

BOAZ BARAK

AN INTENSIVE
INTRODUCTION TO
CRYPTOGRAPHY

If you can just get your mind together
Then come on across to me
We'll hold hands, and then we'll watch the sunrise
From the bottom of the sea

Jimmy Hendrix, *Are You Experienced?*

Wednesday 11th April, 2018 13:14

Copyright © 2018 Boaz Barak

This work is licensed under a [Creative Commons “Attribution-NonCommercial-NoDerivatives 4.0 International” license](#).



Contents

	<i>Foreword and Syllabus</i>	17
	<i>Mathematical Background</i>	23
1	<i>Introduction</i>	47
2	<i>Computational Secrecy</i>	67
3	<i>Pseudorandomness</i>	85
4	<i>Pseudorandom functions</i>	101
5	<i>Pseudorandom functions from pseudorandom generators</i>	113
6	<i>Chosen Ciphertext Security</i>	127
7	<i>Hash functions and random oracles</i>	143
8	<i>Key derivation, protecting passwords, slow hashes, Merkle trees</i>	157

9	<i>Public key cryptography</i>	167
10	<i>Concrete candidates for public key crypto</i>	193
11	<i>Lattice based crypto</i>	205
12	<i>Chosen ciphertext security for public key encryption</i>	219
13	<i>Establishing secure connections over insecure channels</i>	221
14	<i>Zero knowledge proofs</i>	233
15	<i>Fully homomorphic encryption: Introduction and bootstrapping</i>	249
16	<i>Fully homomorphic encryption : Construction</i>	265
17	<i>Multiparty secure computation I: Definition and Honest-But-Curious to Malicious complier</i>	279
18	<i>Multiparty secure computation: Construction using Fully Homomorphic Encryption</i>	295
19	<i>Quantum computing and cryptography I</i>	305
20	<i>Quantum computing and cryptography II</i>	319
21	<i>Software Obfuscation</i>	331

22	<i>More obfuscation, exotic encryptions</i>	341
23	<i>Anonymous communication</i>	349
24	<i>Ethical, moral, and policy dimensions to cryptography</i>	351
25	<i>Course recap</i>	357
	<i>Bibliography</i>	363

Contents (detailed)

<i>Foreword and Syllabus</i>	17
0.1 <i>Syllabus</i>	18
0.1.1 <i>Prerequisites</i>	20
0.2 <i>Why is cryptography hard?</i>	21
<i>Mathematical Background</i>	23
0.3 <i>A quick overview of mathematical prerequisites</i>	24
0.4 <i>Mathematical Proofs</i>	25
0.4.1 <i>Example: The existence of infinitely many primes.</i>	26
0.5 <i>Probability and Sample spaces</i>	28
0.5.1 <i>Random variables</i>	31
0.5.2 <i>Distributions over strings</i>	33
0.5.3 <i>More general sample spaces.</i>	34
0.6 <i>Correlations and independence</i>	34
0.6.1 <i>Independent random variables</i>	37
0.6.2 <i>Collections of independent random variables.</i>	38
0.7 <i>Concentration</i>	39
0.7.1 <i>Chebyshev's Inequality</i>	41
0.7.2 <i>The Chernoff bound</i>	42
0.8 <i>Exercises</i>	43
1 <i>Introduction</i>	47
1.1 <i>Defining encryptions</i>	51
1.1.1 <i>Generating randomness in actual cryptographic systems</i>	53
1.2 <i>Defining the secrecy requirement.</i>	55
1.3 <i>Perfect Secrecy</i>	58
1.4 <i>Necessity of long keys</i>	63

1.4.1	<i>Advanced comment: Adding probability into the picture</i>	65
2	<i>Computational Secrecy</i>	67
2.0.1	<i>Proof by reduction</i>	71
2.1	<i>The asymptotic approach</i>	72
2.1.1	<i>Counting number of operations.</i>	74
2.2	<i>Our first conjecture</i>	74
2.3	<i>Why care about the cipher conjecture?</i>	76
2.4	<i>Prelude: Computational Indistinguishability</i>	77
2.5	<i>The Length Extension Theorem</i>	79
2.5.1	<i>Appendix: The computational model</i>	83
3	<i>Pseudorandomness</i>	85
3.1	<i>Stream ciphers</i>	89
3.2	<i>What do pseudorandom generators actually look like?</i>	91
3.2.1	<i>Attempt 0: The counter generator</i>	92
3.2.2	<i>Attempt 1: The linear checksum / linear feedback shift register (LFSR)</i>	92
3.2.3	<i>From insecurity to security</i>	94
3.2.4	<i>Attempt 2: Linear Congruential Generators with dropped bits</i>	95
3.3	<i>Successful examples</i>	96
3.3.1	<i>Case Study 1: Subset Sum Generator</i>	96
3.3.2	<i>Case Study 2: RC4</i>	97
3.4	<i>Non-constructive existence of pseudorandom generators</i>	98
4	<i>Pseudorandom functions</i>	101
4.1	<i>One time passwords (e.g., Google Authenticator, RSA ID, etc.)</i>	103
4.1.1	<i>How do pseudorandom functions help in the login problem?</i>	104
4.2	<i>Message Authentication Codes</i>	108
4.3	<i>MACs from PRFs</i>	110
4.4	<i>Input length extension for MACs and PRFs</i>	111
4.5	<i>Aside: natural proofs</i>	111
5	<i>Pseudorandom functions from pseudorandom generators</i>	113
5.1	<i>Securely encrypting many messages - chosen plaintext security</i>	117
5.2	<i>Pseudorandom permutations / block ciphers</i>	121
5.3	<i>Encryption modes</i>	123

6	<i>Chosen Ciphertext Security</i>	127
	6.1 <i>Short recap</i>	127
	6.2 <i>Going beyond CPA</i>	128
	6.2.1 <i>Example: The Wired Equivalence Protocol (WEP)</i>	128
	6.2.2 <i>Chosen ciphertext security</i>	130
	6.3 <i>Constructing CCA secure encryption</i>	133
	6.4 <i>(Simplified) GCM encryption</i>	138
	6.5 <i>Padding, chopping and their pitfalls: the “buffer overflow” of cryptography</i>	139
	6.6 <i>Chosen ciphertext attack as implementing metaphors</i>	140
7	<i>Hash functions and random oracles</i>	143
	7.1 <i>The “bitcoin” problem</i>	143
	7.1.1 <i>The currency problem</i>	143
	7.1.2 <i>Bitcoin architecture</i>	144
	7.2 <i>The bitcoin ledger</i>	145
	7.2.1 <i>From proof of work to consensus on ledger</i>	148
	7.3 <i>Collision resistance hash functions and creating short “unique” identifiers</i>	150
	7.4 <i>Practical constructions of cryptographic hash functions</i>	152
	7.4.1 <i>Practical random-ish functions</i>	153
	7.4.2 <i>Some history</i>	153
	7.4.3 <i>The NSA and hash functions.</i>	154
	7.4.4 <i>Cryptographic vs non-cryptographic hash functions:</i>	155
8	<i>Key derivation, protecting passwords, slow hashes, Merkle trees</i>	157
	8.1 <i>Keys from passwords</i>	157
	8.2 <i>Merkle trees and verifying storage.</i>	160
	8.3 <i>Proofs of Retrievability</i>	161
	8.4 <i>Entropy extraction</i>	162
	8.4.1 <i>Forward and backward secrecy</i>	166
9	<i>Public key cryptography</i>	167
	9.1 <i>Private key crypto recap</i>	169
	9.2 <i>Public Key Encryptions: Definition</i>	171
	9.2.1 <i>The obfuscation paradigm</i>	173
	9.3 <i>Some concrete candidates:</i>	175
	9.3.1 <i>Diffie-Hellman Encryption (aka El-Gamal)</i>	176
	9.3.2 <i>Sampling random primes</i>	180

9.3.3	<i>A little bit of group theory.</i>	181
9.3.4	<i>Digital Signatures</i>	182
9.3.5	<i>The Digital Signature Algorithm (DSA)</i>	184
9.4	<i>Putting everything together - security in practice.</i>	188
9.5	<i>Appendix: An alternative proof of the density of primes</i>	192
10	<i>Concrete candidates for public key crypto</i>	193
10.1	<i>Some number theory.</i>	193
10.1.1	<i>Primality testing</i>	194
10.1.2	<i>Fields</i>	195
10.1.3	<i>Chinese remainder theorem</i>	196
10.1.4	<i>The RSA and Rabin functions</i>	198
10.1.5	<i>Abstraction: trapdoor permutations</i>	199
10.1.6	<i>Public key encryption from trapdoor permutations</i>	199
10.1.7	<i>Digital signatures from trapdoor permutations</i>	203
10.2	<i>Hardcore bits and security without random oracles</i>	204
11	<i>Lattice based crypto</i>	205
11.1	<i>A world without Gaussian elimination</i>	207
11.2	<i>Security in the real world.</i>	209
11.3	<i>Search to decision</i>	210
11.4	<i>An LWE based encryption scheme</i>	211
11.5	<i>But what are lattices?</i>	215
11.6	<i>Ring based lattices</i>	216
12	<i>Chosen ciphertext security for public key encryption</i>	219
13	<i>Establishing secure connections over insecure channels</i>	221
13.1	<i>Cryptography's obsession with adjectives.</i>	221
13.2	<i>Basic Key Exchange protocol</i>	223
13.3	<i>Authenticated key exchange</i>	224
13.3.1	<i>Bleichenbacher's attack on RSA PKCS #1 V1.5 and SSL V3.0</i>	224
13.4	<i>Chosen ciphertext attack security for public key cryptography</i>	225
13.5	<i>CCA secure public key encryption in the Random Oracle Model</i>	227
13.5.1	<i>Defining secure authenticated key exchange</i>	229
13.5.2	<i>The compiler approach for authenticated key exchange</i>	230
13.6	<i>Password authenticated key exchange.</i>	231

13.7	<i>Client to client key exchange for secure text messaging - ZRTP, OTR, TextSecure</i>	231
13.8	<i>Heartbleed and logjam attacks</i>	231
14	<i>Zero knowledge proofs</i>	233
14.1	<i>Applications for zero knowledge proofs.</i>	234
14.1.1	<i>Nuclear disarmament</i>	234
14.1.2	<i>Voting</i>	235
14.1.3	<i>More applications</i>	235
14.2	<i>Defining and constructing zero knowledge proofs</i>	236
14.3	<i>Defining zero knowledge</i>	239
14.4	<i>Zero knowledge proof for Hamiltonicity.</i>	243
14.4.1	<i>Why is this interesting?</i>	246
14.5	<i>Parallel repetition and turning zero knowledge proofs to signatures.</i>	247
14.5.1	<i>"Bonus features" of zero knowledge</i>	247
15	<i>Fully homomorphic encryption: Introduction and bootstrapping</i>	249
15.1	<i>Defining fully homomorphic encryption</i>	252
15.1.1	<i>Another application: fully homomorphic encryption for verifying computation</i>	253
15.2	<i>Example: An XOR homomorphic encryption</i>	255
15.2.1	<i>Abstraction: A trapdoor pseudorandom generator.</i>	257
15.3	<i>From linear homomorphism to full homomorphism</i>	260
15.4	<i>Bootstrapping: Fully Homomorphic "escape velocity"</i>	260
15.4.1	<i>Radioactive legos analogy</i>	261
15.4.2	<i>Proving the bootstrapping theorem</i>	262
16	<i>Fully homomorphic encryption : Construction</i>	265
16.1	<i>Prelude: from vectors to matrices</i>	265
16.2	<i>Real world partially homomorphic encryption</i>	267
16.3	<i>Noise management via encoding</i>	268
16.4	<i>Putting it all together</i>	271
16.5	<i>Analysis of our scheme</i>	273
16.5.1	<i>Correctness</i>	274
16.5.2	<i>CPA Security</i>	274
16.5.3	<i>Homomorphism</i>	275
16.5.4	<i>Shallow decryption circuit</i>	275
16.6	<i>Example application: Private information retrieval</i>	278

17	<i>Multiparty secure computation I: Definition and Honest-But-Curious to Malicious complier</i>	279
	17.1 <i>Ideal vs. Real Model Security.</i>	280
	17.2 <i>Formally defining secure multiparty computation</i>	281
	17.2.1 <i>First attempt: a slightly “too ideal” definition</i>	281
	17.2.2 <i>Allowing for aborts</i>	282
	17.2.3 <i>Some comments:</i>	284
	17.3 <i>Example: Second price auction using bitcoin</i>	286
	17.3.1 <i>Another example: distributed and threshold cryptography</i>	287
	17.4 <i>Proving the fundamental theorem:</i>	289
	17.5 <i>Malicious to honest but curious reduction</i>	289
	17.5.1 <i>Handling probabilistic strategies:</i>	293
18	<i>Multiparty secure computation: Construction using Fully Homomorphic Encryption</i>	295
	18.1 <i>Constructing 2 party honest but curious computation from fully homomorphic encryption</i>	296
	18.2 <i>Achieving circuit privacy in a fully homomorphic encryption</i>	301
	18.3 <i>Bottom line: A two party honest but curious two party secure computation protocol</i>	303
19	<i>Quantum computing and cryptography I</i>	305
	19.0.1 <i>Quantum computing and computation - an executive summary.</i>	308
	19.1 <i>Quantum 101</i>	309
	19.1.1 <i>Physically realizing quantum computation</i>	313
	19.1.2 <i>Bra-ket notation</i>	314
	19.1.3 <i>Bell’s Inequality</i>	314
	19.2 <i>Grover’s Algorithm</i>	317
20	<i>Quantum computing and cryptography II</i>	319
	20.1 <i>From order finding to factoring and discrete log</i>	319
	20.2 <i>Finding periods of a function: Simon’s Algorithm</i>	320
	20.3 <i>From Simon to Shor</i>	322
	20.3.1 <i>The Fourier transform over \mathbb{Z}_m</i>	323
	20.3.2 <i>Quantum Fourier Transform over \mathbb{Z}_m</i>	324
	20.4 <i>Shor’s Order-Finding Algorithm.</i>	325
	20.4.1 <i>Analysis: the case that $r m$</i>	326

20.5	<i>Rational approximation of real numbers</i>	328
20.5.1	<i>Quantum cryptogrpahy</i>	329
21	<i>Software Obfuscation</i>	331
21.1	<i>Witness encryption</i>	332
21.2	<i>Deniable encryption</i>	333
21.3	<i>Functional encryption</i>	333
21.4	<i>The software patch problem</i>	335
21.5	<i>Software obfuscation</i>	335
21.6	<i>Applications of obfuscation</i>	336
21.7	<i>Impossiblity of obfuscation</i>	337
21.7.1	<i>Proof of impossiblity of VBB obfuscation</i>	337
21.8	<i>Indistinguishability obfuscation</i>	340
22	<i>More obfuscation, exotic encryptions</i>	341
22.1	<i>Slower, weaker, less securer</i>	341
22.2	<i>How to get IBE from pairing based assumptions.</i>	343
22.3	<i>Beyond pairing based cryptography</i>	345
23	<i>Anonymous communication</i>	349
23.1	<i>Steganography</i>	349
23.2	<i>Anonymous routing</i>	350
23.3	<i>Tor</i>	350
23.4	<i>Telex</i>	350
23.5	<i>Riposte</i>	350
24	<i>Ethical, moral, and policy dimensions to cryptography</i>	351
24.1	<i>Reading prior to lecture:</i>	353
24.2	<i>Case studies.</i>	353
24.2.1	<i>The Snowden revelations</i>	354
24.2.2	<i>FBI vs Apple case</i>	354
24.2.3	<i>Juniper backdoor case and the OPM break-in</i>	355
25	<i>Course recap</i>	357
25.1	<i>Some things we did not cover</i>	359
25.2	<i>What I hope you learned</i>	361

Foreword and Syllabus

"Human ingenuity cannot concoct a cipher which human ingenuity cannot resolve." Edgar Allan Poe,
1841

Cryptography - the art or science of “secret writing” - has been around for several millenia, and for almost all of that time Edgar Allan Poe’s quote above held true. Indeed, the history of cryptography is littered with the figurative corpses of cryptosystems believed secure and then broken, and sometimes with the actual corpses of those who have mistakenly placed their faith in these cryptosystems. Yet, something changed in the last few decades. New cryptosystems have been found that have not been broken despite being subjected to immense efforts involving both human ingenuity and computational power on a scale that completely dwarves the “crypto breakers” of Poe’s time. Even more amazingly, these cryptosystems are not only seemingly unbreakable, but they also achieve this under much harsher conditions. Not only do today’s attackers have more computational power but they also have more data to work with. In Poe’s age, an attacker would be lucky if they got access to more than a few ciphertexts with known plaintexts. These days attackers might have massive amounts of data - terabytes or more - at their disposal. In fact, with *public key* encryption, an attacker can generate as many ciphertexts as they wish.

These new types of cryptosystems, both more secure and more versatile, have enabled many applications that in the past were not only impossible but in fact *unimaginable*. These include secure communication without sharing a secret, electronic voting without a trusted authority, anonymous digital cash, and many more. Cryptography now supplies crucial infrastructure without which much of the modern “communication economy” could not function.

This course is about the story of this cryptographic revolution.

However, beyond the cool applications and the crucial importance of cryptography to our society, it contains also intellectual and mathematical beauty. To understand these often paradoxical notions of cryptography, you need to think differently, adapting the point of view of an attacker, and (as we will see) sometimes adapting the points of view of other hypothetical entities. More than anything, this course is about this cryptographic way of thinking. It may not be immediately applicable to protecting your credit card information or to building a secure system, but learning a new way of thinking is its own reward.

0.1 Syllabus

In this fast-paced course, I plan to start from the very basic notions of cryptogrpahy and by the end of the term reach some of the exciting advances that happened in the last few years such as the construction of *fully homomorphic encryption*, a notion that Brian Hayes called “one of the most amazing magic tricks in all of computer science”, and *indistinguishability obfuscators* which are even more amazing. To achieve this, our focus will be on *ideas* rather than *implementations* and so we will present cryptographic notions in their pedagogically simplest form– the one that best illustrates the underlying concepts– rather than the one that is most efficient, widely deployed, or conforms to Internet standards. We will discuss some examples of practical systems and attacks, but only when these serve to illustrate a conceptual point.

Depending on time, I plan to cover the following notions:

- Part I: Introduction
 1. **How do we define security for encryption?** Arguably the most important step in breaking out of the “build-break-tweak” cycle that Poe’s quote described has been the idea that we can have a *mathematically precise definition* of security, rather than relying on fuzzy notions, that allow us only to determine with certainty that a system is *broken* but never have a chance of *proving* that a system is *secure*.
 2. **Perfect security and its limitations:** Showing the possibility (and the limitations) of encryptions that are perfectly secure regardless of the attacker’s computational resources.
 3. **Computational security:** Bypassing the above limitations by restricting to computationally efficient attackers. Proofs of

security by reductions.

Part II: Private Key Cryptography

- 1. **Pseudorandom generators:** The basic building block of cryptography, which also provided a new twist on the age-old philosophical and scientific question of the nature of randomness.
- 2. **Pseudorandom functions, permutations, block ciphers:** Block ciphers are the working horse of crypto.
- 3. **Authentication and active attacks:** *Authentication* turns out to be as crucial, if not more, to security than *secrecy* and often a precondition to the latter. We'll talk about notions such as Message Authentication Codes and Chosen-Ciphertext-Attack secure encryption, as well as real-world examples why these notions are necessary.
- 4. **Hash functions and the “Random Oracle Model”:** Hash functions are used all over in crypto, including for verifying integrity, entropy distillation, and many other cases.
- 5. **Building pseudorandom generators from one-way permutations (optional):** Justifying our “axiom” of pseudo-random generators by deriving it from a weaker assumption.
- Part III: Public key encryption
 - 1. **Public key cryptography and the obfuscation paradigm:** How did Diffie, Hellman, Merkle, Ellis even dare to *imagine* the possibility of public key encryption?
 - 2. **Constructing public key encryption: Factoring, discrete log, and lattice based systems:** We'll discuss several variants for constructing public key systems, including those that are widely deployed such as RSA, Diffie-Hellman, and the elliptic curve variants, as well as some variants of *lattice based cryptosystems* that have the advantage of not being broken by quantum computers, as well as being more versatile. The former is the reason why the NSA has advised people to transition to lattice-based cryptosystems in the not too far future.
 - 3. **Signature schemes:** These are the public key versions of authentication though interestingly are easier to construct in some sense than the latter.
 - 4. **Active attacks for encryption:** Chosen ciphertext attacks for public key encryption.

Part IV: Advanced notions

- 1. **Fully homomorphic encryption:** Computing on encrypted data.
- 2. **Multiparty secure computation:** An amazing construction that enables applications such as playing poker over the net without trusting the server, privacy preserving data mining, electronic auctions without a trusted auctioneer, electronic elections without a trusted central authority.
- 3. **Zero knowledge proofs:** Prove a statement without revealing the reason to *why* its true.
- 4. **Quantum computing and cryptography:** Shor's algorithm to break RSA and friends. Quantum key distribution. On "quantum resistant" cryptography.
- 5. **Indistinguishability obfuscation:** Construction of indistinguishability obfuscators, the potential "master tool" for crypto.
- 6. **Practical protocols:** Techniques for constructing practical protocols for particular tasks as opposed to general (and often inefficient) feasibility proofs.
- 7. **Cryptocurrencies:** Hash chains and Merkle trees, proofs of work, achieving consensus on a ledger via "majority of cycles", smart contracts, achieving anonymity via zero knowledge proofs.

0.1.1 Prerequisites

The main prerequisite is the ability to read, write (and even enjoy!) mathematical proofs. In addition, familiarity with algorithms, basic probability theory and basic linear algebra will be helpful. We'll only use fairly basic concepts from all these areas: e.g. Oh-notation- e.g. $O(n)$ running time- from algorithms, notions such as events, random variables, expectation, from probability theory, and notions such as matrices, vectors, and eigenvectors. Mathematically mature students should be able to pick up the needed notions on their own. See the "mathematical background" handout for more details.

No programming knowledge is needed. If you're interested in the course but are not sure if you have sufficient background, or you have any other questions, please don't hesitate to contact me.

0.2 Why is cryptography hard?

Cryptography is a hard topic. Over the course of history, many brilliant people have stumbled in it, and did not realize subtle attacks on their ciphers. Even today it is frustratingly easy to get crypto wrong, and often system security is compromised because developers used crypto schemes in the wrong, or at least suboptimal, way. Why is this topic (and this course) so hard? Some of the reasons include:

- To argue about the security of a cryptographic scheme, you have to think like an attacker. This requires a very different way of thinking than what we are used to when developing algorithms or systems, and arguing that they perform well.
- To get robust assurances of security you need to argue about *all possible attacks*. The only way I know to analyze this infinite set is via *mathematical proofs*. Moreover, these types of mathematical proofs tend to be rather different than the ones most mathematicians typically work with. Because the proof itself needs to take the viewpoint of the attacker, these often tend to be proofs by contradiction and involve several twists of logic that take some getting used to.
- As we'll see in this course, even *defining* security is a highly non trivial task. Security definitions often get subtle and require quite a lot of creativity. For example, the way we model in general a statement such as "An attacker Eve does not get more information from observing a system above what she knew a-priori" is that we posit a "hypothetical alter ego" of Eve called Lilith who knows everything Eve knew a-priori but does not get to observe the actual interaction in the system. We then want to prove that anything that Eve learned could also have been learned by Lilith. If this sounds confusing, it is. But it is also fascinating, and leads to ways to argue mathematically about *knowledge* as well as beautiful generalizations of the notion of encryption and protecting communication into schemes for protecting *computation*.

If cryptography is so hard, is it really worth studying? After all, given this subtlety, a single course in cryptography is no guarantee of using (let alone inventing) crypto correctly. In my view, regardless of its immense and growing practical importance, cryptography is worth studying for its *intellectual* content. There are many areas of science where we achieve goals once considered to be science fiction. But cryptography is an area where current achievements are so fantastic that in the thousands of years of secret writing people

did not even dare *imagine* them. Moreover, cryptography may be hard because it forces you to think differently, but it is also rewarding because it teaches you to think differently. And once you pass this initial hurdle, and develop a “cryptographer’s mind”, you might find that this point of view is useful in areas that seem to have nothing to do with crypto.

Mathematical Background

This is a brief review of some mathematical tools, and especially probability theory, that we will use in this course. See also the [mathematical background](#) and [probability](#) lectures in my [Notes on Introduction to Theoretical Computer Science](#), which share much of the following text.

At Harvard, much of this material (and more) is taught in Stat 110 “Introduction to Probability”, CS20 “Discrete Mathematics”, and AM107 “Graph Theory and Combinatorics”. Some good sources for this material are the lecture notes by Papadimitriou and Vazirani (see home page of Umesh Vazirani), Lehman, Leighton and Meyer from MIT Course 6.042 “Mathematics For Computer Science” (Chapters 1-2 and 14 to 19 are particularly relevant). The mathematical tool we use most often is discrete probability. The “Probabilistic Method” book by Alon and Spencer is a great resource in this area. Also, the books of Mitzenmacher and Upfal and Prabhakar and Raghavan cover probability from a more algorithmic perspective. For an excellent popular discussion of some of the mathematical concepts we’ll talk about, I can’t recommend highly enough the book “*How Not to Be Wrong*” by Jordan Ellenberg.

