign CZAG: Privacy leak in x.509 certificates. In *11th USENIX Security Symposium (USENIX Security 02)*, San Francisco, CA, Aug. 2002. USENIX Association.
- [271] E. Rescorla. *SSL and TLS: designing and building secure systems*, volume 1. Addison-Wesley Reading, 2001.
- [272] E. Rescorla. The Transport Layer Security (TLS) Protocol Version 1.3. RFC 8446 (Proposed Standard), Aug. 2018.
- [273] E. Rescorla and N. Modadugu. Datagram Transport Layer Security Version 1.2. RFC 6347 (Proposed Standard), Jan. 2012. Obsoleted by RFC 9147, updated by RFCs 7507, 7905, 8996, 9146.
- [274] E. Rescorla, M. Ray, S. Dispensa, and N. Oskov. Transport Layer Security (TLS) Renegotiation Indication Extension. RFC 5746 (Proposed Standard), Feb. 2010.
- [275] E. Rescorla and M. Salter. Extended Random Values for TLS. Internet-Draft draft-rescorla-tls-extended-random-02, Internet Engineering Task Force, Mar. 2009. Work in Progress.
- [276] R. L. Rivest, A. Shamir, and L. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21(2):120–126, 1978.

- [277] J. Rizzo and T. Duong. Crime: Compression ratio info-leak made easy. In *ekoparty Security Conference*, 2012.
- [278] P. Rogaway. Problems with proposed ip cryptography. Unpublished manuscript, available at <https://cs.ucdavis.edu/~rogaway/papers/draft-rogaway-ipsec-comments-00.txt>, 1995.
- [279] P. Rogaway. Authenticated-encryption with associated-data. In V. Atluri, editor, *ACM Conference on Computer and Communications Security*, pages 98–107. ACM, November 2002.
- [280] P. Rogaway. Evaluation of some blockcipher modes of operation. *Cryptography Research and Evaluation Committees (CRYPTREC) for the Government of Japan*, 2011.
- [281] P. Rogaway, M. Bellare, and J. Black. OCB: A block-cipher mode of operation for efficient authenticated encryption. *ACM Transactions on Information and System Security (TISSEC)*, 6(3):365–403, 2003. Extending the version published in CCS’02.
- [282] P. Rogaway and T. Shrimpton. Cryptographic hash-function basics: Definitions, implications, and separations for preimage resistance, second-preimage resistance, and collision resistance. In *International workshop on fast software encryption*, pages 371–388. Springer, 2004.
- [283] E. Ronen, R. Gillham, D. Genkin, A. Shamir, D. Wong, and Y. Yarom. The 9 lives of bleichenbacher’s cat: New cache attacks on tls implementations. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 435–452. IEEE, 2019.
- [284] P. Saint-Andre and J. Hodges. Representation and Verification of Domain-Based Application Service Identity within Internet Public Key Infrastructure Using X.509 (PKIX) Certificates in the Context of Transport Layer Security (TLS). RFC 6125 (Proposed Standard), Mar. 2011.
- [285] J. Salowey, A. Choudhury, and D. McGrew. AES Galois Counter Mode (GCM) Cipher Suites for TLS. RFC 5288 (Proposed Standard), Aug. 2008.
- [286] J. Salowey, H. Zhou, P. Eronen, and H. Tschofenig. Transport Layer Security (TLS) Session Resumption without Server-Side State. RFC 5077 (Proposed Standard), Jan. 2008. Obsoleted by RFC 8446, updated by RFC 8447.
- [287] S. Santesson, M. Myers, R. Ankney, A. Malpani, S. Galperin, and C. Adams. X.509 Internet Public Key Infrastructure Online Certificate Status Protocol - OCSP. RFC 6960 (Proposed Standard), June 2013. Updated by RFC 8954.