Although knowledge of algorithms is not strictly necessary, it would be quite useful. Students who did not take an algorithms class such as CS 124 might want to look at the books (1) Cormen, Leiserson, Rivest and Smith, (2) Dasgupte, Papadimitriou and Vaziarni, or (3) Kleinberg and Tardos. We do not require prior knowledge of complexity or computability but some basic familiarity could be useful. Students who did not take a theory of computation class such as CS 121 might want to look at my lecture notes or the first 2 chapters of my book with Arora.

0.3 A quick overview of mathematical prerequisites

The main notions we will use in this course are the following:

- **Proofs:** First and foremost, this course will involve a heavy dose of formal mathematical reasoning, which includes mathematical *definitions, statements, and proofs*.
- **Sets and functions:** We will assume familiarity with basic notions of sets and operations on sets such as union (denoted \cup), intersection (denoted \cap), and set subtraction (denoted \setminus). We denote by $|A|$ the size of the set A . We also assume familiarity with functions, and notions such as one-to-one (injective) functions and onto (surjective) functions. If f is a function from a set A to a set B , we denote this by $f : A \rightarrow B$. If f is one-to-one then this implies that $|A| \leq |B|$. If f is onto then $|A| \geq |B|$. If f is a permutation/bijection (e.g., one-to-one *and* onto) then this implies that $|A| = |B|$.
- **Big Oh notation:** If f, g are two functions from \mathbb{N} to \mathbb{N} , then (1) $f = O(g)$ if there exists a constant c such that $f(n) \leq c \cdot g(n)$ for every sufficiently large n , (2) $f = \Omega(g)$ if $g = O(f)$, (3) $f = \Theta(g)$ if $f = O(g)$ and $g = O(f)$, (4) $f = o(g)$ if for every $\epsilon > 0$, $f(n) \leq \epsilon \cdot g(n)$ for every sufficiently large n , and (5) $f = \omega(g)$ if $g = o(f)$. To emphasize the input parameter, we often write $f(n) = O(g(n))$ instead of $f = O(g)$, and use similar notation for $o, \Omega, \omega, \Theta$. While this is only an imprecise heuristic, when you see a statement of the form $f(n) = O(g(n))$ you can often replace it in your mind by the statement $f(n) \leq 1000g(n)$ while the statement $f(n) = \Omega(g(n))$ can often be thought of as $f(n) \geq 0.001g(n)$.
- **Logical operations:** The operations AND, OR, and NOT (\wedge, \vee, \neg) and the quantifiers “exists” and “forall” (\exists, \forall).
- **Tuples and strings:** The notation Σ^k and Σ^* where Σ is some finite set which is called the *alphabet* (quite often $\Sigma = \{0, 1\}$).
- **Graphs:** Undirected and directed graphs, connectivity, paths, and cycles.
- **Basic combinatorics:** Notions such as $\binom{n}{k}$ (the number of k -sized subset of a set of size n).
- **Modular arithmetic:** We will use **modular arithmetic** (i.e., addition and multiplication modulo some number m), and in particular talk about operations on vectors and matrices whose elements are taken modulo m . If n is an integer, then we denote by $a \pmod{n}$

the remainder of a when divided by n . $a \pmod n$ is the number $r \in \{0, \dots, n-1\}$ such that $a = kn + r$ for some integer k . It will be very useful that $a \pmod n + b \pmod n = (a+b) \pmod n$ and $a \pmod n \cdot b \pmod n = (a \cdot b) \pmod n$ and so modular arithmetic inherits all of the rules (associativity, commutativity etc.) of integer arithmetic. If a, b are positive integers then $\gcd(a, b)$ is the largest integer that divides both a and b . It is known that for every a, b there exist (not necessarily positive) integers x, y such that $ax + by = \gcd(a, b)$ (it's a good exercise to prove this on your own). In particular, if $\gcd(a, n) = 1$ then there exists a *modular inverse* for a which is a number b such that $ab = 1 \pmod n$. We sometimes write b as $a^{-1} \pmod n$.

- **Group theory, linear algebra:** In later parts of the course we will need the notions of matrices, vectors, matrix multiplication and inverse, determinant, eigenvalues, and eigenvectors. These can be picked up in any basic text on linear algebra. In some parts we might also use some basic facts of group theory (finite groups only, and mostly only commutative ones). These also can be picked up as we go along, and a prior course on group theory is not necessary.
- **Discrete probability:** *Probability theory*, and specifically probability over *finite* samples spaces such as tossing n coins is a crucial part of cryptography, since (as we'll see) there is no secrecy without randomness.

0.4 Mathematical Proofs

Arguably the mathematical prerequisite needed for this course is a certain level of comfort with mathematical proofs. Many students tend to think of mathematical proofs as a very formal object, like the proofs studied in school in geometry, consisting of a sequence of axioms and statements derived from them by very specific rules. In fact,

a proof is a piece of writing meant to convince human readers that a particular statement is true.

(In this class, the particular humans you are trying to convince are me and the teaching fellows.)

To write a proof of some statement X you need to follow three steps:

1. Make sure that you completely understand the statement X.
2. Think about X until you are able to convince *yourself* that X is true.
3. Think how to present the argument in the clearest possible way so you can convince the reader as well.

Like any good piece of writing, a proof should be concise and not be overly formal or cumbersome. In fact, overuse of formalism can often be *detrimental* to the argument since it can mask weaknesses in the argument from both the writer and the reader. Sometimes students try to “throw the kitchen sink” at an answer trying to list all possibly relevant facts in the hope of getting partial credit. But a proof is a piece of writing, and a badly written proof will not get credit even if it contains some correct elements. It is better to write a clear proof of a partial statement. In particular, if you haven’t been able to convince yourself that the statement is true, you should be honest about it and explain which parts of the statement you have been able to verify and which parts you haven’t.

0.4.1 Example: The existence of infinitely many primes.

In the spirit of “do what I say and not what I do”, I will now demonstrate the importance of conciseness by belaboring the point and spending several paragraphs on a simple proof, written by Euclid around 300 BC. Recall that a *prime number* is an integer $p > 1$ whose only divisors are p and 1. Euclid’s Theorem is the following:

Theorem 0.1 — Infinitude of primes. There exist infinitely many primes.

Instead of simply writing down the proof, let us try to understand how we might figure this proof out. (If you haven’t seen this proof before, or you don’t remember it, you might want to stop reading at this point and try to come up with it on your own before continuing.) The first (and often most important) step is to understand what the statement means. Saying that the number of primes is infinite means that it is not finite. More precisely, this means that for every natural number k , there are more than k primes.

Now that we understand what we need to prove, let us try to convince ourselves of this fact. At first, it might seem obvious—since there are infinitely many natural numbers, and every one of them can be factored into primes, there must be infinitely many primes, right?

Wrong. Since we can compose a prime many times with itself, a finite number of primes can generate infinitely many numbers. Indeed the single prime 3 generates the infinite set of all numbers of the form 3^n . So, what we really need to show is that for every finite set of primes $\{p_1, \dots, p_k\}$, there exists a number n that has a prime factor outside this set.

Now we need to start playing around. Suppose that we had just two primes p and q . How would we find a number n that is not generated by p and q ? If you try to draw things on the number line, you would see that there is always some *gap* between multiples of p and q in the sense that they are never consecutive. It is possible to prove that (in fact, it's not a bad exercise) but this observation already suggests a guess for what would be a number that is divisible by neither p nor q , namely $pq + 1$. Indeed, the remainder of $n = pq + 1$ when dividing by either p or q would be 1 (which in particular is not zero). This observation generalizes and we can set $n = pqr + 1$ to be a number that is divisible neither by p, q nor r , and more generally $n = p_1 \cdots, p_k + 1$ is not divisible by p_1, \dots, p_k .

Now we have convinced ourselves of the statement and it is time to think of how to write this down in the clearest way. One issue that arises is that we want to prove things truly from the definition of primes and first principles, and so not assume properties of division and remainders or even the existence of a prime factorization, without proving it. Here is what a proof could look like. We will prove the following two lemmas:

Lemma 0.2 For every integer $n > 1$, there exists a prime $p > 1$ that divides n .

Lemma 0.3 For every set of integers $p_1, \dots, p_k > 1$, there exists a number n such that none of p_1, \dots, p_k divide n .

From these two lemmas it follows that there exist infinitely many primes, since otherwise if we let p_1, \dots, p_k be the set of all primes, then we would get a contradiction as by combining [Lemma 0.2](#) and [Lemma 0.3](#) we would get a number n with a prime factor outside this set. We now prove the lemmas:

Proof of Lemma 0.2. Let $n > 1$ be a number, and let p be the smallest divisor of n that is larger than 1 (there exists such a number p since n divides itself). We claim that p is a prime. Indeed suppose otherwise there was some $1 < q < p$ that divides p . Then since $n = pc$ for some integer c and $p = qc'$ for some integer c' we'll get that $n = qc'c$ and hence q divides n in contradiction to the choice of p as the smallest

■

divisor of n .

Proof of Lemma 0.3. Let $n = p_1 \cdots p_k + 1$ and suppose for the sake of contradiction that there exists some i such that $n = p_i \cdot c$ for some integer c . Then if we divide the equation $n - p_1 \cdots p_k = 1$ by p_i then we get c minus an integer on the lefthand side, and the fraction $1/p_i$ on the righthand side. ■

This completes the proof of [Theorem 0.1](#)

0.5 Probability and Sample spaces

Perhaps the main mathematical background needed in cryptography is probability theory since, as we will see, there is no secrecy without randomness. Luckily, we only need fairly basic notions of probability theory and in particular only probability over finite sample spaces. If you have a good understanding of what happens when we toss k random coins, then you know most of the probability you'll need. The discussion below is not meant to replace a course on probability theory, and if you have not seen this material before, I highly recommend you look at additional resources to get up to speed.¹

The nature of randomness and probability is a topic of great philosophical, scientific and mathematical depth. Is there actual randomness in the world, or does it proceed in a deterministic clock-work fashion from some initial conditions set at the beginning of time? Does probability refer to our uncertainty of beliefs, or to the frequency of occurrences in repeated experiments? How can we define probability over infinite sets?

These are all important questions that have been studied and debated by scientists, mathematicians, statisticians and philosophers. Fortunately, we will not need to deal directly with these questions here. We will be mostly interested in the setting of tossing n random, unbiased and independent coins. Below we define the basic probabilistic objects of *events* and *random variables* when restricted to this setting. These can be defined for much more general probabilistic experiments or *sample spaces*, and later on we will briefly discuss how this can be done. However, the n -coin case is sufficient for almost everything we'll need in this course.

If instead of “heads” and “tails” we encode the sides of each coin by “zero” and “one”, we can encode the result of tossing n coins as a string in $\{0, 1\}^n$. Each particular outcome $x \in \{0, 1\}^n$ is obtained

¹ Harvard’s **STAT 110** class (whose lectures are available on [youtube](#)) is a highly recommended introduction to probability. See also these [lecture notes](#) from MIT’s “Mathematics for Computer Science” course.

with probability 2^{-n} . For example, if we toss three coins, then we obtain each of the 8 outcomes 000, 001, 010, 011, 100, 101, 110, 111 with probability $2^{-3} = 1/8$ (see also Fig. 1). We can describe the experiment of tossing n coins as choosing a string x uniformly at random from $\{0,1\}^n$, and hence we'll use the shorthand $x \sim \{0,1\}^n$ for x that is chosen according to this experiment.

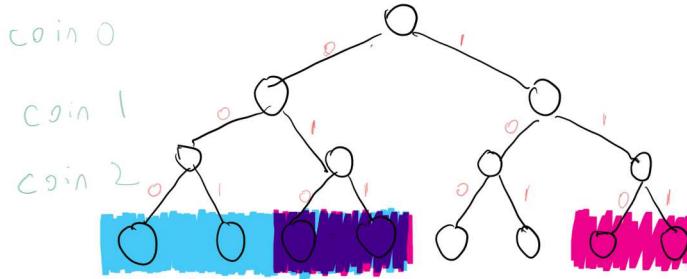


Figure 1: The probabilistic experiment of tossing three coins corresponds to making $2 \times 2 \times 2 = 8$ choices, each with equal probability. In this example, the blue set corresponds to the event $A = \{x \in \{0,1\}^3 \mid x_0 = 0\}$ where the first coin toss is equal to 0, and the pink set corresponds to the event $B = \{x \in \{0,1\}^3 \mid x_1 = 1\}$ where the second coin toss is equal to 1 (with their intersection having a purplish color). As we can see, each of these events contains 4 elements (out of 8 total) and so has probability 1/2. The intersection of A and B contains two elements, and so the probability that both of these events occur is $\frac{2}{8} = \frac{1}{4}$

An *event* is simply a subset A of $\{0,1\}^n$. The *probability* of A , denoted by $\mathbb{P}_{x \sim \{0,1\}^n}[A]$ (or $\mathbb{P}[A]$ for short, when the sample space is understood from the context), is the probability that an x chosen uniformly at random will be contained in A . Note that this is the same as $|A|/2^n$ (where $|A|$ as usual denotes the number of elements in the set A). For example, the probability that x has an even number of ones is $\mathbb{P}[A]$ where $A = \{x : \sum_{i=0}^{n-1} x_i \equiv 0 \pmod{2}\}$. In the case $n = 3$, $A = \{000, 011, 101, 110\}$, and hence $\mathbb{P}[A] = \frac{4}{8} = \frac{1}{2}$. It turns out this is true for every n :

Lemma 0.4

$$\mathbb{P}_{x \sim \{0,1\}^n} \left[\sum_{i=0}^{n-1} x_i \text{ is even} \right] = 1/2 \quad (1)$$



To test your intuition on probability, try to stop here and prove the lemma on your own.

Proof of Lemma 0.4. Let $A = \{x \in \{0,1\}^n : \sum_{i=0}^{n-1} x_i \equiv 0 \pmod{2}\}$. Since every x is obtained with probability 2^{-n} , to show this we need to show that $|A| = 2^n/2 = 2^{n-1}$. For every x_0, \dots, x_{n-2} , if $\sum_{i=0}^{n-2} x_i$ is even then $(x_0, \dots, x_{n-1}, 0) \in A$ and $(x_0, \dots, x_{n-1}, 1) \notin A$. Similarly,

if $\sum_{i=0}^{n-2} x_i$ is odd then $(x_0, \dots, x_{n-1}, 1) \in A$ and $(x_0, \dots, x_{n-1}, 0) \notin A$. Hence, for every one of the 2^{n-1} prefixes (x_0, \dots, x_{n-2}) , there is exactly a single continuation of (x_0, \dots, x_{n-2}) that places it in A . ■

We can also use the *intersection* (\cap) and *union* (\cup) operators to talk about the probability of both event A and event B happening, or the probability of event A or event B happening. For example, the probability p that x has an *even* number of ones and $x_0 = 1$ is the same as $\mathbb{P}[A \cap B]$ where $A = \{x \in \{0,1\}^n : \sum_{i=0}^{n-1} x_i = 0 \pmod{2}\}$ and $B = \{x \in \{0,1\}^n : x_0 = 1\}$. This probability is equal to $1/4$. (It is a great exercise for you to pause here and verify that you understand why this is the case.)

Because intersection corresponds to considering the logical AND of the conditions that two events happen, while union corresponds to considering the logical OR, we will sometimes use the \wedge and \vee operators instead of \cap and \cup , and so write this probability $p = \mathbb{P}[A \cap B]$ defined above also as

$$\mathbb{P}_{x \sim \{0,1\}^n} \left[\sum_i x_i = 0 \pmod{2} \wedge x_0 = 1 \right]. \quad (2)$$

If $A \subseteq \{0,1\}^n$ is an event, then $\bar{A} = \{0,1\}^n \setminus A$ corresponds to the event that A does *not* happen. Since $|\bar{A}| = 2^n - |A|$, we get that

$$\mathbb{P}[\bar{A}] = \frac{|\bar{A}|}{2^n} = \frac{2^n - |A|}{2^n} = 1 - \frac{|A|}{2^n} = 1 - \mathbb{P}[A] \quad (3)$$

This makes sense: since A happens if and only if \bar{A} does *not* happen, the probability of \bar{A} should be one minus the probability of A .

R **Remember the sample space** While the above definition might seem very simple and almost trivial, the human mind seems not to have evolved for probabilistic reasoning, and it is surprising how often people can get even the simplest settings of probability wrong. One way to make sure you don't get confused when trying to calculate probability statements is to always ask yourself the following two questions: (1) Do I understand what is the **sample space** that this probability is taken over?, and (2) Do I understand what is the definition of the **event** that we are analyzing?.

For example, suppose that I were to randomize seating in my course, and then it turned out that students sitting in row 7 performed better on the final: how surprising should we find this? If we started out with the hypothesis that there is something special about the number 7 and chose it ahead of time, then the event that we are discussing is the

event A that students sitting in number 7 had better performance on the final, and we might find it surprising. However, if we first looked at the results and then chose the row whose average performance is best, then the event we are discussing is the event B that there exists *some* row where the performance is higher than the overall average. B is a superset of A , and its probability (even if there is no correlation between sitting and performance) can be quite significant.

0.5.1 Random variables

Events correspond to Yes/No questions, but often we want to analyze finer questions. For example, if we make a bet at the roulette wheel, we don't want to just analyze whether we won or lost, but also *how much* we've gained. A (real valued) *random variable* is simply a way to associate a number with the result of a probabilistic experiment. Formally, a random variable is simply a function $X : \{0,1\}^n \rightarrow \mathbb{R}$ that maps every outcome $x \in \{0,1\}^n$ to a real number $X(x)$.² For example, the function $\text{sum} : \{0,1\}^n \rightarrow \mathbb{R}$ that maps x to the sum of its coordinates (i.e., to $\sum_{i=0}^{n-1} x_i$) is a random variable.

The *expectation* of a random variable X , denoted by $\mathbb{E}[X]$, is the average value that that this number takes, taken over all draws from the probabilistic experiment. In other words, the expectation of X is defined as follows:

$$\mathbb{E}[X] = \sum_{x \in \{0,1\}^n} 2^{-n} X(x). \quad (4)$$

If X and Y are random variables, then we can define $X + Y$ as simply the random variable that maps a point $x \in \{0,1\}^n$ to $X(x) + Y(x)$. One basic and very useful property of the expectation is that it is *linear*:

Lemma 0.5 — Linearity of expectation.

$$\mathbb{E}[X + Y] = \mathbb{E}[X] + \mathbb{E}[Y] \quad (5)$$

Proof.

$$\begin{aligned} \mathbb{E}[X + Y] &= \sum_{x \in \{0,1\}^n} 2^{-n} (X(x) + Y(x)) = \\ &\sum_{x \in \{0,1\}^n} 2^{-n} X(x) + \sum_{x \in \{0,1\}^n} 2^{-n} Y(x) = \\ &\mathbb{E}[X] + \mathbb{E}[Y] \end{aligned} \quad (6)$$

² In many probability texts a random variable is always defined to have values in the set \mathbb{R} of real numbers, and this will be our default option as well. However, in some contexts in theoretical computer science we can consider random variables mapping to other sets such as $\{0,1\}^*$.

■

Similarly, $\mathbb{E}[kX] = k\mathbb{E}[X]$ for every $k \in \mathbb{R}$. For example, using the linearity of expectation, it is very easy to show that the expectation of the sum of the x_i 's for $x \sim \{0,1\}^n$ is equal to $n/2$. Indeed, if we write $X = \sum_{i=0}^{n-1} x_i$ then $X = X_0 + \dots + X_{n-1}$ where X_i is the random variable x_i . Since for every i , $\mathbb{P}[X_i = 0] = 1/2$ and $\mathbb{P}[X_i = 1] = 1/2$, we get that $\mathbb{E}[X_i] = (1/2) \cdot 0 + (1/2) \cdot 1 = 1/2$ and hence $\mathbb{E}[X] = \sum_{i=0}^{n-1} \mathbb{E}[X_i] = n \cdot (1/2) = n/2$.

P If you have not seen discrete probability before, please go over this argument again until you are sure you follow it; it is a prototypical simple example of the type of reasoning we will employ again and again in this course.

If A is an event, then 1_A is the random variable such that $1_A(x)$ equals 1 if $x \in A$, and $1_A(x) = 0$ otherwise. Note that $\mathbb{P}[A] = \mathbb{E}[1_A]$ (can you see why?). Using this and the linearity of expectation, we can show one of the most useful bounds in probability theory:

Lemma 0.6 — Union bound. For every two events A, B , $\mathbb{P}[A \cup B] \leq \mathbb{P}[A] + \mathbb{P}[B]$

P Before looking at the proof, try to see why the union bound makes intuitive sense. We can also prove it directly from the definition of probabilities and the cardinality of sets, together with the equation $|A \cup B| \leq |A| + |B|$. Can you see why the latter equation is true? (See also Fig. 2.)

Proof of Lemma 0.6. For every x , the variable $1_{A \cup B}(x) \leq 1_A(x) + 1_B(x)$. Hence, $\mathbb{P}[A \cup B] = \mathbb{E}[1_{A \cup B}] \leq \mathbb{E}[1_A + 1_B] = \mathbb{E}[1_A] + \mathbb{E}[1_B] = \mathbb{P}[A] + \mathbb{P}[B]$. ■

The way we often use this in theoretical computer science is to argue that, for example, if there is a list of 100 bad events that can happen, and each one of them happens with probability at most $1/10000$, then with probability at least $1 - 100/10000 = 0.99$, no bad event happens.

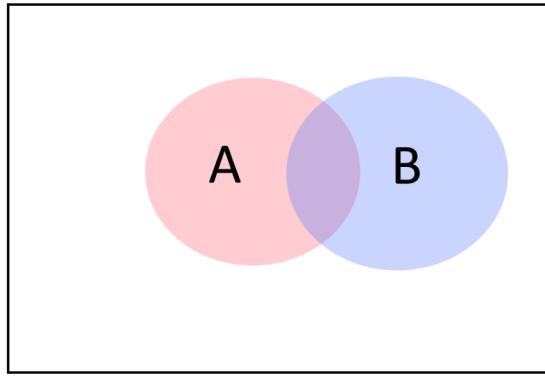


Figure 2: The *union bound* tells us that the probability of A or B happening is at most the sum of the individual probabilities. We can see it by noting that for every two sets $|A \cup B| \leq |A| + |B|$ (with equality only if A and B have no intersection).

0.5.2 Distributions over strings

While most of the time we think of random variables as having as output a *real number*, we sometimes consider random variables whose output is a *string*. That is, we can think of a map $Y : \{0,1\}^n \rightarrow \{0,1\}^*$ and consider the “random variable” Y such that for every $y \in \{0,1\}^*$, the probability that Y outputs y is equal to $\frac{1}{2^n} |\{x \in \{0,1\}^n \mid Y(x) = y\}|$. To avoid confusion, we will typically refer to such string-valued random variables as *distributions* over strings. So, a *distribution* Y over strings $\{0,1\}^*$ can be thought of as a finite collection of strings $y_0, \dots, y_{M-1} \in \{0,1\}^*$ and probabilities p_0, \dots, p_{M-1} (which are non-negative numbers summing up to one), so that $\mathbb{P}[Y = y_i] = p_i$.

Two distributions Y and Y' are *identical* if they assign the same probability to every string. For example, consider the following two functions $Y, Y' : \{0,1\}^2 \rightarrow \{0,1\}^2$. For every $x \in \{0,1\}^2$, we define $Y(x) = x$ and $Y'(x) = x_0(x_0 \oplus x_1)$ where \oplus is the XOR operation. Although these are two different functions, they induce the same distribution over $\{0,1\}^2$ when invoked on a uniform input. The distribution $Y(x)$ for $x \sim \{0,1\}^2$ is of course the uniform distribution over $\{0,1\}^2$. On the other hand Y' is simply the map $00 \mapsto 00, 01 \mapsto 01, 10 \mapsto 11, 11 \mapsto 10$ which is a permutation over the map $F : \{0,1\}^2 \rightarrow \{0,1\}^2$ defined as $F(x_0x_1) = x_0x_1$ and the map $G : \{0,1\}^2 \rightarrow \{0,1\}^2$ defined as $G(x_0x_1) = x_0(x_0 \oplus x_1)$

0.5.3 More general sample spaces.

While in this lecture we assume that the underlying probabilistic experiment corresponds to tossing n independent coins, everything we say easily generalizes to sampling x from a more general finite or countable set S (and not-so-easily generalizes to uncountable sets S as well). A *probability distribution* over a finite set S is simply a function $\mu : S \rightarrow [0, 1]$ such that $\sum_{x \in S} \mu(x) = 1$. We think of this as the experiment where we obtain every $x \in S$ with probability $\mu(x)$, and sometimes denote this as $x \sim \mu$. An *event* A is a subset of S , and the probability of A , which we denote by $\mathbb{P}_\mu[A]$, is $\sum_{x \in A} \mu(x)$. A *random variable* is a function $X : S \rightarrow \mathbb{R}$, where the probability that $X = y$ is equal to $\sum_{x \in S \text{ s.t. } X(x)=y} \mu(x)$.

3

³ TODO: add exercise on simulating die tosses and choosing a random number in $[m]$ by coin tosses

0.6 Correlations and independence

One of the most delicate but important concepts in probability is the notion of *independence* (and the opposing notion of *correlations*). Subtle correlations are often behind surprises and errors in probability and statistical analysis, and several mistaken predictions have been blamed on miscalculating the correlations between, say, housing prices in Florida and Arizona, or voter preferences in Ohio and Michigan. See also Joe Blitzstein's aptly named talk "**Conditioning is the Soul of Statistics**".⁴

Two events A and B are *independent* if the fact that A happens makes B neither more nor less likely to happen. For example, if we think of the experiment of tossing 3 random coins $x \in \{0, 1\}^3$, and we let A be the event that $x_0 = 1$ and B the event that $x_0 + x_1 + x_2 \geq 2$, then if A happens it is more likely that B happens, and hence these events are *not* independent. On the other hand, if we let C be the event that $x_1 = 1$, then because the second coin toss is not affected by the result of the first one, the events A and C are independent.

The formal definition is that events A and B are *independent* if $\mathbb{P}[A \cap B] = \mathbb{P}[A] \cdot \mathbb{P}[B]$. If $\mathbb{P}[A \cap B] > \mathbb{P}[A] \cdot \mathbb{P}[B]$ then we say that A and B are *positively correlated*, while if $\mathbb{P}[A \cap B] < \mathbb{P}[A] \cdot \mathbb{P}[B]$ then we say that A and B are *negatively correlated* (see Fig. 1).

If we consider the above examples on the experiment of choosing

⁴ Another thorny issue is of course the difference between *correlation* and *causation*. Luckily, this is another point we don't need to worry about in our clean setting of tossing n coins.

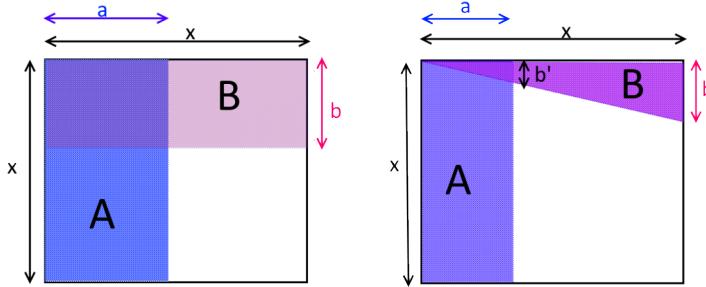


Figure 3: Two events A and B are *independent* if $\mathbb{P}[A \cap B] = \mathbb{P}[A] \cdot \mathbb{P}[B]$. In the two figures above, the empty $x \times x$ square is the sample space, and A and B are two events in this sample space. In the left figure, A and B are independent, while in the right figure they are negatively correlated, since B is less likely to occur if we condition on A (and vice versa). Mathematically, one can see this by noticing that in the left figure the areas of A and B respectively are $a \cdot x$ and $b \cdot x$, and so their probabilities are $\frac{a \cdot x}{x^2} = \frac{a}{x}$ and $\frac{b \cdot x}{x^2} = \frac{b}{x}$ respectively, while the area of $A \cap B$ is $a \cdot b$ which corresponds to the probability $\frac{a \cdot b}{x^2}$. In the right figure, the area of the triangle B is $\frac{b \cdot x}{2}$ which corresponds to a probability of $\frac{b}{2x}$, but the area of $A \cap B$ is $\frac{b' \cdot a}{2}$ for some $b' < b$. This means that the probability of $A \cap B$ is $\frac{b' \cdot a}{2x^2} < \frac{b}{2x} \cdot \frac{a}{x}$, or in other words $\mathbb{P}[A \cap B] < \mathbb{P}[A] \cdot \mathbb{P}[B]$.

$x \in \{0, 1\}^3$ then we can see that

$$\begin{aligned}\mathbb{P}[x_0 = 1] &= \frac{1}{2} \\ \mathbb{P}[x_0 + x_1 + x_2 \geq 2] &= \mathbb{P}[\{011, 101, 110, 111\}] = \frac{4}{8} = \frac{1}{2}\end{aligned}\quad (7)$$

but

$$\mathbb{P}[x_0 = 1 \wedge x_0 + x_1 + x_2 \geq 2] = \mathbb{P}[\{101, 110, 111\}] = \frac{3}{8} > \frac{1}{2} \cdot \frac{1}{2} \quad (8)$$

and hence, as we already observed, the events $\{x_0 = 1\}$ and $\{x_0 + x_1 + x_2 \geq 2\}$ are not independent and in fact are positively correlated. On the other hand, $\mathbb{P}[x_0 = 1 \wedge x_1 = 1] = \mathbb{P}[\{110, 111\}] = \frac{2}{8} = \frac{1}{2} \cdot \frac{1}{2}$ and hence the events $\{x_0 = 1\}$ and $\{x_1 = 1\}$ are indeed independent.



Disjointness vs independence People sometimes confuse the notion of *disjointness* and *independence*, but these are actually quite different. Two events A and B are *disjoint* if $A \cap B = \emptyset$, which means that if A happens then B definitely does not happen. They are *independent* if $\mathbb{P}[A \cap B] = \mathbb{P}[A] \mathbb{P}[B]$ which means that knowing that A happens gives us no information about whether B happened or not. If A and B have nonzero probability, then being disjoint implies

that they are *not* independent, since in particular it means that they are negatively correlated.

Conditional probability: If A and B are events, and A happens with nonzero probability then we define the probability that B happens *conditioned on* A to be $\mathbb{P}[B|A] = \mathbb{P}[A \cap B] / \mathbb{P}[A]$. This corresponds to calculating the probability that B happens if we already know that A happened. Note that A and B are independent if and only if $\mathbb{P}[B|A] = \mathbb{P}[B]$.

More than two events: We can generalize this definition to more than two events. We say that events A_1, \dots, A_k are *mutually independent* if knowing that any set of them occurred or didn't occur does not change the probability that an event outside the set occurs. Formally, the condition is that for every subset $I \subseteq [k]$,

$$\mathbb{P}[\bigwedge_{i \in I} A_i] = \prod_{i \in I} \mathbb{P}[A_i]. \quad (9)$$

For example, if $x \sim \{0,1\}^3$, then the events $\{x_0 = 1\}$, $\{x_1 = 1\}$ and $\{x_2 = 1\}$ are mutually independent. On the other hand, the events $\{x_0 = 1\}$, $\{x_1 = 1\}$ and $\{x_0 + x_1 = 0 \pmod 2\}$ are *not* mutually independent, even though every pair of these events is independent (can you see why? see also Fig. 4).

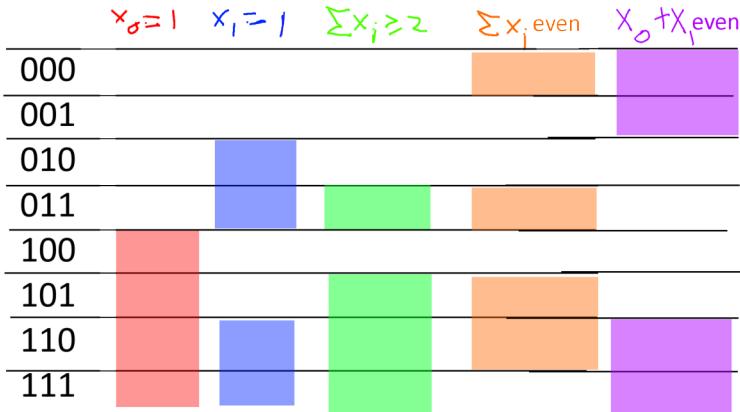


Figure 4: Consider the sample space $\{0,1\}^n$ and the events A, B, C, D, E corresponding to $A: x_0 = 1$, $B: x_1 = 1$, $C: x_0 + x_1 + x_2 \geq 2$, $D: x_0 + x_1 + x_2 = 0 \pmod 2$ and $E: x_0 + x_1 = 0 \pmod 2$. We can see that A and B are independent, C is positively correlated with A and positively correlated with B , the three events A, B, D are mutually independent, and while every pair out of A, B, E is independent, the three events A, B, E are not mutually independent since their intersection has probability $\frac{2}{8} = \frac{1}{4}$ instead of $\frac{1}{2} \cdot \frac{1}{2} \cdot \frac{1}{2} = \frac{1}{8}$.

o.6.1 Independent random variables

We say that two random variables $X : \{0, 1\}^n \rightarrow \mathbb{R}$ and $Y : \{0, 1\}^n \rightarrow \mathbb{R}$ are independent if for every $u, v \in \mathbb{R}$, the events $\{X = u\}$ and $\{Y = v\}$ are independent.⁵ In other words, X and Y are independent if $\mathbb{P}[X = u \wedge Y = v] = \mathbb{P}[X = u]\mathbb{P}[Y = v]$ for every $u, v \in \mathbb{R}$. For example, if two random variables depend on the result of tossing different coins then they are independent:

Lemma 0.7 Suppose that $S = \{s_0, \dots, s_{k-1}\}$ and $T = \{t_0, \dots, t_{m-1}\}$ are disjoint subsets of $\{0, \dots, n-1\}$ and let $X, Y : \{0, 1\}^n \rightarrow \mathbb{R}$ be random variables such that $X = F(x_{s_0}, \dots, x_{s_{k-1}})$ and $Y = G(x_{t_0}, \dots, x_{t_{m-1}})$ for some functions $F : \{0, 1\}^k \rightarrow \mathbb{R}$ and $G : \{0, 1\}^m \rightarrow \mathbb{R}$. Then X and Y are independent.

⁵ We use $\{X = u\}$ as shorthand for $\{x \mid X(x) = u\}$.



The notation in the lemma's statement is a bit cumbersome, but at the end of the day, it simply says that if X and Y are random variables that depend on two disjoint sets S and T of coins (for example, X might be the sum of the first $n/2$ coins, and Y might be the largest consecutive stretch of zeroes in the second $n/2$ coins), then they are independent.

Proof of Lemma 0.7. Let $a, b \in \mathbb{R}$, and let $A = \{x \in \{0, 1\}^k : F(x) = a\}$ and $B = \{x \in \{0, 1\}^m : G(x) = b\}$. Since S and T are disjoint, we can reorder the indices so that $S = \{0, \dots, k-1\}$ and $T = \{k, \dots, k+m-1\}$ without affecting any of the probabilities. Hence we can write $\mathbb{P}[X = a \wedge Y = b] = |C|/2^n$ where $C = \{x_0, \dots, x_{n-1} : (x_0, \dots, x_{k-1}) \in A \wedge (x_k, \dots, x_{k+m-1}) \in B\}$. Another way to write this using string concatenation is that $C = \{xyz : x \in A, y \in B, z \in \{0, 1\}^{n-k-m}\}$, and hence $|C| = |A||B|2^{n-k-m}$, which means that

$$\frac{|C|}{2^n} = \frac{|A|}{2^k} \frac{|B|}{2^m} \frac{2^{n-k-m}}{2^{n-k-m}} = \mathbb{P}[X = a]\mathbb{P}[Y = b]. \quad (10)$$

■

Note that if X and Y are independent random variables then (if we let S_X, S_Y denote all the numbers that have positive probability of being the output of X and Y , respectively) it holds that:

$$\begin{aligned} \mathbb{E}[XY] &= \sum_{a \in S_X, b \in S_Y} \mathbb{P}[X = a \wedge Y = b] \cdot ab =^{(1)} \sum_{a \in S_X, b \in S_Y} \mathbb{P}[X = a]\mathbb{P}[Y = b] \cdot ab =^{(2)} \\ &\quad \left(\sum_{a \in S_X} \mathbb{P}[X = a] \cdot a \right) \left(\sum_{b \in S_Y} \mathbb{P}[Y = b] \cdot b \right) =^{(3)} \\ &\quad \mathbb{E}[X]\mathbb{E}[Y] \end{aligned} \quad (11)$$

where the first equality ($=^{(1)}$) follows from the independence of X and Y , the second equality ($=^{(2)}$) follows by “opening the parentheses” of the righthand side, and the third inequality ($=^{(3)}$) follows from the definition of expectation. (This is not an “if and only if”; see ??.)

Another useful fact is that if X and Y are independent random variables, then so are $F(X)$ and $G(Y)$ for all functions $F, G : \mathbb{R} \rightarrow \mathbb{R}$. This is intuitively true since learning $F(X)$ can only provide us with less information than does learning X itself. Hence, if learning X does not teach us anything about Y (and so also about $F(Y)$) then neither will learning $F(X)$. Indeed, to prove this we can write for every $a, b \in \mathbb{R}$:

$$\begin{aligned} \mathbb{P}[F(X) = a \wedge G(Y) = b] &= \sum_{x \text{ s.t. } F(x)=a, y \text{ s.t. } G(y)=b} \mathbb{P}[X = x \wedge Y = y] = \\ &\quad \sum_{x \text{ s.t. } F(x)=a, y \text{ s.t. } G(y)=b} \mathbb{P}[X = x] \mathbb{P}[Y = y] = \\ &\quad \left(\sum_{x \text{ s.t. } F(x)=a} \mathbb{P}[X = x] \right) \cdot \left(\sum_{y \text{ s.t. } G(y)=b} \mathbb{P}[Y = y] \right) = \\ &\quad \mathbb{P}[F(X) = a] \mathbb{P}[G(Y) = b]. \end{aligned} \tag{12}$$

o.6.2 Collections of independent random variables.

We can extend the notions of independence to more than two random variables: we say that the random variables X_0, \dots, X_{n-1} are *mutually independent* if for every $a_0, \dots, a_{n-1} \in \mathbb{E}$,

$$\mathbb{P}[X_0 = a_0 \wedge \dots \wedge X_{n-1} = a_{n-1}] = \mathbb{P}[X_0 = a_0] \cdots \mathbb{P}[X_{n-1} = a_{n-1}]. \tag{13}$$

And similarly, we have that

Lemma 0.8 — Expectation of product of independent random variables. If X_0, \dots, X_{n-1} are mutually independent then

$$\mathbb{E}\left[\prod_{i=0}^{n-1} X_i\right] = \prod_{i=0}^{n-1} \mathbb{E}[X_i]. \tag{14}$$

Lemma 0.9 — Functions preserve independence. If X_0, \dots, X_{n-1} are mutually independent, and Y_0, \dots, Y_{n-1} are defined as $Y_i = F_i(X_i)$ for some functions $F_0, \dots, F_{n-1} : \mathbb{R} \rightarrow \mathbb{R}$, then Y_0, \dots, Y_{n-1} are mutually independent as well.

P We leave proving [Lemma 0.8](#) and [Lemma 0.9](#) as ???. It is good idea for you stop now and do these exercises to make sure you are comfortable with the notion of independence, as we will use it heavily later on in this course.

0.7 Concentration

The name “expectation” is somewhat misleading. For example, suppose that you and I place a bet on the outcome of 10 coin tosses, where if they all come out to be 1’s then I pay you 100,000 dollars and otherwise you pay me 10 dollars. If we let $X : \{0,1\}^{10} \rightarrow \mathbb{R}$ be the random variable denoting your gain, then we see that

$$\mathbb{E}[X] = 2^{-10} \cdot 100000 - (1 - 2^{-10})10 \sim 90. \quad (15)$$

But we don’t really “expect” the result of this experiment to be for you to gain 90 dollars. Rather, 99.9 percent of the time you will pay me 10 dollars, and you will hit the jackpot 0.01 percent of the times.

However, if we repeat this experiment again and again (with fresh and hence *independent* coins), then in the long run we do expect your average earning to be 90 dollars, which is the reason why casinos can make money in a predictable way even though every individual bet is random. For example, if we toss n coins, then as n grows, the number of coins that come up ones will be more and more concentrated around $n/2$ according to the famous “bell curve” (see [Fig. 5](#)).

Much of probability theory is concerned with so called *concentration* or *tail* bounds, which are upper bounds on the probability that a random variable X deviates too much from its expectation. The first and simplest one of them is Markov’s inequality:

Theorem 0.10 — Markov’s inequality. If X is a non-negative random variable then $\mathbb{P}[X \geq k \mathbb{E}[X]] \leq 1/k$.

P Markov’s Inequality is actually a very natural statement (see also [Fig. 6](#)). For example, if you know that the average (not the median!) household income in the US is 70,000 dollars, then in particular you can deduce that at most 25 percent of households make

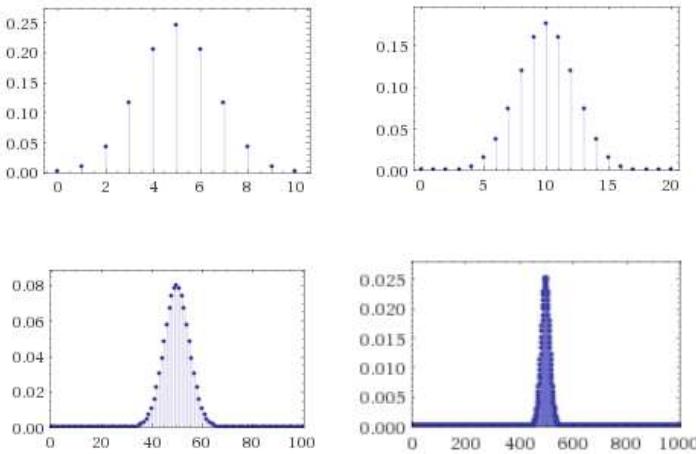


Figure 5: The probabilities that we obtain a particular sum when we toss $n = 10, 20, 100, 1000$ coins converge quickly to the Gaussian/normal distribution.

more than 280,000 dollars, since otherwise, even if the remaining 75 percent had zero income, the top 25 percent alone would cause the average income to be larger than 70,000. From this example you can already see that in many situations, Markov's inequality will not be *tight* and the probability of deviating from expectation will be much smaller: see the Chebyshev and Chernoff inequalities below.

Proof of Theorem 0.10. Let $\mu = \mathbb{E}[X]$ and define $Y = 1_{X \geq k\mu}$. That is, $Y(x) = 1$ if $X(x) \geq k\mu$ and $Y(x) = 0$ otherwise. Note that by definition, for every x , $Y(x) \leq X/(k\mu)$. We need to show $\mathbb{E}[Y] \leq 1/k$. But this follows since $\mathbb{E}[Y] \leq \mathbb{E}[X/k(\mu)] = \mathbb{E}[X]/(k\mu) = \mu/(k\mu) = 1/k$. ■

Going beyond Markov's Inequality: Markov's inequality says that a (non-negative) random variable X can't go too crazy and be, say, a million times its expectation, with significant probability. But ideally we would like to say that with high probability, X should be very close to its expectation, e.g., in the range $[0.99\mu, 1.01\mu]$ where $\mu = \mathbb{E}[X]$. This is not generally true, but does turn out to hold when X is obtained by combining (e.g., adding) many independent random variables. This phenomenon, variants of which are known as "law of large numbers", "central limit theorem", "invariance principles" and "Chernoff bounds", is one of the most fundamental in probability and statistics, and is one that we heavily use in computer science as

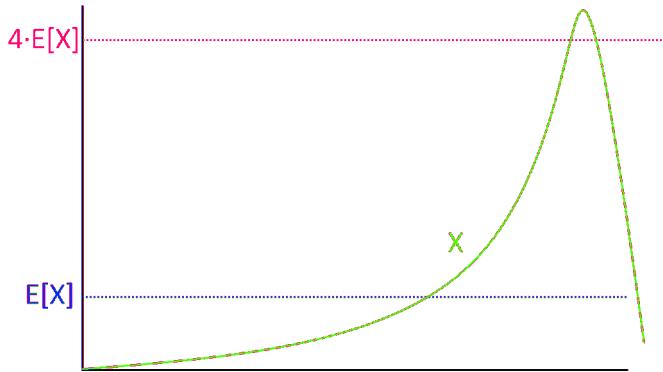


Figure 6: Markov's Inequality tells us that a non-negative random variable X cannot be much larger than its expectation, with high probability. For example, if the expectation of X is μ , then the probability that $X > 4\mu$ must be at most $1/4$, as otherwise just the contribution from this part of the sample space will be too large.

well.

0.7.1 Chebyshev's Inequality

A standard way to measure the deviation of a random variable from its expectation is by using its *standard deviation*. For a random variable X , we define the *variance* of X as $\text{Var}[X] = \mathbb{E}[(X - \mu)^2]$ where $\mu = \mathbb{E}[X]$; i.e., the variance is the average squared distance of X from its expectation. The *standard deviation* of X is defined as $\sigma[X] = \sqrt{\text{Var}[X]}$. (This is well-defined since the variance, being an average of a square, is always a non-negative number.)

Using Chebyshev's inequality, we can control the probability that a random variable is too many standard deviations away from its expectation.

Theorem 0.11 — Chebyshev's inequality. Suppose that $\mu = \mathbb{E}[X]$ and $\sigma^2 = \text{Var}[X]$. Then for every $k > 0$, $\mathbb{P}[|X - \mu| \geq k\sigma] \leq 1/k^2$.

Proof. The proof follows from Markov's inequality. We define the random variable $Y = (X - \mu)^2$. Then $\mathbb{E}[Y] = \text{Var}[X] = \sigma^2$, and hence by Markov the probability that $Y > k^2\sigma^2$ is at most $1/k^2$. But clearly $(X - \mu)^2 \geq k^2\sigma^2$ if and only if $|X - \mu| \geq k\sigma$. ■

One example of how to use Chebyshev's inequality is the setting when $X = X_1 + \dots + X_n$ where X_i 's are *independent and identically distributed* (i.i.d for short) variables with values in $[0, 1]$ where each has expectation $1/2$. Since $\mathbb{E}[X] = \sum_i \mathbb{E}[X_i] = n/2$, we would like to say that X is very likely to be in, say, the interval $[0.499n, 0.501n]$. Using Markov's inequality directly will not help us, since it will only tell us that X is very likely to be at most $100n$ (which we already knew, since it always lies between 0 and n). However, since X_1, \dots, X_n are independent,

$$\text{Var}[X_1 + \dots + X_n] = \text{Var}[X_1] + \dots + \text{Var}[X_n]. \quad (16)$$

(We leave showing this to the reader as ??.)

For every random variable X_i in $[0, 1]$, $\text{Var}[X_i] \leq 1$ (if the variable is always in $[0, 1]$, it can't be more than 1 away from its expectation), and hence Eq. (16) implies that $\text{Var}[X] \leq n$ and hence $\sigma[X] \leq \sqrt{n}$. For large n , $\sqrt{n} \ll 0.001n$, and in particular if $\sqrt{n} \leq 0.001n/k$, we can use Chebyshev's inequality to bound the probability that X is not in $[0.499n, 0.501n]$ by $1/k^2$.

0.7.2 The Chernoff bound

Chebyshev's inequality already shows a connection between independence and concentration, but in many cases we can hope for a quantitatively much stronger result. If, as in the example above, $X = X_1 + \dots + X_n$ where the X_i 's are bounded i.i.d random variables of mean $1/2$, then as n grows, the distribution of X would be roughly the *normal* or *Gaussian* distribution— that is, distributed according to the *bell curve* (see Fig. 5 and Fig. 7). This distribution has the property of being *very* concentrated in the sense that the probability of deviating k standard deviations from the mean is not merely $1/k^2$ as is guaranteed by Chebyshev, but rather is roughly e^{-k^2} .⁶ That is, we have an *exponential decay* of the probability of deviation.

The following extremely useful theorem shows that such exponential decay occurs every time we have a sum of independent and bounded variables. This theorem is known under many names in different communities, though it is mostly called the **Chernoff bound** in the computer science literature:

Theorem 0.12 — Chernoff/Hoeffding bound. If X_1, \dots, X_n are i.i.d random variables such that $X_i \in [0, 1]$ and $\mathbb{E}[X_i] = p$ for every i , then

⁶ Specifically, for a normal random variable X of expectation μ and standard deviation σ , the probability that $|X - \mu| \geq k\sigma$ is at most $2e^{-k^2/2}$.

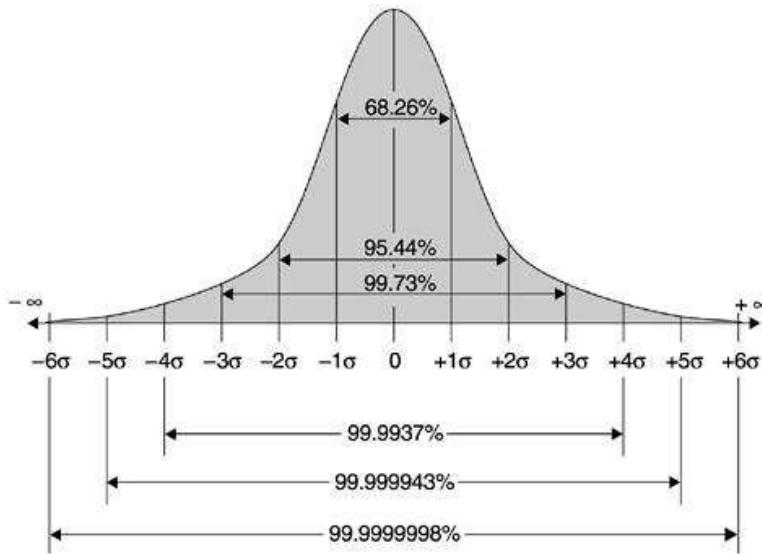


Figure 7: In the *normal distribution* or the Bell curve, the probability of deviating k standard deviations from the expectation shrinks exponentially in k^2 , and specifically with probability at least $1 - 2e^{-k^2/2}$, a random variable X of expectation μ and standard deviation σ satisfies $\mu - k\sigma \leq X \leq \mu + k\sigma$. This figure gives more precise bounds for $k = 1, 2, 3, 4, 5, 6$. (Image credit:Imran Baghirov)

for every $\epsilon > 0$

$$\mathbb{P}\left[\left|\sum_{i=0}^{n-1} X_i - pn\right| > \epsilon n\right] \leq 2 \cdot e^{-2\epsilon^2 n}. \quad (17)$$

We omit the proof, which appears in many texts, and uses Markov's inequality on i.i.d random variables Y_0, \dots, Y_n that are of the form $Y_i = e^{\lambda X_i}$ for some carefully chosen parameter λ . See ?? for a proof of the simple (but highly useful and representative) case where each X_i is $\{0, 1\}$ valued and $p = 1/2$. (See also ?? for a generalization.)

o.8 Exercises

The following exercises will be part of the first problem set in the course, so you can get a head start by working on them now.