- [288] J. Schaad, B. Kaliski, and R. Housley. Additional Algorithms and Identifiers for RSA Cryptography for use in the Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile. RFC 4055 (Proposed Standard), June 2005. Updated by RFC 5756.
- [289] S. Schechter, A. B. Brush, and S. Egelman. It's no secret. measuring the security and reliability of authentication via "secret" questions. In *2009 30th IEEE Symposium on Security and Privacy*, pages 375–390. IEEE, 2009.
- [290] S. Schechter, S. Egelman, and R. W. Reeder. It's not what you know, but who you know: a social approach to last-resort authentication. In *CHI '09: Proceedings of the 27th international conference on Human factors in computing systems*, pages 1983–1992, New York, NY, USA, 2009. ACM.
- [291] C. P. Schnorr. Fast factoring integers by svp algorithms. Cryptology ePrint Archive, Report 2021/232, 2021. <https://eprint.iacr.org/2021/232>.
- [292] C. E. Shannon. Communication theory of secrecy systems. *Bell System Technical Journal*, 28:656–715, Oct. 1949.
- [293] Y. Sheffer, R. Holz, and P. Saint-Andre. Summarizing Known Attacks on Transport Layer Security (TLS) and Datagram TLS (DTLS). RFC 7457 (Informational), Feb. 2015.
- [294] P. W. Shor. Algorithms for quantum computation: discrete logarithms and factoring. In *Proceedings 35th annual symposium on foundations of computer science*, pages 124–134. Ieee, 1994.
- [295] P. W. Shor and J. Preskill. Simple proof of security of the bb84 quantum key distribution protocol. *Physical review letters*, 85(2):441, 2000.
- [296] J. Simpson. An in-depth technical analysis of CurveBall (CVE-2020-0601). Published in Trend Micro security intelligence blog, online at:<https://blog.trendmicro.com/trendlabs-security-intelligence/an-in-depth-technical-analysis-of-curveball-cve-2020-0601/>, February 2020.
- [297] S. Singh. *The Science of Secrecy: The Secret History of Codes and Codebreaking*. Fourth Estate, London, UK, 2001.
- [298] N. P. Smart. *Cryptography Made Simple*. Information Security and Cryptography. Springer, 2016.
- [299] T. Smith, L. Dickenson, and K. E. Seamons. Let's revoke: Scalable global certificate revocation. In *NDSS*. The Internet Society, 2020.
- [300] E. Stark. Expect-CT Extension for HTTP. RFC 9163 (Experimental), June 2022.

- [301] M. Stevens, A. K. Lenstra, and B. de Weger. Chosen-prefix collisions for md5 and colliding x.509 certificates for different identities. In M. Naor, editor, *Advances in Cryptology - EUROCRYPT 2007, 26th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Barcelona, Spain, May 20-24, 2007, Proceedings*, volume 4515 of *Lecture Notes in Computer Science*, pages 1–22. Springer, 2007.
- [302] M. Stevens, A. Sotirov, J. Appelbaum, A. K. Lenstra, D. Molnar, D. A. Osvik, and B. de Weger. Short Chosen-Prefix Collisions for MD5 and the Creation of a Rogue CA Certificate. In S. Halevi, editor, *Advances in Cryptology - CRYPTO 2009*, volume 5677 of *Lecture Notes in Computer Science*, pages 55–69. Springer, 2009.
- [303] D. R. Stinson. *Cryptography: theory and practice*. CRC press, 2005.
- [304] G. Tsudik. Message authentication with one-way hash functions. *ACM SIGCOMM Computer Communication Review*, 22(5):29–38, 1992.
- [305] A. M. Turing. On computable numbers, with an application to the entscheidungsproblem. *Proceedings of the London Mathematical Society*, 42(2):230–265, 1936.
- [306] A. M. Turing. Computing machinery and intelligence. *Mind*, 59(236):433–460, Oct. 1950.
- [307] S. Turner, D. Brown, K. Yiu, R. Housley, and T. Polk. Updates for RSAES-OAEP and RSASSA-PSS Algorithm Parameters. RFC 5756 (Proposed Standard), Jan. 2010.
- [308] S. Turner and T. Polk. Prohibiting Secure Sockets Layer (SSL) Version 2.0. RFC 6176 (Proposed Standard), Mar. 2011. Updated by RFC 8996.
- [309] Unknown-author. A warm welcome to ASN.1 and DER. Part of Let’s Encrypt documentation.
- [310] M. Vanhoef. Fragment and forge: Breaking wi-fi through frame aggregation and fragmentation. In *30th {USENIX} Security Symposium ({USENIX} Security 21)*, pages 161–178, 2021.
- [311] M. Vanhoef and F. Piessens. Key reinstallation attacks: Forcing nonce reuse in WPA2. In B. M. Thuraisingham, D. Evans, T. Malkin, and D. Xu, editors, *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA*, pages 1313–1328. ACM, 2017.
- [312] S. Vaudenay. Security flaws induced by cbc padding—applications to ssl, ipsec, wtls... In *International Conference on the Theory and Applications of Cryptographic Techniques*, pages 534–545. Springer, 2002.