1. In the following exercise X, Y denote random variables over some sample space S . You can assume that the probability on S is the uniform distribution—every point s is output with probability $1/|S|$. Thus $\mathbb{E}[X] = (1/|S|) \sum_{s \in S} X(s)$. We define the variance and standard deviation of X and Y as above (e.g., $\text{Var}[X] = \mathbb{E}[(X - \mathbb{E}[X])^2]$ and the standard deviation is the square

root of the variance).

- (a) Prove that $\text{Var}[X]$ is always non-negative.
- (b) Prove that $\text{Var}[X] = \mathbb{E}[X^2] - \mathbb{E}[X]^2$.
- (c) Prove that always $\mathbb{E}[X^2] \geq \mathbb{E}[X]^2$.
- (d) Give an example for a random variable X such that $\mathbb{E}[X^2] \neq \mathbb{E}[X]^2$.
- (e) Give an example for a random variable X such that its standard deviation is *not equal* to $\mathbb{E}[|X - \mathbb{E}[X]|]$.
- (f) Give an example for two random variables X, Y such that $\mathbb{E}[XY] = \mathbb{E}[X]\mathbb{E}[Y]$.
- (g) Give an example for two random variables X, Y such that $\mathbb{E}[XY] \neq \mathbb{E}[X]\mathbb{E}[Y]$.
- (h) Prove that if X and Y are independent random variables (i.e., for every x, y , $\mathbb{P}[X = x \wedge Y = y] = \mathbb{P}[X = x]\mathbb{P}[Y = y]$) then $\mathbb{E}[XY] = \mathbb{E}[X]\mathbb{E}[Y]$ and $\text{Var}[X + Y] = \text{Var}[X] + \text{Var}[Y]$.

Suppose that H is chosen to be a random function mapping the numbers $\{1, \dots, n\}$ to the numbers $\{1, \dots, m\}$. That is, for every $i \in \{1, \dots, n\}$, $H(i)$ is chosen to be a random number in $\{1, \dots, m\}$ and that choice is done independently for every i . For every $i < j \in \{1, \dots, n\}$, define the random variable $X_{i,j}$ to equal 1 if there was a *collision* between $H(i)$ and $H(j)$ in the sense that $H(i) = H(j)$ and to equal 0 otherwise.

2. (a) For every $i < j$, compute $\mathbb{E}[X_{i,j}]$.
- (b) Define $Y = \sum_{i < j} X_{i,j}$ to be the total number of collisions. Compute $\mathbb{E}[Y]$ as a function of n and m . In particular your answer should imply that if $m < n^2/1000$ then $\mathbb{E}[Y] > 1$ and hence in expectation there should be at least one collision and so the function H will not be one to one.
- (c) Prove that if $m > 1000 \cdot n^2$ then the probability that H is one to one is at least 0.9.
- (d) Give an example of a random variable Z (unrelated to the function H) that is always equal to a non-negative integer, and such that $\mathbb{E}[Z] \geq 1000$ but $\mathbb{P}[Z > 0] < 0.001$.
- (e) Prove that if $m < n^2/1000$ then the probability that H is one to one is at most 0.1.

3. In this exercise we will work out an important special case of the Chernoff bound. You can take as a given the following facts:

- (a) The number of $x \in \{0, 1\}^n$ such that $\sum x_i = k$ is $\binom{n}{k} = \frac{n!}{k!(n-k)!}$.
- (b) Stirling's approximation formula: for every $n \geq 1$,

$$\sqrt{2\pi n} \left(\frac{n}{e}\right)^n \leq n! \leq 2\sqrt{2\pi n} \left(\frac{n}{e}\right)^n \quad (18)$$

where $e = 2.7182\dots$ is the base of the natural logarithm.

Do the following:

- (a) Prove that for every n , $\mathbb{P}_{x \leftarrow_R \{0,1\}^n} [\sum x_i \geq 0.6n] < 2^{-n/1000}$

The above shows that if you were given a coin of bias at least 0.6, you should only need some constant number of samples to be able to reject the "null hypothesis" that the coin is completely unbiased with extremely high confidence. In the following somewhat more challenging questions (which can be considered as bonus exercise) we try to show a converse to this:

- (a) Let P be the uniform distribution over $\{0, 1\}^n$ and Q be the $1/2 + \epsilon$ -biased distribution corresponding to tossing n coins in which each one has a probability of $1/2 + \epsilon$ of equalling 1 and probability $1/2 - \epsilon$ of equalling 0. Namely the probability of $x \in \{0, 1\}^n$ according to Q is equal to $\prod_{i=1}^n (1/2 - \epsilon + 2\epsilon x_i)$.
 - i. Prove that for every threshold θ between 0 and n , if $n < 1/(100\epsilon)^2$ then the probabilities that $\sum x_i \leq \theta$ under P and Q respectively differ by at most 0.1. Therefore, one cannot use the test whether the number of heads is above or below some threshold to reliably distinguish between these two possibilities unless the number of samples n of the coins is at least some constant times $1/\epsilon^2$.
 - ii. Prove that for every function F mapping $\{0, 1\}^n$ to $\{0, 1\}$, if $n < 1/(100\epsilon)^2$ then the probabilities that $F(x) = 1$ under P and Q respectively differ by at most 0.1. Therefore, if the number of samples is smaller than a constant times $1/\epsilon^2$ then there is simply *no test* that can reliably distinguish between these two possibilities.

1

Introduction

Additional reading: Sections 2.1 (Introduction) and 2.2 (Shannon ciphers and perfect security) in the Boneh Shoup book. Chapters 1 and 2 of Katz-Lindell book.¹

Ever since people started to communicate, there were some messages that they wanted kept secret. Thus cryptography has an old though arguably *undistinguished* history. For a long time cryptography shared similar features with Alchemy as a domain in which many otherwise smart people would be drawn into making fatal mistakes.

The definitive text on the history of cryptography is David Kahn's "The Codebreakers", whose title already hints at the ultimate fate of most cryptosystems.² (See also "The Code Book" by Simon Singh.) We now recount just a few stories to get a feel for this field. But, before we do so, we should introduce the cast of characters. The basic setting of "encryption" or "secret writing" is the following: one person, whom we will call **Alice**, wishes to send another person, whom we will call **Bob**, a **secret** message. Since Alice and Bob are not in the same room (perhaps because Alice is imprisoned in a castle by her cousin the queen of England), they cannot communicate directly and need to send their message in writing. Alas, there is a third person, whom we will call **Eve**, that can see their message. Therefore Alice needs to find a way to *encode* or *encrypt* the message so that only Bob (and not Eve) will be able to understand it.

In 1587, Mary the queen of Scots, and the heir to the throne of England, wanted to arrange the assassination of her cousin, queen Elisabeth I of England, so that she could ascend to the throne and finally escape the house arrest under which she has been for the last 18 years. As part of this complicated plot, she sent a coded letter to Sir Anthony Babington. It is what's known as a *substitution cipher*

¹ In the current state of these lecture notes, almost all references and credits are omitted unless the name has become standard in the literature, or I believe that the story of some discovery can serve a pedagogical point. See the Katz-Lindell book for historical notes and references. This lecture shares a lot of text with (though is not identical to) my lecture on cryptography in the [introduction to theoretical computer science](#) lecture notes.

² Traditionally, *cryptography* was the name for the activity of *making* codes, while *cryptoanalysis* is the name for the activity of *breaking* them, and *cryptology* is the name for the union of the two. These days *cryptography* is often used as the name for the broad science of constructing and analyzing the security of not just encryptions but many schemes and protocols for protecting the confidentiality and integrity of communication and computation.

where each letter is transformed into a different symbol, and so the resulting letter looks something like the following (see Fig. 1.1):

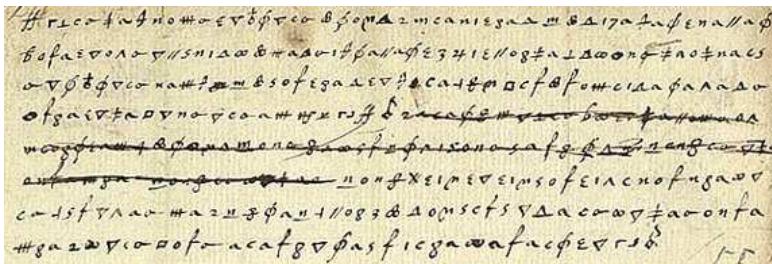


Figure 1.1: Snippet from encrypted communication between queen Mary and Sir Babington

At a first look, such a letter might seem rather inscrutable- a meaningless sequence of strange symbols. However, after some thought, one might recognize that these symbols *repeat* several times and moreover that different symbols repeat with different frequencies. Now it doesn't take a large leap of faith to assume that perhaps each symbol corresponds to a different letter and the more frequent symbols correspond to letters that occur in the alphabet with higher frequency. From this observation, there is a short gap to completely breaking the cipher, which was in fact done by Queen Elisabeth's spies who used the decoded letters to learn of all the co-conspirators and to convict Queen Mary of treason, a crime for which she was executed.

Trusting in superficial security measures (such as using "inscrutable" symbols) is a trap that users of cryptography have been falling into again and again over the years. As in many things, this is the subject of a great XKCD cartoon (see Fig. 1.2):

The **Vigenère cipher** is named after Blaise de Vigenère who described it in a book in 1586 (though it was invented earlier by Belaso). The idea is to use a collection of substitution ciphers - if there are n different ciphers then the first letter of the plaintext is encoded with the first cipher, the second with the second cipher, the n^{th} with the n^{th} cipher, and then the $n + 1^{st}$ letter is again encoded with the first cipher. The key is usually a word or a phrase of n letters, and the i^{th} substitution cipher is obtained by shifting each letter k_i positions in the alphabet. This "flattens" the frequencies and makes it much harder to do frequency analysis, which is why this cipher was considered "unbreakable" for 300+ years and got the nickname "le chiffre indéchiffrable" ("the unbreakable cipher"). Nevertheless, Charles Babbage cracked the Vigenère cipher in 1854 (though he did not publish it). In 1863 Friedrich Kasiski broke the cipher and published the result. The idea is that once you guess the length of the cipher, you can reduce the task to breaking a simple substitution cipher which

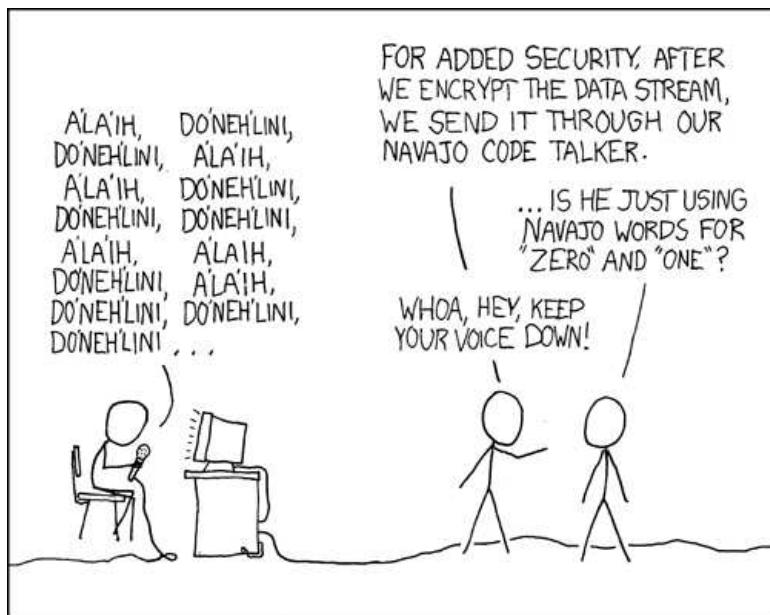


Figure 1.2: On the added security of using uncommon symbols

can be done via frequency analysis (can you see why?). Confederate generals used Vigenère regularly during the civil war, and their messages were routinely cryptanalyzed by Union officers.



Figure 1.3: Confederate Cipher Disk for implementing the Vigenère cipher

The story of the *Enigma* cipher had been told many times (see for example Kahn's book as well as Andrew Hodges' biography of Alan Turing). This was a mechanical cipher (looking like a typewriter) where each letter typed would get mapped into a different letter depending on the (rather complicated) key and current state of the machine which had several rotors that rotated at different paces. An



Figure 1.4: Confederate encryption of the message "Gen'l Pemberton: You can expect no help from this side of the river. Let Gen'l Johnston know, if possible, when you can attack the same point on the enemy's lines. Inform me also and I will endeavor to make a diversion. I have sent some caps. I subjoin a despatch from General Johnston."

identically wired machine at the other end could be used to decrypt. Just as many ciphers in history, this has also been believed by the Germans to be "impossible to break" and even quite late in the war they refused to believe it was broken despite mounting evidence to that effect. (In fact, some German generals refused to believe it was broken even *after* the war.) Breaking Enigma was an heroic effort which was initiated by the Poles and then completed by the British at Bletchley Park; as part of this effort they built arguably the world's first large scale mechanical computation devices (though they looked more similar to washing machines than to iPhones). They were also helped along the way by some quirks and errors of the german operators. For example, the fact that their messages ended with "Heil Hitler" turned out to be quite useful. Here is one entertaining anecdote: the Enigma machine would never map a letter to itself. In March 1941, Mavis Batey, a cryptanalyst at Bletchley Park received a very long message that she tried to decrypt. She then noticed a curious property—the message did *not* contain the letter "L".³ She realized that for such a long message not contain "L" could not happen by chance, and hence surmised that the original message probably composed *only* of L's. That is, it must have been the case that the operator, perhaps to test the machine, have simply sent out a message where he repeatedly pressed the letter "L". This observation helped her decode the next message, which helped inform of a planned Italian attack and secure a resounding British victory in what became known as "the Battle of Cape Matapan". Mavis also helped break another Enigma machine which helped in the effort of feeding the Germans with the false information that the main allied invasion would take place in Pas de Calais rather than on Normandy. See [this interview with Sir Harry Hinsley](#) for more on the effect of breaking the Enigma on the war. General Eisenhower said that the

³ Here is a nice exercise: compute (up to an order of magnitude) the probability that a 50-letter long message composed of random letters will end up not containing the letter "L".

intelligence from Bletchley park was of “priceless value” and made a “very decisive contribution to the Allied war effort”.

1.1 Defining encryptions

Many of the troubles that cryptosystem designers faced over history (and still face!) can be attributed to not properly defining or understanding what are the goals they want to achieve in the first place. We now turn to actually defining what is an encryption scheme.

Clearly we can encode every message as a string of bits, i.e., an element of $\{0,1\}^\ell$ for some ℓ . Similarly, we can encode the *key* as a string of bits as well, i.e., an element of $\{0,1\}^n$ for some n . Thus, we can think of an encryption scheme as composed of two functions.

The *encryption function* E maps a secret key $k \in \{0,1\}^n$ and a message (known also as *plaintext*) $m \in \{0,1\}^\ell$ into a *ciphertext* $c \in \{0,1\}^L$ for some L . We write this as $c = E_k(m)$. The *decryption function* D does the reverse operation, mapping the secret key k and the ciphertext c back into the plaintext message m , which we write as $m = D_k(c)$. The basic equation is that if we use the same key for encryption and decryption, then we should get the same message back. That is, for every $k \in \{0,1\}^n$ and $m \in \{0,1\}^\ell$,

$$m = D_k(E_k(m)) . \quad (1.1)$$

Formally, we make the following definition:

Definition 1.1 — Valid encryption scheme. A pair of functions (E, D) mapping strings to strings is a *valid private key encryption scheme* (or *encryption scheme* for short) if there are some numbers n, ℓ, L such that $E : \{0,1\}^n \times \{0,1\}^\ell \rightarrow \{0,1\}^L$ and $D : \{0,1\}^n \times \{0,1\}^L \rightarrow \{0,1\}^\ell$ and for every for every $k \in \{0,1\}^n$ and $x \in \{0,1\}^\ell$,

$$D(k, E(k, x)) = x . \quad (1.2)$$

We will typically write the first input (i.e., the key) to the encryption and decryption functions as a subscript, and so write Eq. (1.2) as $D_k(E_k(x)) = x$.

A note on notation: We will always use i, j, ℓ, n to denote natural numbers. n will often denote the length of our secret key, and ℓ the length of the message, sometimes also known as “block length” since longer messages are simply chopped into “blocks” of length ℓ and also appropriately padded.

We will use k to denote the secret key, m to denote the secret plaintext message, and c to denote the encrypted ciphertext. Note that c, m and k are bit strings of lengths ℓ , n and n respectively. The length of the secret key is often known as the “security parameter” and in other texts it is often denoted by k or κ . We use n to correspond with the standard algorithmic notation for input length (as in $O(n)$ time algorithms).

Definition 1.1 says nothing about security and does not rule out trivial “encryption” schemes such as the scheme $E_k(m) = m$ that simply outputs the plaintext as is. Defining security is tricky, and we’ll take it one step at a time, but let’s start by pondering what is secret and what is not. A priori we are thinking of an attacker Eve that simply sees the ciphertext $y = E_k(x)$ and does not know anything on how it was generated. So, it does not know the details of E and D , and certainly does not know the secret key k . However, many of the troubles past cryptosystems went through was caused by them relying on “security through obscurity”— trusting that the fact their *methods* are not known to their enemy will protect them from being broken. This is a faulty assumption - if you reuse a method again and again (even with a different key each time) then eventually your adversaries will figure out what you are doing. And if Alice and Bob meet frequently in a secure location to decide on a new method, they might as well take the opportunity to exchange their secrets.. These considerations led Kerchoffs to state the following principle:

A cryptosystem should be secure even if everything about the system, except the key, is public knowledge.
(Auguste Kerckhoffs, 1883)

(The actual quote is “Il faut qu’il n’exige pas le secret, et qu’il puisse sans inconvénient tomber entre les mains de l’ennemi” loosely translated as “The system must not require secrecy and can be stolen by the enemy without causing trouble”. According to Steve Bellovin the NSA version is “assume that the first copy of any device we make is shipped to the Kremlin”.)

Why is it OK to assume the key is secret and not the algorithm? Because we can always choose a fresh key. But of course if we choose our key to be “1234” or “password!” then that is not exactly secure. In fact, if you use any deterministic algorithm to choose the key then eventually your adversary will figure out. Therefore for security we

must choose the key at *random*. Thus following can be thought of as a restatement of Kerckoffs's principle:

There is no secrecy without randomness

This is such a crucial point that is worth repeating:

There is no secrecy without randomness

At the heart of every cryptographic scheme there is a secret key, and the secret key is always chosen at random. A corollary of that is that to understand cryptography, you need to know some probability theory. Fortunately, we don't need much of probability- only probability over finite spaces, and basic notions such as expectation, variance, concentration and the union bound suffice for most of what we need. In fact, understanding the following two statements will already get you much of what you need for cryptography:

- For every fixed string $x \in \{0, 1\}^n$, if you toss a coin n times, the probability that the heads/tails pattern will be exactly x is 2^{-n} .
- A probability of 2^{-128} is really really small.

1.1.1 Generating randomness in actual cryptographic systems

How do we actually get random bits in actual systems? The main idea is to use a two stage approach. First we need to get some data that is *unpredictable* from the point of view of an attacker on our system. Some sources for this could be measuring latency on the network or hard drives (getting harder with solid state disk), user keyboard and mouse movement patterns (problematic when you need fresh randomness at boot time), clock drift and more, there are some other sources including audio, video, and network. All of these can be problematic, especially for servers or virtual machines, and so hardware based random number generators based on phenomena such as thermal noise or nuclear decay are becoming more popular. Once we have some data X that is unpredictable, we need to estimate the *entropy* in it. You can roughly imagine that X has k bits of entropy if the probability that an attacker can guess X is at most 2^{-k} . People then use a *hash function* (an object we'll talk about more later) to map X into a string of length k which is then hopefully distributed (close to) uniformly at random. All of this process, and especially understanding the amount of information an attacker may have on

the entropy sources, is a bit of a dark art and indeed a number of attacks on cryptographic systems were actually enabled by weak generation of randomness. Here are a few examples.

One of the first attacks was on the SSL implementation of Netscape (*the* browser at the time). Netscape used the following “unpredictable” information—the time of day and a process ID both of which turned out to be quite predictable (who knew attackers have clocks too?). Netscape tried to protect its security through “security through obscurity” by not releasing the source code for their pseudorandom generator, but it was reverse engineered by **Ian Goldberg and David Wagner** (Ph.D students at the time) who demonstrated this attack.

In 2006 a programmer removed a line of code from the procedure to generate entropy in OpenSSL package distributed by Debian since it caused a warning in some automatic verification code. As a result for two years (until this was discovered) all the randomness generated by this procedure used only the process ID as an “unpredictable” source. This means that all communication done by users in that period is fairly easily breakable (and in particular, if some entities recorded that communication they could break it also retroactively). This caused a huge headache and a worldwide regeneration of keys, though it is believed that many of the weak keys are still used. See [XKCD’s take](#) on that incident.

```
int getRandomNumber()
{
    return 4; // chosen by fair dice roll.
              // guaranteed to be random.
}
```

Figure 1.5: XKCD Cartoon: Random number generator

In 2012 two separate teams of researchers scanned a large number of RSA keys on the web and found out that about 4 percent of them are easy to break. The main issue were devices such as routers, internet-connected printers and such. These devices sometimes run variants of Linux—a desktop operating system—but without a hard drive, mouse or keyboard, they don’t have access to many of the entropy sources that desktop have. Coupled with some good old fashioned ignorance of cryptography and software bugs, this led to many keys that are downright trivial to break, see [this blog post](#) and [this web page](#) for more details.

After the entropy is collected and then “purified” or “extracted” to a uniformly random string that is, say, a few hundred bits long, we often need to “expand” it into a longer string that is also uniform (or at least looks like that for all practical purposes). We will discuss how to go about that in the next lecture. This step has its weaknesses too and in particular the Snowden documents, combined with observations of Shumow and Ferguson, strongly suggest that the NSA has deliberately inserted a *trapdoor* in one of the pseudo-random generators published by the National Institute of Standards and Technologies (NIST). Fortunately, this generator wasn’t widely adapted but apparently the NSA did pay 10 million dollars to RSA security so the latter would make this generator their default option in their products.

1.2 Defining the secrecy requirement.

Defining the secrecy requirement for an encryption is not simple. Over the course of history, many smart people got it wrong and convinced themselves that ciphers were impossible to break. The first person to truly ask the question in a rigorous way was Claude Shannon in 1945 (though a partial version of his manuscript was only declassified in 1949). Simply by asking this question, he made an enormous contribution to the science of cryptography and practical security. We now will try to examine how one might answer it.

Let me warn you ahead of time that we are going to insist on a *mathematically precise definition* of security. That means that the definition must capture security in all cases, and the existence of a single counterexample, no matter how “silly”, would make us rule out a candidate definition. This exercise of coming up with “silly” counterexamples might seem, well, silly. But in fact it is this method that has led Shannon to formulate his theory of secrecy, which (after much followup work) eventually revolutionized cryptography, and brought this science to a new age where Edgar Allan Poe’s maxim no longer holds, and we are able to design ciphers which human (or even nonhuman) ingenuity cannot break.

The most natural way to attack an encryption is for Eve to guess all possible keys. In many encryption schemes this number is enormous and this attack is completely infeasible. For example, the theoretical number of possibilities in the Enigma cipher was about 10^{113} which roughly means that even if we built a filled the milky way galaxy with computers operating at light speed, the sun would still die out before it finished examining all the possibilities.⁴ One

⁴ There are about 10^{68} atoms in the galaxy, so even if we assumed that each one of those atoms was a computer that can process say 10^{21} decryption attempts per second (as the speed of light is 10^9 meters per second and the diameter of an atom is about 10^{-12} meters), then it would still take $10^{113-89} = 10^{24}$ seconds, which is about 10^{17} years to exhaust all possibilities, while the sun is estimated to burn out in about 5 billion years.

can understand why the Germans thought it was impossible to break. (Note that despite the number of possibilities being so enormous, such a key can still be easily specified and shared between Alice and Bob by writing down 113 digits on a piece of paper.) Ray Miller of the NSA had calculated that, in the way the Germans used the machine, the number of possibilities was “only” 10^{23} , but this is still extremely difficult to pull off even today, and many orders of magnitudes above the computational powers during the WW-II era. Thus clearly, it is sometimes possible to break an encryption without trying all possibilities. A corollary is that having a huge number of key combinations does not guarantee security, as an attacker might find a shortcut (as the allies did for Enigma) and recover the key without trying all options.