- [313] S. Venkata, S. Harwani, C. Pignataro, and D. McPherson. Dynamic Hostname Exchange Mechanism for OSPF. RFC 5642 (Proposed Standard), Aug. 2009.
- [314] G. S. Vernam. Secret signaling system, July 22 1919. US Patent 1,310,719.
- [315] J. Von Neumann. Various techniques used in connection with random digits. *Applied Math Series*, 12(36-38):1, 1951.
- [316] D. Wagner, B. Schneier, et al. Analysis of the ssl 3.0 protocol. In *The Second USENIX Workshop on Electronic Commerce Proceedings*, pages 29–40, 1996.
- [317] G. Wassermann. Savitech USB audio drivers install a new root CA certificate. Technical report, CERT Coordination Center, November 2017.
- [318] K. Watsen. NETCONF Call Home and RESTCONF Call Home. RFC 8071 (Proposed Standard), Feb. 2017.
- [319] N. Wiener. *Cybernetics, or control and communication in the animal and the machine*. John Wiley, New York, 1948.
- [320] Wikipedia. Block cipher mode of operation, 2017. [Online; accessed 1-Sept-2017].
- [321] Wikipedia contributors. Forward secrecy — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=Forward_secrecy&oldid=973196673, 2020. [Online; accessed 16-August-2020].
- [322] Wikipedia contributors. Diginotar — Wikipedia, the free encyclopedia. <https://en.wikipedia.org/w/index.php?title=DigiNotar&oldid=1036090956>, 2021. [Online; accessed 7-August-2021].
- [323] Wikipedia contributors. Dual ec drbg — Wikipedia, the free encyclopedia, 2021. [Online; accessed 14-August-2021].
- [324] Wikipedia contributors. Rc4 — Wikipedia, the free encyclopedia. <https://en.wikipedia.org/w/index.php?title=RC4&oldid=1058328794>, 2021. [Online; accessed 3-January-2022].
- [325] K. Wilson. Revoking trust in one ANSSI certificate. *Mozilla Security Blog*, December 2013.
- [326] R. J. Wilson. *Introduction to Graph Theory*. Pearson, New York, NY, 5 edition, 2010.
- [327] N. S. Yanofsky and M. A. Mannucci. *Quantum computing for computer scientists*. Cambridge University Press, 2008.
- [328] L. Zhang, D. R. Choffnes, T. Dumitras, D. Levin, A. Mislove, A. Schulman, and C. Wilson. Analysis of ssl certificate reissues and revocations in the wake of heartbleed. *Commun. ACM*, 61(3):109–116, 2018.

- [329] K. Zuse. Method for automatic execution of calculations with the aid of computers (1936). In B. Randell, editor, *The Origins of Digital Computers: Selected Papers*, Texts and monographs in computer science, pages 163–170. Springer-Verlag, pub-SV:adr, third edition, 1982.
- [330] M. Zusman. Criminal charges are not pursued: Hacking pki. *DEFCON* 17, 2009.
- [331] M. Zviran and W. J. Haga. Password security: an empirical study. *Journal of Management Information Systems*, 15(4):161–185, 1999.