Since it is possible to recover the key with some tiny probability (e.g. by guessing it at random), perhaps one way to define security of an encryption scheme is that an attacker can never recover the key with probability significantly higher than that. Here is an attempt at such a definition:

Definition 1.2 — Security of encryption: first attempt. An encryption scheme (E, D) is n -secure if no matter what method Eve employs, the probability that she can recover the true key k from the ciphertext c is at most 2^{-n} .



When you see a mathematical definition that attempts to model some real-life phenomenon such as security, you should pause and ask yourself:

1. Do I understand mathematically what is the definition stating?
2. Is it a reasonable way to capture the real life phenomenon we are discussing?

One way to answer question 2 is to try to think of both examples of objects that satisfy the definition and examples of objects that violate it, and see if this conforms to your intuition about whether these objects display the phenomenon we are trying to capture. Try to do this for [Definition 1.2](#)

You might wonder if [Definition 1.2](#) is not *too strong*. After all how are we going ever to prove that Eve cannot recover the secret key no matter what she does? Edgar Allan Poe would say that there can always be a method that we overlooked. However, in fact this definition is too *weak*! Consider the following encryption: the secret key k is chosen at random in $\{0, 1\}^n$ but our encryption scheme simply ig-

nores it and lets $E_k(x) = x$ and $D_k(y) = y$. This is a valid encryption, but of course completely insecure as we are simply outputting the plaintext in the clear. Yet, no matter what Eve does, if she only sees c and not k , there is no way she can guess the true value of k with probability better than 2^{-n} , since it was chosen completely at random and she gets no information about it. Formally, one can prove the following result:

Lemma 1.3 Let (E, D) be the encryption scheme above. For every function $Eve : \{0, 1\}^\ell \rightarrow \{0, 1\}^n$ and for every $x \in \{0, 1\}^\ell$, the probability that $Eve(E_k(x)) = k$ is exactly 2^{-n} .

Proof. This follows because $E_k(x) = x$ and hence $Eve(E_k(x)) = Eve(x)$ which is some fixed value $k' \in \{0, 1\}^n$ that is independent of k . Hence the probability that $k = k'$ is 2^{-n} . QED ■

The math behind the above argument is very simple, yet I urge you to read and re-read the last two paragraphs until you are sure that you completely understand why this encryption is in fact secure according to the above definition. This is a “toy example” of the kind of reasoning that we will be employing constantly throughout this course, and you want to make sure that you follow it.

So, Lemma 1.3 is true, but one might question its meaning. Clearly this silly example was not what we meant when stating this definition. However, as mentioned above, we are not willing to ignore even silly examples and must amend the definition to rule them out. One obvious objection is that we don’t care about hiding the key- it is the *message* that we are trying to keep secret. This suggests the next attempt:

Definition 1.4 — Security of encryption: second attempt. An encryption scheme (E, D) is n -secure if for every message x no matter what method Eve employs, the probability that she can recover x from the ciphertext $y = E_k(x)$ is at most 2^{-n} .

Now this seems like it captures our intended meaning. But remember that we are being anal, and truly insist that the definition holds as stated, namely that for every plaintext message x and every function $Eve : \{0, 1\}^L \rightarrow \{0, 1\}^\ell$, the probability over the choice of k that $Eve(E_k(x)) = x$ is at most 2^{-n} . But now we see that this is clearly impossible. After all, this is supposed to work for *every* message x and *every* function Eve , but clearly if x is the all-zeroes message 0^ℓ and Eve is the function that ignores its input and simply outputs 0^ℓ , then it will hold that $Eve(E_k(x)) = x$ with probability one.

So, if before the definition was too weak, the new definition is too strong and is impossible to achieve. The problem is that of course we could guess a fixed message with probability one, so perhaps we could try to consider a definition with a *random* message. That is:

Definition 1.5 — Security of encryption: third attempt. An encryption scheme (E, D) is n -secure if no matter what method Eve employs, if x is chosen at random from $\{0, 1\}^\ell$, the probability that she can recover x from the ciphertext $c = E_k(x)$ is at most 2^{-n} .

This weakened definition can in fact be achieved, but we have again weakened it too much. Consider an encryption that hides the last $\ell/2$ bits of the message, but completely reveals the first $\ell/2$ bits. The probability of guessing a random message is $2^{-\ell/2}$, and so such a scheme would be “ $\ell/2$ secure” per Definition 1.5 but this is still a scheme that you would not want to use. The point is that in practice we don’t encrypt random messages—our messages might be in English, might have common headers, and might have even more structures based on the context. In fact, it may be that the message is either “Yes” or “No” (or perhaps either “Attack today” or “Attack tomorrow”) but we want to make sure Eve doesn’t learn which one it is. So, using an encryption scheme that reveals the first half of the message (or frankly even only the first bit) is unacceptable.

1.3 Perfect Secrecy

So far all of our attempts at definitions oscillated between being too strong (and hence impossible) or too weak (and hence not guaranteeing actual security). The key insight of Shannon was that in a secure encryption scheme the ciphertext should not reveal *any additional information* about the plaintext. So, if for example it was a priori possible for Eve to guess the plaintext with some probability $1/k$ (e.g., because there were only k possibilities for it) then she should not be able to guess it with higher probability after seeing the ciphertext. This can be formalized as follows:

Definition 1.6 — Perfect secrecy. An encryption scheme (E, D) is perfectly secret if there for every set $M \subseteq \{0, 1\}^\ell$ of plaintexts, and for every strategy used by Eve, if we choose at random $x \in M$ and a random key $k \in \{0, 1\}^n$, then the probability that Eve guesses x after seeing $E_k(x)$ is at most $1/|M|$.

In particular, if we encrypt either “Yes” or “No” with probability

$1/2$, then Eve won't be able to guess which one it is with probability better than half. In fact, that turns out to be the heart of the matter:

Theorem 1.7 — Two to many theorem. An encryption scheme (E, D) is perfectly secret if and only if for every two distinct plaintexts $\{x_0, x_1\} \subseteq \{0, 1\}^\ell$ and every strategy used by Eve, if we choose at random $b \in \{0, 1\}$ and a random key $k \in \{0, 1\}^n$, then the probability that Eve guesses x_b after seeing $E_k(x_b)$ is at most $1/2$.

Proof. The “only if” direction is obvious—this condition is a special case of the perfect secrecy condition for a set M of size 2.

The “if” direction is trickier. We need to show that if there is some set M (of size possibly much larger than 2) and some strategy for Eve to guess (based on the ciphertext) a plaintext chosen from M with probability larger than $1/|M|$, then there is also some set M' of size two and a strategy Eve' for Eve to guess a plaintext chosen from M' with probability larger than $1/2$.

Let's fix the message x_0 to be the all zeroes message and pick x_1 at random in M . Under our assumption, it holds that for random key k and message $x_1 \in M$,

$$\mathbb{P}_{k \leftarrow \{0,1\}^n, x_1 \leftarrow M} [Eve(E_k(x_1)) = x_1] > 1/|M|. \quad (1.3)$$

On the other hand, for every choice of k , $x' = Eve(E_k(x_0))$ is a fixed string independent on the choice of x_1 , and so if we pick x_1 at random in M , then the probability that $x_1 = x'$ is at most $1/|M|$, or in other words

$$\mathbb{P}_{k \leftarrow \{0,1\}^n, x_1 \leftarrow M} [Eve(E_k(x_0)) = x_1] \leq 1/|M|. \quad (1.4)$$

Thus in particular, due to linearity of expectation, there exists some x_1 satisfying

$$\mathbb{P}[Eve(E_k(x_1)) = x_1] > \mathbb{P}[Eve(E_k(x_0)) = x_1]. \quad (1.5)$$

(Can you see why? This is worthwhile stopping and reading again.) But this can be turned into an attacker Eve' such that for $b \leftarrow_R \{0, 1\}$, the probability that $Eve'(E_k(x_b)) = x_b$ is larger than $1/2$. Indeed, we can define $Eve'(y)$ to output x_1 if $Eve(y) = x_1$ and otherwise output a random message in $\{x_0, x_1\}$. The probability that $Eve'(y)$ equals x_1 is higher when $y = E_k(x_1)$ than when $y = E_k(x_0)$, and since Eve' outputs either x_0 or x_1 , this means that the probability that $Eve'(E_k(x_b)) = x_b$ is larger than $1/2$. (Can you see why?) ■

P The proof of [Theorem 1.7](#) is not trivial, and is worth reading again and making sure you understand it. An excellent exercise, which I urge you to pause and do now is to prove the following: (E, D) is perfectly secret if for every plaintexts $x, x' \in \{0, 1\}^\ell$, the two random variables $\{E_k(x)\}$ and $\{E_{k'}(x')\}$ (for randomly chosen keys k and k') have precisely the same distribution.

So, perfect secrecy is a natural condition, and does not seem to be too weak for applications, but can it actually be achieved? After all, the condition that two different plaintexts are mapped to the same distribution seems somewhat at odds with the condition that Bob would succeed in decrypting the ciphertexts and find out if the plaintext was in fact x or x' . It turns out the answer is yes! For example, [Fig. 1.6](#) details a perfectly secret encryption for two bits.

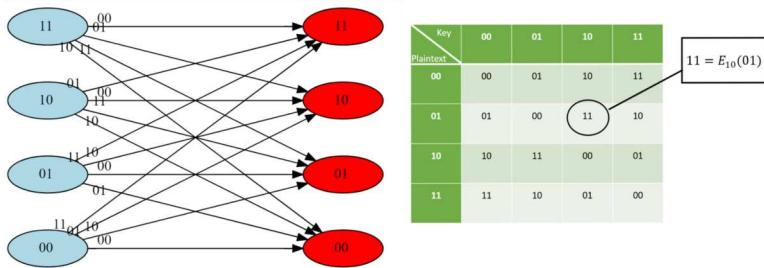


Figure 1.6: A perfectly secret encryption scheme for two-bit keys and messages. The blue vertices represent plaintexts and the red vertices represent ciphertexts, each edge mapping a plaintext x to a ciphertext $y = E_k(x)$ is labeled with the corresponding key k . Since there are four possible keys, the degree of the graph is four and it is in fact a complete bipartite graph. The encryption scheme is valid in the sense that for every $k \in \{0, 1\}^2$, the map $x \mapsto E_k(x)$ is one-to-one, which in other words means that the set of edges labeled with k is a *matching*.

In fact, this can be generalized to any number of bits:

Theorem 1.8 — One Time Pad (Vernam 1917, Shannon 1949). There is a perfectly secret valid encryption scheme (E, D) with $L(n) = n$.

Proof Idea: Our scheme is the **one-time pad** also known as the “Vernam Cipher”, see [Fig. 1.8](#). The encryption is exceedingly simple: to encrypt a message $x \in \{0, 1\}^n$ with a key $k \in \{0, 1\}^n$ we simply output $x \oplus k$ where \oplus is the bitwise XOR operation that outputs the string corresponding to XORing each coordinate of x and k .

Proof of Theorem 1.8. For two binary strings a and b of the same length n , we define $a \oplus b$ to be the string $c \in \{0, 1\}^n$ such that $c_i = a_i +$

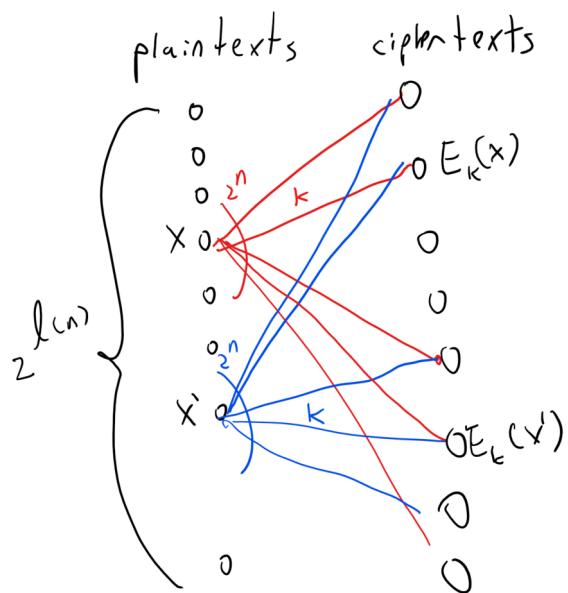


Figure 1.7: For any key length n , we can visualize an encryption scheme (E, D) as a graph with a vertex for every one of the $2^{L(n)}$ possible plaintexts and for every one of the ciphertexts in $\{0, 1\}^*$ of the form $E_k(x)$ for $k \in \{0, 1\}^n$ and $x \in \{0, 1\}^{L(n)}$. For every plaintext x and key k , we add an edge labeled k between x and $E_k(x)$. By the validity condition, if we pick any fixed key k , the map $x \mapsto E_k(x)$ must be one-to-one. The condition of perfect secrecy simply corresponds to requiring that every two plaintexts x and x' have exactly the same set of neighbors (or multi-set, if there are parallel edges).

$b_i \bmod 2$ for every $i \in [n]$. The encryption scheme (E, D) is defined as follows: $E_k(x) = x \oplus k$ and $D_k(y) = y \oplus k$. By the associative law of addition (which works also modulo two), $D_k(E_k(x)) = (x \oplus k) \oplus k = x \oplus (k \oplus k) = x \oplus 0^n = x$, using the fact that for every bit $\sigma \in \{0, 1\}$, $\sigma + \sigma \bmod 2 = 0$ and $\sigma + 0 = \sigma \bmod 2$. Hence (E, D) form a valid encryption.

To analyze the perfect secrecy property, we claim that for every $x \in \{0, 1\}^n$, the distribution $Y_x = E_k(x)$ where $k \sim \{0, 1\}^n$ is simply the uniform distribution over $\{0, 1\}^n$, and hence in particular the distributions Y_x and $Y_{x'}$ are identical for every $x, x' \in \{0, 1\}^n$. Indeed, for every particular $y \in \{0, 1\}^n$, the value y is output by Y_x if and only if $y = x \oplus k$ which holds if and only if $k = x \oplus y$. Since k is chosen uniformly at random in $\{0, 1\}^n$, the probability that k happens to equal $x \oplus y$ is exactly 2^{-n} , which means that every string y is output by Y_x with probability 2^{-n} . ■

Key:	1	0	1	1	0	0	1	1	1	0	0	1
\oplus												
Plaintext:	0	1	1	0	1	0	0	0	1	1	0	1
—————												
Ciphertext:	1	1	0	1	1	0	1	1	0	1	0	0

Figure 1.8: In the one time pad encryption scheme we encrypt a plaintext $x \in \{0, 1\}^n$ with a key $k \in \{0, 1\}^n$ by the ciphertext $x \oplus k$ where \oplus denotes the bitwise XOR operation.



The argument above is quite simple but is worth reading again. To understand why the one-time pad is perfectly secret, it is useful to envision it as a bipartite graph as we've done in Fig. 1.6. (In fact the encryption scheme of Fig. 1.6 is precisely the one-time pad for $n = 2$.) For every n , the one-time pad encryption scheme corresponds to a bipartite graph with 2^n vertices on the “left side” corresponding to the plaintexts in $\{0, 1\}^n$ and 2^n vertices on the “right side” corresponding to the ciphertexts $\{0, 1\}^n$. For every $x \in \{0, 1\}^n$ and $k \in \{0, 1\}^n$, we connect x to the vertex $y = E_k(x)$ with an edge that we label with k . One can see that this is the complete bipartite graph, where every vertex on the left is connected to all vertices on the right. In particular this means that for every left vertex x , the distribution on the ciphertexts obtained by taking a random $k \in \{0, 1\}^n$ and going to the neighbor of x on the edge labeled k is the uniform distribution over $\{0, 1\}^n$. This ensures the perfect secrecy condition.

1.4 Necessity of long keys

So, does Theorem 1.8 give the final word on cryptography, and means that we can all communicate with perfect secrecy and live happily ever after? No it doesn't. While the one-time pad is efficient, and gives perfect secrecy, it has one glaring disadvantage: to communicate n bits you need to store a key of length n . In contrast, practically used cryptosystems such as AES-128 have a short key of 128 bits (i.e., 16 bytes) that can be used to protect terabytes or more of communication! Imagine that we all needed to use the one time pad. If that was the case, then if you had to communicate with m people, you would have to maintain (securely!) m huge files that are each as long as the length of the maximum total communication you expect with that person. Imagine that every time you opened an account with Amazon, Google, or any other service, they would need to send you in the mail (ideally with a secure courier) a DVD full of random numbers, and every time you suspected a virus, you'd need to ask all these services for a fresh DVD. This doesn't sound so appealing.

This is not just a theoretical issue. The Soviets have used the one-time pad for their confidential communication since before the 1940's. In fact, even before Shannon's work, the U.S. intelligence already knew in 1941 that the one-time pad is in principle "unbreakable" (see page 32 in the [Venona document](#)). However, it turned out that the hassle of manufacturing so many keys for all the communication took its toll on the Soviets and they ended up reusing the same keys for more than one message. They did try to use them for completely different receivers in the (false) hope that this wouldn't be detected. The [Venona Project](#) of the U.S. Army was founded in February 1943 by Gene Gabeel (see Fig. 1.9), a former home economics teacher from Madison Heights, Virginia and Lt. Leonard Zubko. In October 1943, they had their breakthrough when it was discovered that the Russians were reusing their keys.⁵ In the 37 years of its existence, the project has resulted in a treasure chest of intelligence, exposing hundreds of KGB agents and Russian spies in the U.S. and other countries, including Julius Rosenberg, Harry Gold, Klaus Fuchs, Alger Hiss, Harry Dexter White and many others.

Unfortunately it turns out that (as shown by Shannon) that such long keys are *necessary* for perfect secrecy:

Theorem 1.9 — Perfect secrecy requires long keys. For every perfectly secret encryption scheme (E, D) the length function L satisfies $L(n) \leq n$.

⁵ Credit to this discovery is shared by Lt. Richard Hallock, Carrie Berry, Frank Lewis, and Lt. Karl Elmquist, and there are others that have made important contribution to this project. See pages 27 and 28 in the document.



Figure 1.9: Gene Grabeel, who founded the U.S. Russian SigInt program on 1 Feb 1943.
Photo taken in 1942, see Page 7 in the Venona historical study.

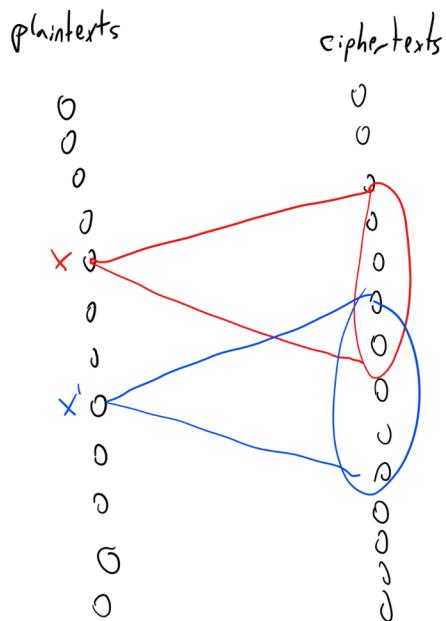


Figure 1.10: An encryption scheme where the number of keys is smaller than the number of plaintexts corresponds to a bipartite graph where the degree is smaller than the number of vertices on the left side. Together with the validity condition this implies that there will be two left vertices x, x' with non-identical neighborhoods, and hence the scheme does *not* satisfy perfect secrecy.

Proof Idea: The idea behind the proof is illustrated in Fig. 1.10. If the number of keys is smaller than the number of messages then the neighborhoods of all vertices in the corresponding graphs cannot be identical.

Proof of Theorem 1.9. Let E, D be a valid encryption scheme with messages of length L and key of length $n < L$. We will show that (E, D) is not perfectly secret by providing two plaintexts $x_0, x_1 \in \{0, 1\}^L$ such that the distributions Y_{x_0} and Y_{x_1} are not identical, where Y_x is the distribution obtained by picking $k \sim \{0, 1\}^n$ and outputting $E_k(x)$. We choose $x_0 = 0^L$. Let $S_0 \subseteq \{0, 1\}^*$ be the set of all ciphertexts that have nonzero probability of being output in Y_{x_0} . That is, $S = \{y \mid \exists_{k \in \{0, 1\}^n} y = E_k(x_0)\}$. Since there are only 2^n keys, we know that $|S_0| \leq 2^n$.

We will show the following claim:

Claim I: There exists some $x_1 \in \{0, 1\}^L$ and $k \in \{0, 1\}^n$ such that $E_k(x_1) \notin S_0$.

Claim I implies that the string $E_k(x_1)$ has positive probability of being output by Y_{x_1} and zero probability of being output by Y_{x_0} and hence in particular Y_{x_0} and Y_{x_1} are not identical. To prove Claim I, just choose a fixed $k \in \{0, 1\}^n$. By the validity condition, the map $x \mapsto E_k(x)$ is a one to one map of $\{0, 1\}^L$ to $\{0, 1\}^*$ and hence in particular the *image* of this map: the set $I = \{y \mid \exists_{x \in \{0, 1\}^L} y = E_k(x)\}$ has size at least (in fact exactly) 2^L . Since $|S_0| = 2^n < 2^L$, this means that $|I| > |S_0|$ and so in particular there exists some string y in $I \setminus S_0$. But by the definition of I this means that there is some $x \in \{0, 1\}^L$ such that $E_k(x) \notin S_0$ which concludes the proof of Claim I and hence of Theorem 1.9. ■

1.4.1 Advanced comment: Adding probability into the picture

There is a sense in which both our secrecy and our impossibility results might not be fully convincing, and that is that we did not explicitly consider algorithms that use *randomness*. For example, maybe Eve can break a perfectly secret encryption if she is not modeled as a deterministic function $Eve : \{0, 1\}^o \rightarrow \{0, 1\}^\ell$ but rather a *probabilistic* process. Similarly, maybe the encryption and decryption functions could be probabilistic processes as well. It turns out that none of those matter. For the former, note that a probabilistic process can be thought of as a *distribution* over functions, in the sense that we have a collection of functions f_1, \dots, f_N mapping $\{0, 1\}^o$ to $\{0, 1\}^\ell$, and some

probabilities p_1, \dots, p_N (non-negative numbers summing to 1), so we now think of Eve as selecting the function f_i with probability p_i . But if none of those functions can give an advantage better than $1/2$, then neither can this collection. A similar (though more involved) argument shows that the impossibility result showing that the key must be at least as long as the message still holds even if the encryption and decryption algorithms are allowed to be probabilistic processes as well (working this out is a great exercise).

2

Computational Secrecy

Additional reading: Sections 2.3 and 2.4 in Boneh-Shoup book.
Chapter 3 up to and including Section 3.3 in Katz-Lindell book.

Recall our cast of characters- Alice and Bob want to communicate securely over a channel that is monitored by the nosy Eve. In the last lecture, we have seen the definition of *perfect secrecy* that guarantees that Eve cannot learn *anything* about their communication beyond what she already knew. However, this security came at a price. For every bit of communication, Alice and Bob have to exchange in advance a bit of a secret key. In fact, the proof of this result gives rise to the following simple Python program that can break every encryption scheme that uses, say, a 128 bit key, with a 129 bit message:

```
# Gets ciphertext as input and two potential plaintexts
# Positive return value means first is more likely,
# negative means second is more likely,
# 0 means both have same likelihood.
#
# We assume we have access to the function Decrypt(key,
#         ciphertext)
def Distinguish(ciphertext,plaintext1,plaintext2):
    bias = 0
    key = [0]*128 #128 0's
    while(sum(key)<128):
        p = Decrypt(key,ciphertext)
        if p==plaintext1: bias++
        if p==plaintext2: bias--
        increment(key)
    return bias

# increment key when thought of as a number sorted from
# least significant
```

```
# to most significant bit. Assume not all bits are 1.
def increment(key):
    i = key.index(0);
    for j in range(i-1): key[j]=0
    key[i]=1
```

Now, generating, distributing, and protecting huge keys causes immense logistical problems, which is why almost all encryption schemes used in practice do in fact utilize short keys (e.g., 128 bits long) with messages that can be much longer (sometimes even terabytes or more of data).

So, why can't we use the above Python program to break all encryptions in the Internet and win infamy and fortune? We can in fact, but we'll have to wait a *really* long time, since the loop in *Distinguish* will run 2^{128} times, which will take much more than the lifetime of the universe to complete, even if we used all the computers on the planet.

However, the fact that *this* particular program is not a feasible attack, does not mean there does not exist a different attack. But this still suggests a tantalizing possibility: if we consider a relaxed version of perfect secrecy that restricts Eve to performing computations that can be done in this universe (e.g., less than 2^{256} steps should be safe not just for human but for all potential alien civilizations) then can we bypass the impossibility result and allow the key to be much shorter than the message?

This in fact does seem to be the case, but as we've seen, defining security is a subtle task, and will take some care. As before, the way we avoid (at least some of) the pitfalls of so many cryptosystems in history is that we insist on very precisely *defining* what it means for a scheme to be secure.

Let us defer the discussion how one defines a function being computable in "less than T operations" and just say that there is a way to formally do so. Given the perfect secrecy definition we saw last time, a natural attempt for defining Computational secrecy would be the following:

Definition 2.1 — Computational secrecy (first attempt). An encryption scheme (E, D) has t bits of computational secrecy if for every two distinct plaintexts $\{m_0, m_1\} \subseteq \{0, 1\}^\ell$ and every strategy of Eve using at most 2^t computational steps, if we choose at random $b \in \{0, 1\}$ and a random key $k \in \{0, 1\}^n$, then the probability that Eve guesses m_b after seeing $E_k(m_b)$ is at most $1/2$.¹

Definition 2.1 seems very natural, but is in fact *impossible* to achieve if the key is shorter than the message.

P

Before reading further, you might want to stop and think if you can *prove* that there is no, say, \sqrt{n} secure encryption scheme satisfying **Definition 2.1** with $\ell = n + 1$ and where the time to compute the encryption is polynomial.

The reason **Definition 2.1** can't be achieved is that if the message is even one bit longer than the key, we can always have a very efficient procedure that achieves success probability of about $1/2 + 2^{-n-1}$ by guessing the key. That is, we can replace the loop in the Python program `Distinguish` by choosing the key at random. Since we have some small chance of guessing correctly, we will get a small advantage over half.

To fix this definition, we do not consider guessing with such a tiny advantage as a “true break” of the scheme, and hence this will be the actual definition we use.

Definition 2.2 — Computational secrecy (concrete). An encryption scheme (E, D) has t bits of computational secrecy² if for every two distinct plaintexts $\{m_0, m_1\} \subseteq \{0,1\}^\ell$ and every strategy of Eve using at most 2^t computational steps, if we choose at random $b \in \{0,1\}$ and a random key $k \in \{0,1\}^n$, then the probability that Eve guesses m_b after seeing $E_k(m_b)$ is at most $1/2 + 2^{-t}$.

Having learned our lesson, let's try to see that this strategy does give us the kind of conditions we desired. In particular, let's verify that this definition implies the analogous condition to perfect secrecy.

Theorem 2.3 — Guessing game for computational secrecy. If (E, D) is has t bits of Computational secrecy as per **Definition 2.2** then every subset $M \subseteq \{0,1\}^\ell$ and every strategy of Eve using at most $2^t - (100\ell + 100)$ computational steps, if we choose at random $m \in M$ and a random key $k \in \{0,1\}^n$, then the probability that Eve guesses m after seeing $E_k(m_b)$ is at most $1/|M| + 2^{-t+1}$.

Before proving this theorem note that it gives us a pretty strong guarantee. In the exercises we will strengthen it even further showing that no matter what prior information Eve had on the message before, she will never get any non-negligible new information on it. One way to phrase it is that if the sender used a 256-bit secure en-

¹ It is important to keep track of what is known and unknown to the adversary Eve. The adversary knows the set $\{m_0, m_1\}$ of potential messages, and the ciphertext $y = E_k(m_b)$. The only things she doesn't know are whether $b = 0$ or $b = 1$, and the value of the secret key k . In particular, because m_0 and m_1 are known to Eve, it does not matter whether we define Eve's goal in this “security game” as outputting m_b or as outputting b .

² This is a slight simplification of the typical notion of “ t bits of security”. In the more standard definition we'd say that a scheme has t bits of security if for every $t_1 + t_2 \leq t$, an attacker running in 2^{t_1} time can't get success probability advantage more than 2^{-t_2} . However these two definitions only differ from one another by at most a factor of two. This may be important for practical applications (where the difference between 64 and 32 bits of security could be crucial) but won't matter for our concerns.

cryption to encrypt a message, then your chances of getting to learn any additional information about it before the universe collapses are more or less the same as the chances that a fairy will materialize and whisper it in your ear.



Before reading the proof, try to again review the proof of [Theorem 1.7](#), and see if you can generalize it yourself to the computational setting.

Proof of Theorem 2.3. The proof is rather similar to the equivalence of guessing one of two messages vs. one of many messages for perfect secrecy (i.e., [Theorem 1.7](#)). However, in the computational context we need to be careful in keeping track of Eve's running time. In the proof of [Theorem 1.7](#) we showed that if there exists:

- A subset $M \subseteq \{0, 1\}^\ell$ of messages
and
- An adversary $Eve : \{0, 1\}^0 \rightarrow \{0, 1\}^\ell$ such that

$$\mathbb{P}_{m \leftarrow_R M, k \leftarrow_R \{0, 1\}^n} [Eve(E_k(m)) = m] > 1/|M| \quad (2.1)$$

Then there exist two messages m_0, m_1 and an adversary $Eve' : \{0, 1\}^0 \rightarrow \{0, 1\}^\ell$ such that $\mathbb{P}_{b \leftarrow_R \{0, 1\}, k \leftarrow_R \{0, 1\}^n} [Eve'(E_k(m_b)) = m_b] > 1/2$.

To adapt this proof to the computational setting and complete the proof of the current theorem it suffices to show that:

- If the probability of Eve succeeding was $\frac{1}{|M|} + \epsilon$ then the probability of Eve' succeeding is at least $\frac{1}{2} + \epsilon/2$.
- If Eve can be computed in T operations, then Eve' can be computed in $T + 100\ell + 100$ operations.

This will imply that if Eve ran in polynomial time and had polynomial advantage over $1/|M|$ in guessing a plaintext chosen from M , then Eve' would run in polynomial time and have polynomial advantage over $1/2$ in guessing a plaintext chosen from $\{m_0, m_1\}$.

The first item can be shown by simply doing the same proof more carefully, keeping track how the advantage over $\frac{1}{|M|}$ for Eve translates into an advantage over $\frac{1}{2}$ for Eve' . As the world's most annoying saying goes, doing this is an excellent exercise for the reader. The item point is obtained by looking at the definition of

Eve' from that proof. On input c , Eve' computed $m = Eve(c)$ (which costs T operations), checked if $m = m_0$ (which costs, say, at most 5ℓ operations), and then outputted either 1 or a random bit (which is a constant, say at most 100 operations). ■

2.0.1 Proof by reduction

The proof of [Theorem 2.3](#) is a model to how a great many of the results in this course will look like. Generally we will have many theorems of the form:

"If there is a scheme S' satisfying security definition X' then there is a scheme S satisfying security definition $X"$

In the context of [Theorem 2.3](#), X' was "having t bits of security" (in the context distinguishing between encryptions of two ciphertexts) and X was the more general notion of hardness of getting a non-trivial advantage over guessing for an encryption of a random $m \in M$. While in [Theorem 2.3](#) the encryption scheme S was the same as S' , this need not always be the case. However, all of the proofs of such statements will have the same global structure—we will assume towards a contradiction, that there is an efficient adversary strategy Eve demonstrating that the scheme S violates the security notion X , and build from Eve a strategy Eve' demonstrating that S' violates X . This is such an important point that it deserves repeating:

The way you show that if S' is secure then S is secure is by giving a transformation from an adversary that breaks S into an adversary that breaks S'

For computational secrecy, we will always want that Eve' will be efficient if Eve is, and that will usually be the case because Eve' will simply use Eve as a black box, which it will not invoke too many times, and addition will use some polynomial time preprocessing and postprocessing. The more challenging parts of such proofs are typically:

- Coming up with the strategy Eve' .
- Analyzing the probability of success and in particular showing that if Eve had non-negligible advantage then so will Eve' .

Note that, just like in the context of NP completeness or uncomputability reductions, security reductions work *backwards*. That is, we

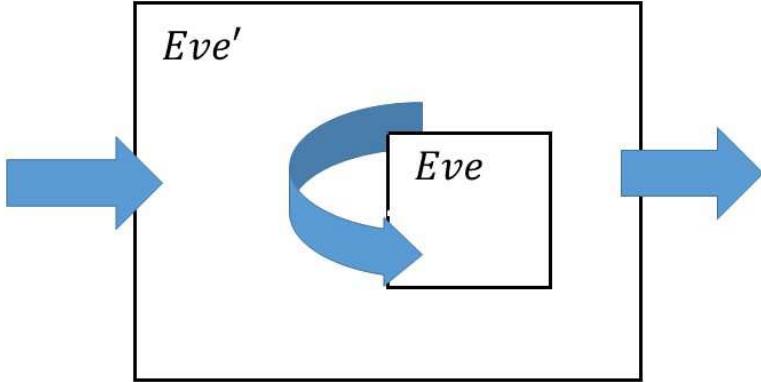


Figure 2.1: We show that the security of S' implies the security of S by transforming an adversary Eve breaking S into an adversary Eve' breaking S'

construct the scheme S based on the scheme S' , but then prove that we can transform an algorithm breaking S into an algorithm breaking S' . Just like in computational complexity, it can sometimes be hard to keep track of the direction of the reduction. In fact, cryptographic reductions can be even subtler, since they involve an interplay of several entities (for example, sender, receiver, and adversary) and probabilistic choices (e.g., over the message to be sent and the key).

2.1 The asymptotic approach

For practical security, often every bit of security matters. We want our keys to be as short as possible and our schemes to be as fast as possible while satisfying a particular level of security. However, for understanding the *principles* behind cryptography, keeping track of those bits can be a distraction, and so just like we do for algorithms, we will use *asymptotic analysis* (also known as *big Oh notation*) to sweep many of those details under the carpet.

To a first approximation, there will be only two types of running times we will encounter in this course:

- *Polynomial* running time of the form $d \cdot n^c$ for some constants $d, c > 0$ (or $\text{poly}(n) = n^{O(1)}$ for short), which we will consider as *efficient*
- *Exponential* running time of the form $2^{d \cdot n^\epsilon}$ for some constants $d, \epsilon > 0$ (or $2^{n^{\Omega(1)}}$ for short) which we will consider as *infeasible*.³

Another way to say it is that in this course, if a scheme has any security at all, it will have at least n^ϵ bits of security where n is

³ Some texts reserve the term *exponential* to functions of the form 2^{en} for some $e > 0$ and call a function such as, say, $2\sqrt{n}$ *subexponential*. However, we will generally not make this distinction in this course.

the length of the key and $\epsilon > 0$ is some absolute constant such as $\epsilon = 1/3$.

These are not all the theoretically possible running times. One can have intermediate functions such as $n^{\log n}$ though we will generally not encounter those. To make things clean (and to correspond to standard terminology), we will say that an algorithm A is *efficient* if it runs in time $\text{poly}(n)$ when n is its input length (which will always be the same, up to polynomial factors, as the key length). If $\mu(n)$ is some probability that depends on the input/key length parameter n , then we say that $\mu(n)$ is *negligible* if it's smaller than every polynomial. That is, for every c, d there is some N , such that if $n > N$ then $\mu(n) < 1/(cn)^d$. Note that for every non-constant polynomials p, q , $\mu(n)$ is negligible if and only if the function $\mu'(n) = p(\mu(q(n)))$ is negligible.

R

Asymptotic analysis The above definitions could be confusing if you haven't encountered asymptotic analysis before. Reading the beginning of Chapter 3 (pages 43-51) in the KL book, as well as the mathematical background lecture in my [intro to TCS notes](#) can be extremely useful. As a rule of thumb, if every time you see the word "polynomial" you imagine the function n^{10} and every time you see the word "negligible" you imagine the function $2^{-\sqrt{n}}$ then you will get the right intuition.

What you need to remember is that negligible is much smaller than any inverse polynomial, while polynomials are closed under multiplication, and so we have the "equations" $\text{negligible} \times \text{polynomial} = \text{negligible}$ and $\text{polynomial} \times \text{polynomial} = \text{polynomial}$. As mentioned, in practice people really want to get as close as possible to n bits of security with an n bit key, but we would be happy as long as the security grows with the key, so when we say a scheme is "secure" you can think of it having \sqrt{n} bits of security (though any function growing faster than $\log n$ would be fine as well).

From now on, we will require all of our encryption schemes to be *efficient* which means that the encryption and decryption algorithms should run in polynomial time. Security will mean that any efficient adversary can make at most a negligible gain in the probability of guessing the message over its a priori probability.⁴ That is, we make the following definition:

⁴ Note that there is a subtle issue here with the order of quantifiers. For a scheme to be efficient, the algorithms such as encryption and decryption need to run in some *fixed* polynomial time such as n^2 or n^3 . In contrast we allow the adversary to run in *any* polynomial time. That is, for every c , if n is large enough, then the scheme should be secure against an adversary that runs in time n^c . This is a general principle in cryptography that we always allow the adversary potentially much more resources than those used by the honest users. In practical security we often assume that the gap between the honest use and the adversary resources can be *exponential*. For example, a low power embedded device can encrypt messages that, as far as we know, are undecipherable even by a nation-state using super-computers and massive data centers.

Definition 2.4 — Computational secrecy (asymptotic). An encryption scheme (E, D) is *computationally secret* if for every two distinct plaintexts $\{m_0, m_1\} \subseteq \{0, 1\}^\ell$ and every efficient (i.e., polynomial time) strategy of Eve, if we choose at random $b \in \{0, 1\}$ and a random key $k \in \{0, 1\}^n$, then the probability that Eve guesses m_b after seeing $E_k(m_b)$ is at most $1/2 + \mu(n)$ for some negligible function $\mu(\cdot)$.

2.1.1 Counting number of operations.

One more detail that we've so far ignored is what does it mean exactly for a function to be computable using at most T operations. Fortunately, when we don't really care about the difference between T and, say, T^2 , then essentially every reasonable definition gives the same answer. Formally, we can use the notions of Turing machines, Boolean circuits, or straightline programs to define complexity. For concreteness, let's define that a function $F : \{0, 1\}^n \rightarrow \{0, 1\}^m$ has complexity at most T if there is a Boolean circuit that computes F using at most T NAND gates (or equivalently, there is a NAND program computing F in at most T lines). (There is nothing special about NAND, and we can use any other universal gate set.) We will often also consider *probabilistic* functions in which case we allow the circuit a RAND gate that outputs a single random bit (though this in general does not give extra power). The fact that we only care about asymptotics means you don't really need to think of gates, etc.. when arguing in cryptography. However, it is comforting to know that this notion has a precise mathematical formulation.

2.2 Our first conjecture

We are now ready to make our first conjecture:

The Cipher Conjecture:⁵ There exists a computationally secret encryption scheme (E, D) (where E, D are efficient) with a key of size n for messages of size $n + 1$.

A *conjecture* is a well defined mathematical statement which (1) we believe is true but (2) don't know yet how to prove. Proving the cipher conjecture will be a great achievement and would in particular settle the P vs NP question, which is arguably *the* fundamental

⁵ As will be the case for other conjectures we talk about, the name "The Cipher Conjecture" is not a standard name, but rather one we'll use in this course. In the literature this conjecture is mostly referred to as the conjecture of existence of *one way functions*, a notion we will learn about later. These two conjectures a priori seem quite different but have been shown to be equivalent.

question of computer science. That is, the following theorem is known:

Theorem 2.5 — Breaking crypto if P=NP. If $P = NP$ then there does not exist a computationally secret encryption with efficient E and D and where the message is longer than the key.

Proof. We just sketch the proof, as this is not the focus of this course. If $P = NP$ then whenever we have a loop that searches through some domain to find some string that satisfies a particular property (like the loop in the `Distinguish` subroutine above that searches over all keys) then this loop can be sped up *exponentially*. ■

While it is very widely believed that $P \neq NP$, at the moment we do not know how to *prove* this, and so have to settle for accepting the cipher conjecture as essentially an axiom, though we will see later in this course that we can show it follows from some seemingly weaker conjectures.

There are several reasons to believe the cipher conjecture. We now briefly mention some of them:

- *Intuition:* If the cipher conjecture is false then it means that for *every* possible cipher we can make the exponential time attack described above become efficient. It seems “too good to be true” in a similar way that the assumption that $P=NP$ seems too good to be true.
- *Concrete candidates:* As we will see in the next lecture, there are several concrete candidate ciphers using keys shorter than messages for which despite *tons* of effort, no one knows how to break them. Some of them are widely used and hence governments and other benign or not so benign organizations have every reason to invest huge resources in trying to break them. Despite that as far as we know (and we know a little more after Edward Snowden’s revelations) there is no significant break known for the most popular ciphers. Moreover, there are other ciphers that can be based on canonical mathematical problems such as factoring large integers or decoding random linear codes that are immensely interesting in their own right, independently of their cryptographic applications.
- *Minimalism:* Clearly if the cipher conjecture is false then we also don’t have a secure encryption with a key, say, twice as long as the message. But it turns out the cipher conjecture is in fact *necessary* for essentially every cryptographic primitive, including not just private key and public key encryptions but also digital signatures,

hash functions, pseudorandom generators, and more. That is, if the cipher conjecture is false then to a large extent cryptography does not exist, and so we essentially have to assume this conjecture if we want to do any kind of cryptography.

2.3 Why care about the cipher conjecture?

"Give me a place to stand, and I shall move the world"
Archimedes, circa 250 BC

Every perfectly secure encryption scheme is clearly also computationally secret, and so if we required a message of size n instead $n + 1$, then the conjecture would have been trivially satisfied by the one-time pad. However, having a message longer than the key by just a single bit does not seem that impressive. Sure, if we used such a scheme with 128-bit long keys, our communication will be smaller by a factor of 128/129 (or a saving of about 0.8%) over the one-time pad, but this doesn't seem worth the risk of using an unproven conjecture. However, it turns out that if we assume this rather weak condition, we can actually get a computationally secret encryption scheme with a message of size $p(n)$ for every polynomial $p(\cdot)$. In essence, we can fix a single n -bit long key and communicate securely as many bits as we want!

Moreover, this is just the beginning. There is a huge range of other useful cryptographic tools that we can obtain from this seemingly innocent conjecture: (We will see what all these names and some of these reductions mean later in the course.)

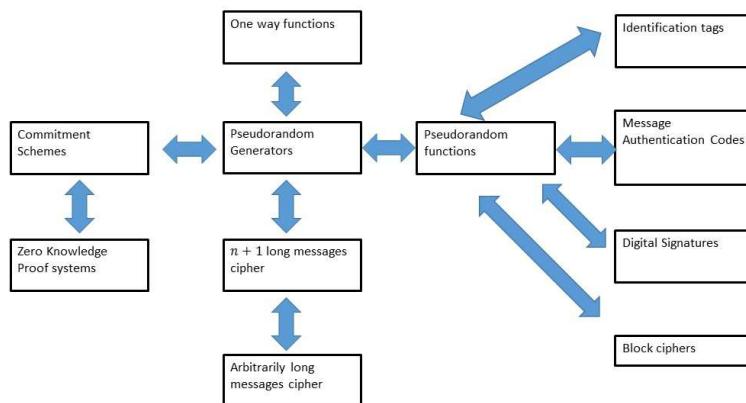


Figure 2.2: Web of reductions between notions equivalent to ciphers with larger than key messages

We will soon see the first of the many reductions we'll learn in this course. Together this "web of reductions" forms the scientific core of cryptography, connecting many of the core concepts and enabling us to construct increasingly sophisticated tools based on relatively simple "axioms" such as the cipher conjecture.

2.4 Prelude: Computational Indistinguishability

The task of Eve in breaking an encryption scheme is to *distinguish* between an encryption of m_0 and an encryption of m_1 . It turns out to be useful to consider this question of when two distributions are *computationally indistinguishable* more broadly:

Definition 2.6 — Computational Indistinguishability. Let X and Y be two distributions over $\{0,1\}^o$. We say that X and Y are (T,ϵ) -*computationally indistinguishable*, denoted by $X \approx_{T,\epsilon} Y$, if for every function Eve computable with at most T operations,

$$|\mathbb{P}[Eve(X) = 1] - \mathbb{P}[Eve(Y) = 1]| \leq \epsilon. \quad (2.2)$$

We say that X and Y are simply *computationally indistinguishable*, denoted by $X \approx Y$, if they are (T,ϵ) indistinguishable for every polynomial $T(o)$ and inverse polynomial $\epsilon(o)$.⁶

Note: The expression $\mathbb{P}[Eve(X) = 1]$ can also be written as $\mathbb{E}[Eve(X)]$ (since we can assume that whenever $Eve(x)$ does not output 1 it outputs zero). This notation will be useful for us sometimes.

We can use computational indistinguishability to phrase the definition of Computational secrecy more succinctly:

Theorem 2.7 — Computational Indistinguishability phrasing of security. Let (E,D) be a valid encryption scheme. Then (E,D) is computationally secret if and only if for every two messages $m_0, m_1 \in \{0,1\}^\ell$,

$$\{E_k(m_0)\} \approx \{E_k(m_1)\} \quad (2.3)$$

where each of these two distributions is obtained by sampling a random $k \leftarrow_R \{0,1\}^n$.

Working out the proof is an excellent way to make sure you understand both the definition of Computational secrecy and computa-

⁶ This definition implicitly assumes that X and Y are actually parameterized by some number n (that is polynomially related to o) so for every polynomial $T(o)$ and inverse polynomial $\epsilon(o)$ we can take n to be large enough so that X and Y will be (T,ϵ) indistinguishable. In all the cases we will consider, the choice of the parameter n (which is usually the length of the key) will be clear from the context.

tional indistinguishability, and hence we leave it as an exercise.

One intuition for computational indistinguishability is that it is related to some notion of *distance*. If two distributions are computationally indistinguishable, then we can think of them as “very close” to one another, at least as far as efficient observers are concerned. Intuitively, if X is close to Y and Y is close to Z then X should be close to Z .⁷ Similarly if four distributions X, X', Y, Y' satisfy that X is close to Y and X' is close to Y' , then you might expect that the distribution (X, X') where we take two independent samples from X and X' respectively, is close to the distribution (Y, Y') where we take two independent samples from Y and Y' respectively. We will now verify that these intuitions are in fact correct:

Theorem 2.8 — Triangle Inequality for Computational Indistinguishability. Suppose $\{X_1\} \approx_{T,\epsilon} \{X_2\} \approx_{T,\epsilon} \dots \approx_{T,\epsilon} \{X_m\}$. Then $\{X_1\} \approx_{T,(m-1)\epsilon} \{X_m\}$.

Proof. Suppose that there exists a T time Eve such that

$$|\mathbb{P}[Eve(X_1) = 1] - \mathbb{P}[Eve(X_m) = 1]| > (m-1)\epsilon. \quad (2.4)$$

Write

$$\mathbb{P}[Eve(X_1) = 1] - \mathbb{P}[Eve(X_m) = 1] = \sum_{i=1}^{m-1} (\mathbb{P}[Eve(X_i) = 1] - \mathbb{P}[Eve(X_{i+1}) = 1]). \quad (2.5)$$

Thus,

$$\sum_{i=1}^{m-1} |\mathbb{P}[Eve(X_i) = 1] - \mathbb{P}[Eve(X_{i+1}) = 1]| > (m-1)\epsilon \quad (2.6)$$

and hence in particular there must exist some $i \in \{1, \dots, m-1\}$ such that

$$|\mathbb{P}[Eve(X_i) = 1] - \mathbb{P}[Eve(X_{i+1}) = 1]| > \epsilon \quad (2.7)$$

contradicting the assumption that $\{X_i\} \approx_{T,\epsilon} \{X_{i+1}\}$ for all $i \in \{1, \dots, m-1\}$. ■

Theorem 2.9 — Computational Indistinguishability is preserved under repetition. Suppose that $X_1, \dots, X_\ell, Y_1, \dots, Y_\ell$ are distributions over $\{0,1\}^n$ such that $X_i \approx_{T,\epsilon} Y_i$. Then $(X_1, \dots, X_\ell) \approx_{T-10\ell n, \ell\epsilon} (Y_1, \dots, Y_\ell)$.

Proof. For every $i \in \{0, \dots, \ell\}$ we define H_i to be the distribution $(X_1, \dots, X_i, Y_{i+1}, \dots, Y_\ell)$. Clearly $H_0 = (X_1, \dots, X_\ell)$ and $H_\ell =$

⁷ Results of this form are known as “triangle inequalities” since they can be viewed as generalizations of the statement that for every three points on the plane x, y, z , the distance from x to z is not larger than the distance from x to y plus the distance from y to z . In other words, the edge $\overline{x,z}$ of the triangle (x,y,z) is not longer than the sum of the lengths of the other two edges $\overline{x,y}$ and $\overline{y,z}$.

(Y_1, \dots, Y_ℓ) . We will prove that for every i , $H_i \approx_{T-10\ell n, \epsilon} H_{i+1}$, and the proof will then follow from the triangle inequality (can you see why?). Indeed, suppose towards the sake of contradiction that there was some $i \in \{0, \dots, \ell\}$ and some $T - 10\ell n$ -time Eve's $s : \{0, 1\}^{n\ell} \rightarrow \{0, 1\}$ such that

$$|\mathbb{E}[Eve'(H_i)] - \mathbb{E}[Eve(H_{i+1})]| > \epsilon. \quad (2.8)$$

In other words

$$|\mathbb{E}_{X_1, \dots, X_{i-1}, Y_i, \dots, Y_\ell}[Eve'(X_1, \dots, X_{i-1}, Y_i, \dots, Y_\ell)] - \mathbb{E}_{X_1, \dots, X_i, Y_{i+1}, \dots, Y_\ell}[Eve'(X_1, \dots, X_i, Y_{i+1}, \dots, Y_\ell)]| > \epsilon. \quad (2.9)$$

By linearity of expectation we can write the difference of these two expectations as

$$\mathbb{E}_{X_1, \dots, X_{i-1}, X_i, Y_i, Y_{i+1}, \dots, Y_\ell} [Eve'(X_1, \dots, X_{i-1}, Y_i, Y_{i+1}, \dots, Y_\ell) - Eve'(X_1, \dots, X_{i-1}, X_i, Y_{i+1}, \dots, Y_\ell)] \quad (2.10)$$

By the *averaging principle*⁸ this means that there exist some values $x_1, \dots, x_{i-1}, y_{i+1}, \dots, y_\ell$ such that

$$|\mathbb{E}_{X_i, Y_i} [Eve'(x_1, \dots, x_{i-1}, Y_i, y_{i+1}, \dots, y_\ell) - Eve'(x_1, \dots, x_{i-1}, X_i, y_{i+1}, \dots, y_\ell)]| \quad (2.11)$$

Now X_i and Y_i are simply independent draws from the distributions X and Y respectively, and so if we define

$Eve(z) = Eve'(x_1, \dots, x_{i-1}, z, y_{i+1}, \dots, y_\ell)$ then Eve runs in time at most the running time of Eve plus $2\ell n$ and it satisfies

$$|\mathbb{E}_{X_i}[Eve(X_i)] - \mathbb{E}_{Y_i}[Eve(Y_i)]| > \epsilon \quad (2.12)$$

contradicting the assumption that $X_i \approx_{T, \epsilon} Y_i$. ■

⁸ This is the principle that if the average grade in an exam was at least α then *someone* must have gotten at least α , or in other words that if a real-valued random variable Z satisfies $\mathbb{E}Z \geq \alpha$ then $\mathbb{P}[Z \geq \alpha] > 0$.



The hybrid argument The above proof illustrates a powerful technique known as the *hybrid argument* whereby we show that two distributions C^0 and C^1 are close to each other by coming up with a sequence of distributions H_0, \dots, H_t such that $H_t = C^1$, $H_0 = C^0$, and we can argue that H_i is close to H_{i+1} for all i . This type of argument repeats itself time and again in cryptography, and so it is important to get comfortable with it.

2.5 The Length Extension Theorem

We now turn to show the *length extension theorem*, stating that if we have an encryption for $n + 1$ -length messages with n -length keys, then

we can obtain an encryption with $p(n)$ -length messages for every polynomial $p(n)$. For a warm-up, let's show that the easier fact that we can transform an encryption such as above, into one that has keys of length tn and messages of length $t(n + 1)$ for every integer t :

Theorem 2.10 — Security of repetition. Suppose that (E', D') is a computationally secret encryption scheme with n bit keys and $n + 1$ bit messages. Then the scheme (E, D) where $E_{k_1, \dots, k_t}(m_1, \dots, m_t) = (E'_{k_1}(m_1), \dots, E'_{k_t}(m_t))$ and $D_{k_1, \dots, k_t}(c_1, \dots, c_t) = (D'_{k_1}(c_1), \dots, D'_{k_t}(c_t))$ is a computationally secret scheme with tn bit keys and $t(n + 1)$ bit messages.

Proof. This might seem “obvious” but in cryptography, even obvious facts are sometimes wrong, so it's important to prove this formally. Luckily, this is a fairly straightforward implication of the fact that computational indistinguishability is preserved under many samples. That is, by the security of (E', D') we know that for every two messages $m, m' \in \{0, 1\}^{n+1}$, $E_k(m) \approx E_k(m')$ where k is chosen from the distribution U_n . Therefore by the indistinguishability of many samples lemma, for every two tuples $m_1, \dots, m_t \in \{0, 1\}^{n+1}$ and $m'_1, \dots, m'_t \in \{0, 1\}^{n+1}$,

$$(E'_{k_1}(m_1), \dots, E'_{k_t}(m_t)) \approx (E'_{k_1}(m'_1), \dots, E'_{k_t}(m'_t)) \quad (2.13)$$

for random k_1, \dots, k_t chosen independently from U_n which is exactly the condition that (E, D) is computationally secret. ■

We can now prove the full length extension theorem. Before doing so, we will need to generalize the notion of an encryption scheme to allow a *randomized encryption scheme*. That is, we will consider encryption schemes where the encryption algorithm can “toss coins” in its computation. There is a crucial difference between key material and such “as hoc” randomness. Keys need to be not only chosen at random, but also shared in advance between the sender and receiver, and stored securely throughout their lifetime. The “coin tosses” used by a randomized encryption scheme are generated “on the fly” and are not known to the receiver, nor do they need to be stored long term by the sender. So, allowing such randomized encryption does not make a difference for most applications of encryption schemes. In fact, as we will see later in this course, randomized encryption is *necessary* for security against more sophisticated attacks such as chosen plaintext and chosen ciphertext attacks, as well as for obtaining secure *public key* encryptions. We will use the notation $E_k(m; r)$ to denote the output of the encryption algorithm on key k ,

message m and using internal randomness r . We often suppress the notation for the randomness, and hence use $E_k(m)$ to denote the random variable obtained by sampling a random r and outputting $E_k(m; r)$.

We can now show that given an encryption scheme with messages one bit longer than the key, we can obtain a (randomized) encryption scheme with arbitrarily long messages:

Theorem 2.11 — Length extension of ciphers. Suppose that there exists a computationally secret encryption scheme (E', D') with key length n and message length $n + 1$. Then for every polynomial $t(n)$ there exists a (randomized) computationally secret encryption scheme (E, D) with key length n and message length $t(n)$.

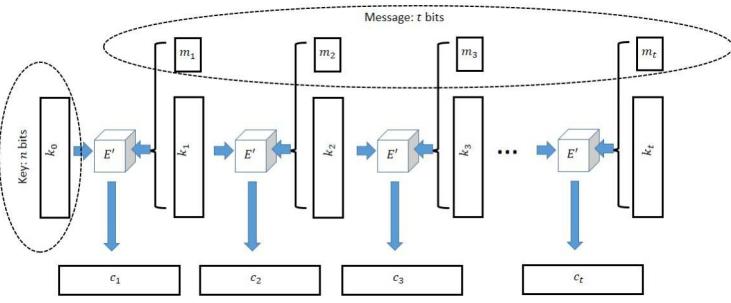


Figure 2.3: Constructing a cipher with t bit long messages from one with $n + 1$ long messages

Proof. Let $t = t(n)$. We are given a cipher E' which can encrypt $n + 1$ -bit long messages with an n -bit long key and we need to encrypt a t -bit long message $m = (m_1, \dots, m_t) \in \{0, 1\}^t$. Our idea is simple (at least in hindsight). Let $k_0 \leftarrow_R \{0, 1\}^n$ be our key (which is chosen at random). To encrypt m using k_0 , the encryption function will choose t random strings $k_1, \dots, k_t \leftarrow_R \{0, 1\}^n$. We will then encrypt the $n + 1$ -bit long message (k_1, m_1) with the key k_0 to obtain the ciphertext c_1 , then encrypt the $n + 1$ -bit long message (k_2, m_2) with the key k_1 to obtain the ciphertext c_2 , and so on and so forth until we encrypt the message (k_t, m_t) with the key k_{t-1} .⁹ We output (c_1, \dots, c_t) as the final ciphertext.¹⁰

To decrypt (c_1, \dots, c_t) using the key k_0 , first decrypt c_1 to learn (k_1, m_1) , then use k_1 to decrypt c_2 to learn (k_2, m_2) , and so on until we use k_{t-1} to decrypt c_t and learn (k_t, m_t) . Finally we can simply output (m_1, \dots, m_t) .

The above are clearly valid encryption and decryption algorithms, and hence the real question becomes *is it secure??*. The intuition

⁹ The keys k_1, \dots, k_t are sometimes known as *ephemeral keys* in the crypto literature, since they are created only for the purposes of this particular interaction.

¹⁰ The astute reader might note that the key k_t is actually not used anywhere in the encryption nor decryption and hence we could encrypt n more bits of the message instead in this final round. We used the current description for the sake of symmetry and simplicity of exposition.

is that c_1 hides all information about (k_1, m_1) and so in particular the first bit of the message is encrypted securely, and k_1 still can be treated as an unknown random string even to an adversary that saw c_1 . Thus, we can think of k_1 as a random secret key for the encryption c_2 , and hence the second bit of the message is encrypted securely, and so on and so forth.

Our discussion above looks like a reasonable intuitive argument, but to make sure it's true we need to give an actual proof. Let $m, m' \in \{0,1\}^t$ be two messages. We need to show that $E_{U_n}(m) \approx E_{U_n}(m')$. The heart of the proof will be the following claim:

Claim: Let \hat{E} be the algorithm that on input a message m and key k_0 works like E except that its the i^{th} block contains $E'_{k_{i-1}}(k'_i, m_i)$ where k'_i is a random string in $\{0,1\}^n$, that is chosen *independently* of everything else including the key k_i . Then, for every message $m \in \{0,1\}^t$

$$E_{U_n}(m) \approx \hat{E}_{U_n}(m). \quad (2.14)$$

Note that \hat{E} is not a valid encryption scheme since it's not at all clear there is a decryption algorithm for it. It is just an hypothetical tool we use for the proof. Since both E and \hat{E} are randomized encryption schemes (with E using $(t-1)n$ bits of randomness for the ephemeral keys k_1, \dots, k_{t-1} and \hat{E} using $(2t-1)n$ bits of randomness for the ephemeral keys $k_1, \dots, k_t, k'_1, \dots, k'_t$), we can also write Eq. (2.14) as

$$E_{U_n}(m; U'_{tn}) \approx \hat{E}_{U_n}(m; U'_{(2t-1)n}) \quad (2.15)$$

where we use U'_{ℓ} to denote a random variable that is chosen uniformly at random from $\{0,1\}^{\ell}$ and independently from the choice of U_n (which is chosen uniformly at random from $\{0,1\}^n$).

Once we prove the claim then we are done since we know that for every pair of message m, m' , $E_{U_n}(m) \approx \hat{E}_{U_n}(m)$ and $E_{U_n}(m') \approx \hat{E}_{U_n}(m')$ but $\hat{E}_{U_n}(m) \approx \hat{E}_{U_n}(m')$ since \hat{E} is essentially the same as the t -times repetition scheme we analyzed above. Thus by the triangle inequality we can conclude that $E_{U_n}(m) \approx E_{U_n}(m')$ as we desired.

Proof of claim: We prove the claim by the hybrid method. For $j \in \{0, \dots, \ell\}$, let H_j be the distribution of ciphertexts where in the first j blocks we act like \hat{E} and in the last $t-j$ blocks we act like E . That is, we choose $k_0, \dots, k_t, k'_1, \dots, k'_t$ independently at random from U_n and the i^{th} block of H_j is equal to $E'_{k_{i-1}}(k_i, m_i)$ if $i > j$ and is equal to $E'_{k_{i-1}}(k'_i, m_i)$ if $i \leq j$. Clearly, $H_t = \hat{E}_{U_n}(m)$ and $H_0 = E_{U_n}(m)$ and so it suffices to prove that for every j , $H_j \approx H_{j+1}$. Indeed, let

$j \in \{0, \dots, \ell\}$ and suppose towards the sake of contradiction that there exists an efficient Eve' such that

$$|\mathbb{E}[Eve'(H_j)] - \mathbb{E}[Eve'(H_{j+1})]| \geq \epsilon \quad (*)$$
(2.16)

where $\epsilon = \epsilon(n)$ is noticeable. By the averaging principle, there exists some fixed choice for $k'_1, \dots, k'_t, k_0, \dots, k_{j-2}, k_j, \dots, k_t$ such that $(*)$ still holds. Note that in this case the only randomness is the choice of $k_{j-1} \leftarrow_R U_n$ and moreover the first $j - 1$ blocks and the last $t - j$ blocks of H_j and H_{j+1} would be identical and we can denote them by α and β respectively and hence write $(*)$ as

$$\left| \mathbb{E}_{k_{j-1}}[Eve'(\alpha, E'_{k_{j-1}}(k_j, m_j), \beta) - Eve'(\alpha, E'_{k_{j-1}}(k'_j, m_j), \beta)] \right| \geq \epsilon \quad (**)$$
(2.17)

But now consider the adversary Eve that is defined as $Eve(c) = Eve'(\alpha, c, \beta)$. Then Eve is also efficient and by $(**)$ it can distinguish between $E'_{U_n}(k_j, m_j)$ and $E'_{U_n}(k'_j, m_j)$ thus contradicting the security of (E', D') . This concludes the proof of the claim and hence the theorem. ■

2.5.1 Appendix: The computational model

For concreteness sake let us give a precise definition of what it means for a function or probabilistic process f mapping $\{0, 1\}^n$ to $\{0, 1\}^m$ to be computable using T operations. This is the model of RAND programs as in my [introduction to TCS lecture notes](#), also known as the model of (probabilistic) Boolean circuits.

Definition 2.12 — Probabilistic straightline program. A *probabilistic straightline program* consists of a sequence of lines, each one of them one of the following forms:

- `foo = bar NAND baz` where `foo`, `bar`, `baz` are variable identifiers.
- `foo = RAND` where `foo` is a variable identifier.

Given a program π , we say that its *size* is the number of lines it contains. Variables beginning with x_- and y_- are considered input and output variables respectively. We require such variables to have the forms x_{-0}, \dots, x_{-n-1} for some $n > 0$ and y_{-0}, \dots, y_{-m-1} . The program computes the probabilistic process that maps $\{0, 1\}^n$ to $\{0, 1\}^m$ in the natural way. If F is a (probabilistic

or deterministic) map of $\{0,1\}^n$ to $\{0,1\}^m$, the *complexity* of F is the size of the smallest program P that computes it.

If you haven't taken a class such as CS121 before, you might wonder how such a simple model captures complicated programs that use loops, conditionals, and more complex data types than simply a bit in $\{0,1\}$, not to mention some special purpose crypto-breaking devices that might involve tailor-made hardware. It turns out that it does (for the same reason we can compile complicated programming languages to run on silicon chips with a very limited instruction set). In fact, as far as we know, this model can capture even computations that happen in nature, whether it's in a bee colony or the human brain (which contains about 10^{10} neurons, so should in principle be simulatable by a program that has up to a few order of magnitudes of the same number of lines). Crucially, for cryptography, we care about such programs not because we want to actually run them, but because we want to argue about their *non existence*.¹¹ If we have a process that cannot be computed by a straightline program of length shorter than $2^{128} > 10^{38}$ then it seems safe to say that a computer the size of the human brain (or even all the human and nonhuman brains on this planet) will not be able to perform it either.

Advanced note: The computational model we use in this class is *non uniform* (corresponding to Boolean circuits) as opposed to *uniform* (corresponding to Turing machines). If this distinction doesn't mean anything to you, you can ignore it as it won't play a significant role in what we do next. It basically means that we do allow our programs to have hardwired constants of $\text{poly}(n)$ bits where n is the input/key length. In fact, to be precise, we will hold ourselves to a higher standard than our adversary, in the sense that we require our algorithms to be efficient in the stronger sense of being computable in uniform probabilistic polynomial time (for some fixed polynomial, often $O(n)$ or $O(n^2)$), while the adversary is allowed to use non uniformity.

¹¹ An interesting potential exception to this principle that every natural process should be simulatable by a straightline program of comparable complexity are processes where the quantum mechanical notions of *interference* and *entanglement* play a significant role. We will talk about this notion of *quantum computing* towards the end of the course, though note that much of what we say does not really change when we add quantum into the picture. As discussed in [my lecture notes](#), we can still capture these processes by straightline programs (that now have somewhat more complex form), and so most of what we'll do just carries over in the same way to the quantum realm as long as we are fine with conjecturing the strong form of the cipher conjecture, namely that the cipher is infeasible to break even for quantum computers. (All current evidence points toward this strong form being true as well.)

3

Pseudorandomness

Reading: Katz-Lindell Section 3.3, Boneh-Shoup Chapter 3

The nature of randomness has troubled philosophers, scientists, statisticians and laypeople for many years.¹ Over the years people have given different answers to the question of what does it mean for data to be random, and what is the nature of probability. The movements of the planets initially looked random and arbitrary, but then the early astronomers managed to find *order* and make some *predictions* on them. Similarly we have made great advances in predicting the weather, and probably will continue to do so. So, while these days it seems as if the event of whether or not it will rain a week from today is *random*, we could imagine that in with time we will be able to predict the weather further into the future. Even the canonical notion of a random experiment -tossing a coin - turns out that it **might not be as random as you'd think**, with about a 51% chance that the second toss will have the same result as the first one. (Though **see also this experiment.**) It is conceivable that at some point someone would discover some function F that given the first 100 coin tosses by any given person can predict the value of the 101^{th} .² In all these examples, the physics underlying the event, whether it's the planets' movement, the weather, or coin tosses, did not change but only our powers to predict them. So to a large extent, *randomness is a function of the observer*, or in other words

If a quantity is hard to compute, it might as well be random.

¹ Even lawyers grapple with this question, with a recent example being the debate of whether fantasy football is a game of chance or of skill.

² In fact such a function must exist in some sense since in the entire history of the world, presumably no sequence of 100 fair coin tosses has ever repeated.

Much of cryptography is about trying to make this intuition more formal, and harnessing it to build secure systems. The basic object we want is the following:

Definition 3.1 — Pseudorandom generator. A function $G : \{0,1\}^n \rightarrow \{0,1\}^\ell$ is a (T, ϵ) pseudorandom generator if $G(U_n) \approx_{T,\epsilon} U_\ell$ where U_t denotes the uniform distribution on $\{0,1\}^t$.

We say that $G : \{0,1\}^* \rightarrow \{0,1\}^*$ is a *pseudorandom generator* with length function $\ell : \mathbb{N} \rightarrow \mathbb{N}$ (where $\ell(n) > n$) if G is computable in polynomial time, and there are functions $T(n) > n^{\omega(1)}$ and $\epsilon(n) < n^{-\omega(1)}$ such that

$$G(U_n) \approx_{T(n), \epsilon(n)} U_{\ell(n)} \quad (3.1)$$

for every $n \in \mathbb{N}$



This definition (as is often the case in cryptography) is a bit long, so you want to take your time parsing it. In particular you should verify that you understand why the condition Eq. (3.2) is the same as saying that for every polynomial $p : \mathbb{N} \rightarrow \mathbb{N}$, if n is sufficiently large, then for every circuit D of at most T gates (or equivalently, for every straightline program D of at most T lines):

$$|\mathbb{P}[D(G(U_n)) = 1] - \mathbb{P}[D(U_\ell) = 1]| < \frac{1}{p(n)} \quad (3.2)$$

Note that the requirement that $\ell > n$ is crucial to make this notion non-trivial, as for $\ell = n$ the function $G(x) = x$ clearly satisfies that $G(U_n)$ is identical to (and hence in particular indistinguishable from) the distribution U_n . (Make sure that you understand this last statement!) However, for $\ell > n$ this is no longer trivial at all, and in particular if we didn't restrict the running time of *Eve* then no such pseudo-random generator would exist:

Lemma 3.2 Suppose that $G : \{0,1\}^n \rightarrow \{0,1\}^{n+1}$. Then there exists an (inefficient) algorithm *Eve* : $\{0,1\}^{n+1} \rightarrow \{0,1\}$ such that $\mathbb{E}[\text{Eve}(G(U_n))] = 1$ but $\mathbb{E}[\text{Eve}(U_{n+1})] \leq 1/2$.

Proof. On input $y \in \{0,1\}^{n+1}$, consider the algorithm *Eve* that goes over all possible $x \in \{0,1\}^n$ and will output 1 if and only if $y = G(x)$ for some x . Clearly $\mathbb{E}[\text{Eve}(G(U_n))] = 1$. However, the set $S = \{G(x) : x \in \{0,1\}^n\}$ on which *Eve* outputs 1 has size at most 2^n , and hence a random $y \leftarrow_R U_{n+1}$ will fall in S with probability at most $1/2$. ■

It is not hard to show that if $P = NP$ then the above algorithm *Eve* can be made efficient. In particular, at the moment we do not know

how to *prove* the existence of pseudorandom generators. Nevertheless they are widely believed to exist and hence we make the following conjecture:

Conjecture (The PRG conjecture): For every n , there exists a pseudorandom generator G mapping n bits to $n + 1$ bits.³

As was the case for the cipher conjecture, and any other conjecture, there are two natural questions regarding the PRG conjecture: why should we believe it and why should we care. Fortunately, the answer to the first question is simple: it is known that the cipher conjecture *implies* the PRG conjecture, and hence if we believe the former we should believe the latter. (The proof is highly non-trivial and we may not get to see it in this course.) As for the second question, we will see that the PRG conjecture implies a great number of useful cryptographic tools, including the cipher conjecture (i.e., the two conjectures are in fact equivalent). We start by showing that once we can get to an output that is one bit longer than the input, we can in fact obtain any number of bits.

Theorem 3.3 — Length Extension for PRG's. Suppose that the PRG conjecture is true. Then for every polynomial $t(n)$, there exists a pseudorandom generator mapping n bits to $t(n)$ bits.

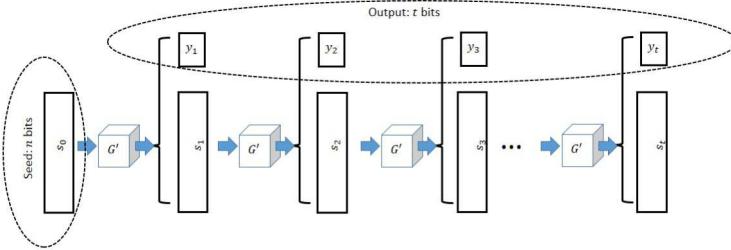


Figure 3.1: Length extension for pseudorandom generators

Proof. The proof of this theorem is very similar to the length extension theorem for ciphers, and in fact this theorem can be used to give an alternative proof for the former theorem.

The construction is illustrated in Fig. 3.1. We are given a pseudorandom generator G' mapping n bits into $n + 1$ bits and need to construct a pseudorandom generator G mapping n bits to $t = t(n)$ bits for some polynomial $t(\cdot)$. The idea is that we maintain a state of n bits, which are originally our input seed⁴ s_0 , and at the i^{th} step we

³ The name “The PRG conjecture” is non-standard. In the literature this is known as the conjecture of existence of pseudorandom generators. This is a weaker form of “The Optimal PRG Conjecture” presented in my [intro to theoretical CS lecture notes](#) since the PRG conjecture only posits the existence of pseudorandom generators with arbitrary polynomial blowup, as opposed to an exponential blowup posited in the optimal PRF conjecture.

⁴ Because we use a small input to grow a large pseudorandom string, the input to a pseudorandom generator is often known as its *seed*.

use G' to map s_{i-1} to the $n+1$ -long bit string (s_i, y_i) , output y_i and keep s_i as our new state. To prove the security of this construction we need to show that the distribution $G(U_n) = (y_1, \dots, y_t)$ is computationally indistinguishable from the uniform distribution U_t . As usual, we will use the hybrid argument. For $i \in \{0, \dots, t\}$ we define H_i to be the distribution where the first i bits chosen at uniform, whereas the last $t-i$ bits are computed as above. Namely, we choose s_i at random in $\{0, 1\}^n$ and continue the computation of y_{i+1}, \dots, y_t from the state s_i . Clearly $H_0 = G(U_n)$ and $H_t = U_t$ and hence by the triangle inequality it suffices to prove that $H_i \approx H_{i+1}$ for all $i \in \{0, \dots, t-1\}$. We illustrate these two hybrids in Fig. 3.2.

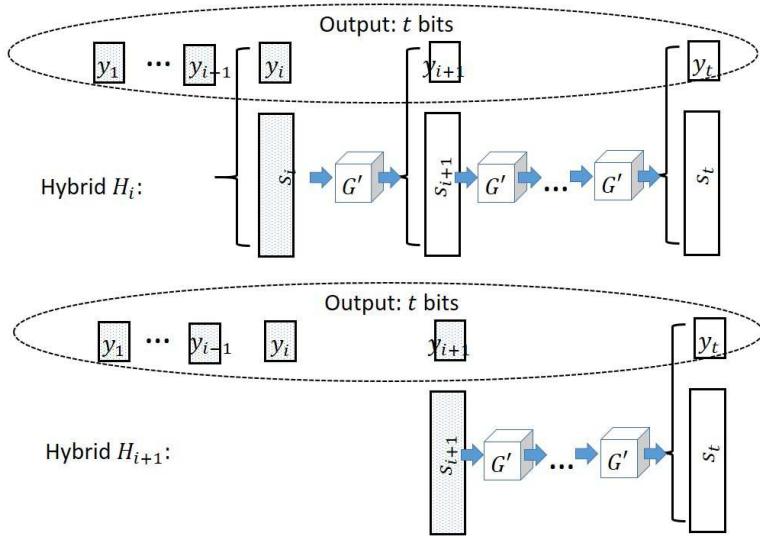


Figure 3.2: Hybrids H_i and H_{i+1} —dotted boxes refer to values that are chosen independently and uniformly at random

Now suppose otherwise, that there exists some adversary Eve such that $|\mathbb{E}[Eve(H_i)] - \mathbb{E}[Eve(H_{i+1})]| \geq \epsilon$ for some non-negligible ϵ . We will build from Eve an adversary Eve' breaking the security of the pseudorandom generator G' (see Fig. 3.3).

On input of string y of length $n+1$, Eve' will interpret y as (s_{i+1}, y_{i+1}) , choose y_1, \dots, y_i randomly and compute y_{i+2}, \dots, y_t as in our pseudorandom generator's construction. Eve' will then feed (y_1, \dots, y_t) to Eve and output whatever Eve does. Clearly, Eve' is efficient if Eve is. Moreover, one can see that if y was random then Eve' is feeding Eve with an input distributed according to H_{i+1} while if y was of the form $G(s)$ for a random s then Eve' will feed Eve with an input distributed according to H_i . Hence we get that $|\mathbb{E}[Eve'(G'(U_n))] - \mathbb{E}[Eve'(U_{n+1})]| \geq \epsilon$ contradicting the security of G' . \blacksquare

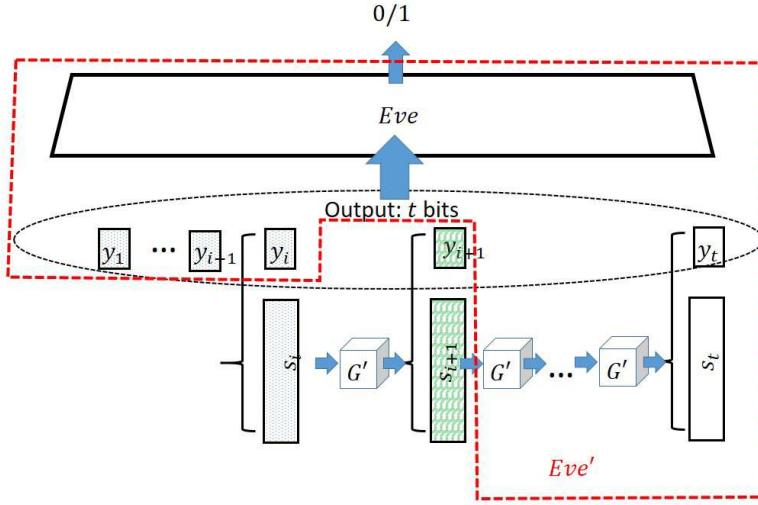


Figure 3.3: Building an adversary Eve' for G' from an adversary Eve distinguishing H_i and H_{i+1} . The boxes marked with question marks are those that are random or pseudorandom depending on whether we are in H_i or H_{i+1} . Everything inside the dashed red lines is simulated by Eve' that gets as input the $n + 1$ -bit string (s_{i+1}, y_{i+1}) .

The proof of [Theorem 3.3](#) is indicative of many practical constructions of pseudorandom generators. Many operating systems keep track of an initial *seed* of randomness, and supply a system call `rand` such that every call to `rand` applies a pseudorandom generator G' to the current seed, uses part of the output to update the seed, and returns the remainder to the caller.

R Unpredictability and indistinguishability- an alternative approach for proving the length extension theorem

The notion that being random is the same as being “unpredictable” can be formalized as follows. One can show that a random variable X over $\{0,1\}^n$ is pseudorandom if and only every efficient algorithm A succeeds in the following experiment with probability at most $1/2 + negl(n)$: A is given i chosen at random in $\{0,\dots,n-1\}$ and x_1, \dots, x_i where (x_1, \dots, x_n) is drawn from X and wins if it outputs x_{i+1} . It is a good optional exercise to prove this, and to use that to give an alternative proof of the length extension theorem.

3.1 Stream ciphers

We now show a connection between our two notions:

Theorem 3.4 — PRG conjecture implies Cipher conjectures. If the PRG conjecture is true then so is the cipher conjecture.

It turns out that the converse direction is also true, and hence these two conjectures are *equivalent*, though we will probably not show the (quite non-trivial) proof of this fact in this course. (We might show some weaker version of this harder direction.)

Proof. The construction is actually quite simple, recall that the *one time pad* is a perfectly secure cipher but its only problem was that to encrypt an $n + 1$ long message it needed an $n + 1$ long bit key. Now using a pseudorandom generator, we can map an n -bit long key into an $n + 1$ -bit long string that looks random enough that we could use it as a key for the one-time pad. That is, our cipher will look as follows:

$$E_k(m) = G(k) \oplus m \quad (3.3)$$

and

$$D_k(c) = G(k) \oplus c \quad (3.4)$$

Just like in the one time pad, $D_k(E_k(m)) = G(k) \oplus G(k) \oplus m = m$. Moreover, the encryption and decryption algorithms are clearly efficient and so the only thing that's left is to prove security or that for every pair m, m' of plaintexts, $E_{U_n}(m) \approx E_{U_n}(m')$. We show this by proving the following claim:

Claim: For every $m \in \{0, 1\}^{n+1}$, $E_{U_n}(m) \approx U_{n+1} \oplus m$.

The claim implies the security of the scheme, since it means that $E_{U_n}(m)$ is indistinguishable from the one-time-pad encryption of m , which is identically distributed to the one-time pad encryption of m' which (by another application of the claim) is indistinguishable from $E_{U_n}(m')$ and so the theorem follows from the triangle inequality. Thus all that's left is to prove the claim:

Proof of claim: Suppose that there was an efficient adversary Eve' such that

$$|\mathbb{E}[Eve'(G(U_n) \oplus m)] - \mathbb{E}[Eve'(U_{n+1} \oplus m)]| \geq \epsilon \quad (3.5)$$

for some non-negligible $\epsilon = \epsilon(n) > 0$. Then the adversary Eve defined as $Eve(y) = Eve'(y \oplus m)$ would be also efficient and would

break the security of the PRG with non-negligible success. This concludes the proof of the claim and hence of the theorem. ■

If the PRG outputs $t(n)$ bits instead of $n + 1$ then we automatically get an encryption scheme with $t(n)$ long message length. In fact, in practice if we use the length extension for PRG's, we don't need to decide on the length of messages in advance. Every time we need to encrypt another bit (or another block) m_i of the message, we run the basic PRG to update our state and obtain some new randomness y_i that we can XOR with the message and output. Such constructions are known as *stream ciphers* in the literature. In much of the practical literature the name *stream cipher* is used both for the pseudorandom generator itself, as well as for the encryption scheme that is obtained by combining it with the one-time pad.

R

Using pseudorandom generators for coin tossing

over the phone The following is a cute application of pseudorandom generators. Alice and Bob want to toss a fair coin over the phone. They use a pseudorandom generator $G : \{0,1\}^b \rightarrow \{0,1\}^{3n}$.

- Alice will send $z \leftarrow_R \{0,1\}^{3n}$ to Bob
- Bob picks $s \leftarrow_R \{0,1\}^n$ and with probability $1/2$ sends $G(s)$ (case I) and with probability $1/2$ sends $G(s) \oplus z$ (case II).
- Alice then picks a random $b \leftarrow_R \{0,1\}$ and sends it to Bob.
- Bob reveals what he sent in the previous stage and if it was case I, their output is b , and if it was case II, their output is $1 - b$.

It can be shown that (assuming the protocol is completed) the output is a random coin, which neither Alice or Bob can control or predict with more than negligible advantage over half. (Trying to formalize this and prove it is an excellent exercise.)

3.2 What do pseudorandom generators actually look like?

So far we have made the conjecture that objects such as ciphers and pseudorandom generators *exist*, without giving any hint as to how they would actually look like. (Though we have examples such as Ceasar cipher, Vignere, and Enigma of what secure ciphers *don't* look like.) As mentioned above, we do not know how to *prove* that any particular function is a pseudorandom generator. However, there are quite simple *candidates* (i.e., functions that are conjectured to be secure pseudorandom generators), though care must be taken in

constructing them. We now consider candidates for functions that maps n bits to $n + 1$ bits (or more generally $n + c$ for some constant c) and look at least somewhat “randomish”. As these constructions are typically used as a basic component for obtaining a longer length PRG via the length extension theorem (Theorem 3.3), we will think of these pseudorandom generators as mapping a string $s \in \{0,1\}^n$ representing the current state into a string $s' \in \{0,1\}^n$ representing the new state as well as a string $b \in \{0,1\}^c$ representing the current output. See also Section 6.1 in Katz-Lindell and (for greater depth) Sections 3.6-3.9 in the Boneh-Shoup book.

3.2.1 Attempt 0: The counter generator

Just to get started, let’s show an example of an obviously bogus pseudorandom generator. We define the “counter pseudorandom generator” $G : \{0,1\}^n \rightarrow \{0,1\}^{n+1}$ as follows. $G(s) = (s', b)$ where $s' = s + 1 \bmod 2^n$ (treating s and s' as numbers in $\{0, \dots, 2^n - 1\}$) and b is the least significant digit of s' . It’s a great exercise to work out why this is *not* a secure pseudorandom generator.



You should really pause here and make sure you see why the “counter pseudorandom generator” is not a secure pseudorandom generator. Show that this is true even if we replace the least significant digit by the k -th digit for every $0 \leq k < n$.

3.2.2 Attempt 1: The linear checksum / linear feedback shift register (LFSR)

LFSR can be thought of as the “mother” (or maybe more like the sick great-uncle) of all psuedorandom generators. One of the simplest ways to generate a “randomish” extra digit given an n digit number is to use a *checksum* - some linear combination of the digits, with a canonical example being the **cyclic redundancy check** or CRC.⁵ This motivates the notion of a *linear feedback shift register generator* (LFSR): if the current state is $s \in \{0,1\}^n$ then the output is $f(s)$ where f is a linear function (modulo 2) and the new state is obtained by right shifting the previous state and putting $f(s)$ at the leftmost location. That is, $s'_1 = f(s)$ and $s'_i = s_{i-1}$ for $i \in \{2, \dots, n\}$.

LFSR’s have several good properties- if the function $f(\cdot)$ is chosen properly then they can have very long *periods* (i.e., it can take an

⁵ CRC are often used to generate a “control digit” to detect mistypes of credit card or social security card number. This has very different goals than its use for pseudorandom generators, though there are some common intuitions behind the two usages.

exponential number of steps until the state repeats itself), though that also holds for the simple “counter” generator we saw above. They also have the property that every individual bit is equal to 0 or 1 with probability exactly half (the counter generator also shares this property).

A more interesting property is that (if the function is selected properly) every two coordinates are independent from one another. That is, there is some super-polynomial function $t(n)$ (in fact $t(n)$ can be exponential in n) such that if $\ell \neq \ell' \in \{0, \dots, t(n)\}$, then if we look at the two random variables corresponding to the ℓ -th and ℓ' -th output of the generator (where randomness is the initial state) then they are distributed like two independent random coins. (This is non-trivial to show, and depends on the choice of f - it is a challenging but useful exercise to work this out.) The counter generator fails badly at this condition: the least significant bits between two consecutive states always flip.

There is a more general notion of a *linear generator* where the new state can be any invertible linear transformation of the previous state. That is, we interpret the state s as an element of \mathbb{Z}_q^t for some integers q, t ,⁶ and let $s' = F(s)$ and the output $b = G(s)$ where $F : \mathbb{Z}_q^t \rightarrow \mathbb{Z}_q^t$ and $G : \mathbb{Z}_q^t \rightarrow \mathbb{Z}_q$ are invertible linear transformations (modulo q). This includes as a special case the *linear congruential generator* where $t = 1$ and the map $F(s)$ corresponds to taking $as \pmod{q}$ where a is number co-prime to q .

All these generators are unfortunately insecure due to the great bane of cryptography- the *Gaussian elimination algorithm* which students typically encounter in any linear algebra class.⁷

Theorem 3.5 — The unfortunate theorem for cryptography. There is a polynomial time algorithm to solve m linear equations in n variables (or to certify no solution exists) over any ring.

Despite its seeming simplicity and ubiquity, Gaussian elimination (and some generalizations and related algorithms such as Euclid’s extended g.c.d algorithm and the LLL lattice reduction algorithm) has been used time and again to break candidate cryptographic constructions. In particular, if we look at the first n outputs of a linear generator b_1, \dots, b_n then we can write linear equations in the unknown initial state of the form $f_1(s) = b_1, \dots, f_n(s) = b_n$ where the f_i ’s are known linear functions. Either those functions are *linearly independent*, in which case we can solve the equations to get the unique solution for the original state s and from which point

⁶ A ring is a set of elements where addition and multiplication are defined and obey the natural rules of associativity and commutativity (though without necessarily having a multiplicative inverse for every element). For every integer q we define \mathbb{Z}_q (known as the *ring of integers modulo q*) to be the set $\{0, \dots, q-1\}$ where addition and multiplication is done modulo q .

⁷ Despite the name, the algorithm goes at least as far back as the Chinese *Jiuzhang Suanshu* manuscript, circa 150 B.C.

we can predict all outputs of the generator, or they are dependent, which means that we can predict some of the outputs even without recovering the original state. Either way the generator is *#!'ed (where *#! refers to whatever verb you prefer to use when your system is broken). See also this [1977 paper](#) of James Reed.



Non-cryptographic PRGs The above means that it is a bad idea to use a linear checksum as a pseudorandom generator in a cryptographic application, and in fact in any adversarial setting (e.g., one shouldn't hope that an attacker would not be able to reverse engineer the algorithm⁸ that computes the control digit of a credit card number). However, that does not mean that there are no legitimate cases where linear generators can be used. In a setting where the application is not adversarial and you have an ability to *test* if the generator is actually successful, it might be reasonable to use such insecure non-cryptographic generators. They tend to be more efficient (though often not by much) and hence are often the default option in many programming environments such as the C `rand()` command. (In fact, the real bottleneck in using cryptographic pseudorandom generators is often the generation of *entropy* for their seed, as discussed in the previous lecture, and not their actual running time.)

3.2.3 From insecurity to security

It is often the case that we want to “fix” a broken cryptographic primitive, such as a pseudorandom generator, to make it secure. At the moment this is still more of an art than a science, but there are some principles that cryptographers have used to try to make this more principled. The main intuition is that there are certain properties of computational problems that make them more amenable to algorithms (i.e., “easier”) and when we want to make the problems useful for cryptography (i.e., “hard”) we often seek variants that don’t possess these properties. The following table illustrates some examples of such properties. (These are not formal statements, but rather is intended to give some intuition)

Many cryptographic constructions can be thought of as trying to transform an easy problem into a hard one by moving from the left to the right column of this table.

The **discrete logarithm problem** is the discrete version of the continuous real logarithm problem. The **learning with errors problem**

⁸ That number is obtained by applying an algorithm of [Hans Peter Luhn](#) which applies a simple map to each digit of the card and then sums them up modulo 10.

Easy	Hard
Continuous	Discrete
Convex	Non-convex
Linear	Non-linear (degree ≥ 2)
Noiseless	Noisy
Local	Global
Shallow	Deep
Low degree	High degree

can be thought of as the noisy version of the linear equations problem (or the discrete version of least squares minimization). When constructing **block ciphers** we often have *mixing* transformation to ensure that the dependency structure between different bits is *global*, *S-boxes* to ensure *non-linearity*, and many *rounds* to ensure *deep structure* and *large algebraic degree*.

This also works in the other direction. Many algorithmic and machine learning advances work by embedding a discrete problem in a continuous convex one. Some attacks on cryptographic objects can be thought of as trying to recover some of the structure (e.g., by embedding modular arithmetic in the real line or “linearizing” non linear equations).

3.2.4 Attempt 2: Linear Congruential Generators with dropped bits

One approach that is widely used in implementations of pseudo-random generators is to take a linear generator such as the linear congruential generators described above, and use for the output a “chopped” version of the linear function and drop some of the least significant bits. The operation of dropping these bits is non-linear and hence the attack above does not immediately apply. Nevertheless, it turns out this attack can be generalized to handle this case, and hence even with dropped bits Linear Congruential Generators are completely insecure and should be used (if at all) only in applications such as simulations where there is no adversary. Section 3.7.1 in the Boneh-Shoup book describes one attack against such generators that uses the notion of *lattice algorithms* that we will encounter later in this course in very different contexts.

3.3 Successful examples

Let's now describe some *successful* (at least per current knowledge) pseudorandom generators:

3.3.1 Case Study 1: Subset Sum Generator

Here is an extremely simple generator that is yet still secure⁹ as far as we know.

```
# seed is a list of 40 zero/one values
# output is a 48 bit integer
def subset_sum_gen(seed):
    modulo = 0x1000000
    constants = [
        0x3D6EA1, 0x1E2795, 0xC802C6, 0xBF742A, 0x45FF31,
        0x53A9D4, 0x927F9F, 0x70E09D, 0x56F00A, 0x78B494,
        0x9122E7, 0xAFB10C, 0x18C2C8, 0x8FF050, 0x0239A3,
        0x02E4E0, 0x779B76, 0x1C4FC2, 0x7C5150, 0x81E05E,
        0x154647, 0xB80E68, 0xA042E5, 0xE20269, 0xD3B7F3,
        0xCC5FB9, 0x0BFC55, 0x847AE0, 0x8CFDF8, 0xE304B7,
        0x869ACE, 0xB4CDAB, 0xC8E31F, 0x00EDC7, 0xC50541,
        0xD6DDD, 0x695A2F, 0xA81062, 0x0123CA, 0xC6C5C3]

    # return the modular sum of the constants
    # corresponding to ones in the seed
    return reduce(lambda x,y: (x+y) % modulo,
                  map(lambda a,b: a*b, constants, seed))
```

The seed to this generator is an array `seed` of 40 bits, with 40 hard-wired constants each 48 bits long (these constants were generated at random, but are fixed once and for all, and are not kept secret and hence are not considered part of the secret random seed). The output is simply

$$\sum_{i=1}^{40} \text{seed}[i] \text{constants}[i] \pmod{2^{48}} \quad (3.6)$$

and hence expands the 40 bit input into a 48 bit output.

This generator is loosely motivated by the “subset sum” computational problem, which is NP hard. However, since NP hardness is a *worst case* notion of complexity, it does not imply security for pseudorandom generators, which requires hardness of an *average case* variant. To get some intuition for its security, we can work out why

⁹ Actually modern computers will be able to break this generator via brute force, but if the length and number of the constants were doubled (or perhaps quadrupled) this should be sufficiently secure, though longer to write down.

(given that it seems to be linear) we cannot break it by simply using Gaussian elimination.



This is an excellent point for you to stop and try to answer this question on your own.

Given the known constants and known output, figuring out the set of potential seeds can be thought of as solving a *single* equation in 40 variables. However, this equation is clearly overdetermined, and will have a solution regardless of whether the observed value is indeed an output of the generator, or it is chosen uniformly at random.

More concretely, we can use linear-equation solving to compute (given the known constants $c_1, \dots, c_{40} \in \mathbb{Z}_{2^{48}}$ and the output $y \in \mathbb{Z}_{2^{48}}$) the linear subspace V of all vectors $(s_1, \dots, s_{40}) \in (\mathbb{Z}_{2^{48}})^{40}$ such that $\sum s_i c_i = y \pmod{2^{48}}$. But, regardless of whether y was generated at random from $\mathbb{Z}_{2^{48}}$, or y was generated as an output of the generator, the subspace V will always have the same dimension (specifically, since it is formed by a single linear equation over 40 variables, the dimension will be 39.) To break the generator we seem to need to be able to decide whether this linear subspace $V \subseteq (\mathbb{Z}_{2^{48}})^{40}$ contains a *Boolean vector* (i.e., a vector $s \in \{0, 1\}^n$). Since the condition that a vector is Boolean is not defined by linear equations, we cannot use Gaussian elimination to break the generator. Generally, the task of finding a vector with *small* coefficients inside a discrete linear subspace is closely related to a classical problem known as finding the **shortest vector in a lattice**. (See also the **short integer solution (SIS)** problem.)

3.3.2 Case Study 2: RC4

The following is another example of an extremely simple generator known as RC4 (this stands for Rivest Cipher 4, as Ron Rivest invented this in 1987) and is still fairly widely used today.

```
def RC4(P, i, j):
    i = (i + 1) % 256
    j = (j + P[i]) % 256
    P[i], P[j] = P[j], P[i]
    return (P, i, j, P[(P[i]+P[j]) % 256])
```

The function RC4 takes as input the current state P, i, j of the generator and returns the new state together with a single output byte. The state of the generator consists of an array P of 256 bytes,

which can be thought of as a *permutation* of the numbers $0, \dots, 255$ in the sense that we maintain the invariant that $P[i] \neq P[j]$ for every $i \neq j$, and two indices $i, j \in \{0, \dots, 255\}$. We can consider the initial state as the case where P is a completely random permutation and i and j are initialized to zero, although to save on initial seed size, typically RC4 uses some “pseudorandom” way to generate P from a shorter seed as well.

RC4 has extremely efficient software implementations and hence has been widely implemented. However, it has several issues with its security. In particular it was shown by Mantin¹⁰ and Shamir that the second bit of RC4 is *not* random, even if the initialization vector was random. This and other issues led to a practical attack on the 802.11b WiFi protocol, see Section 9.9 in Boneh-Shoup. The initial response to those attacks was to suggest to drop the first 1024 bytes of the output, but by now the attacks have been sufficiently extended that RC4 is simply not considered a secure cipher anymore. The ciphers Salsa and ChaCha, designed by Dan Bernstein, have a similar design to RC4, and are considered secure and deployed in several standard protocols such as TLS, SSH and QUIC, see Section 3.6 in Boneh-Shoup.

3.4 Non-constructive existence of pseudorandom generators

We now show that, if we don’t insist on *constructivity* of pseudorandom generators, then we can show that there exists pseudorandom generators with output that *exponentially larger* in the input length.

Lemma 3.6 — Existence of inefficient pseudorandom generators. There is some absolute constant C such that for every ϵ, T , if $\ell > C(\log T + \log(1/\epsilon))$ and $m \leq T$, then there is an (T, ϵ) pseudorandom generator $G : \{0, 1\}^\ell \rightarrow \{0, 1\}^m$.

Proof Idea: The proof uses an extremely useful technique known as the “probabilistic method” which is not too hard mathematically but can be confusing at first.¹¹ The idea is to give a “non constructive” proof of existence of the pseudorandom generator G by showing that if G was chosen at random, then the probability that it would be a valid (T, ϵ) pseudorandom generator is positive. In particular this means that there *exists* a single G that is a valid (T, ϵ) pseudorandom generator. The probabilistic method is just a *proof technique* to demonstrate the existence of such a function. Ultimately, our goal is to show the existence of a *deterministic* function G that satisfies

¹⁰ I typically do not include references in these lecture notes, and leave them to the texts, but I make here an exception because Itsik Mantin was a close friend of mine in grad school.

¹¹ There is a whole (highly recommended) book by Alon and Spencer devoted to this method.

The above discussion might be rather abstract at this point, but would become clearer after seeing the proof.

Proof of Lemma 3.6. Let ϵ, T, ℓ, m be as in the lemma's statement. We need to show that there exists a function $G : \{0,1\}^\ell \rightarrow \{0,1\}^m$ that “fools” every T line program P in the sense of Eq. (3.2). We will show that this follows from the following claim:

Claim I: For every fixed NAND program / Boolean circuit P , if we pick $G : \{0,1\}^\ell \rightarrow \{0,1\}^m$ at random then the probability that Eq. (3.2) is violated is at most 2^{-T^2} .

Before proving Claim I, let us see why it implies Lemma 3.6. We can identify a function $G : \{0,1\}^\ell \rightarrow \{0,1\}^m$ with its “truth table” or simply the list of evaluations on all its possible 2^ℓ inputs. Since each output is an m bit string, we can also think of G as a string in $\{0,1\}^{m \cdot 2^\ell}$. We define \mathcal{G}_ℓ^m to be the set of all functions from $\{0,1\}^\ell$ to $\{0,1\}^m$. As discussed above we can identify \mathcal{F}_ℓ^m with $\{0,1\}^{m \cdot 2^\ell}$ and choosing a random function $G \sim \mathcal{F}_\ell^m$ corresponds to choosing a random $m \cdot 2^\ell$ -long bit string.

For every NAND program / Boolean circuit P let B_P be the event that, if we choose G at random from \mathcal{F}_ℓ^m then Eq. (3.2) is violated with respect to the program P . It is important to understand what is the sample space that the event B_P is defined over, namely this event depends on the choice of G and so B_P is a subset of \mathcal{F}_ℓ^m . An equivalent way to define the event B_P is that it is the subset of all functions mapping $\{0,1\}^\ell$ to $\{0,1\}^m$ that violate Eq. (3.2), or in other words:

$$B_P = \left\{ G \in \mathcal{F}_\ell^m \mid \left| \frac{1}{2^\ell} \sum_{s \in \{0,1\}^\ell} P(G(s)) - \frac{1}{2^m} \sum_{r \in \{0,1\}^m} P(r) \right| > \epsilon \right\}. \quad (3.7)$$

(We've replaced here the probability statements in Eq. (3.2) with the equivalent sums so as to reduce confusion as to what is the sample space that B_P is defined over.)

To understand this proof it is crucial that you pause here and see how the definition of B_P above corresponds to Eq. (3.7). This may well take re-reading the above text once or twice, but it is a good exercise at parsing probabilistic statements and learning how to identify the *sample space* that these statements correspond to.

Now, the number of programs of size T (or circuits of size T) is at most $2^{O(T \log T)}$. Since $T \log T = o(T^2)$ this means that if Claim I is true, then by the union bound it holds that the probability of the union of B_P over all NAND programs of at most T lines is at most

$2^{O(T \log T)} 2^{-T^2} < 0.1$ for sufficiently large T . What is important for us about the number 0.1 is that it is smaller than 1. In particular this means that there exists a single $G^* \in \mathcal{F}_\ell^m$ such that G^* does not violate Eq. (3.2) with respect to any NAND program of at most T lines, but that precisely means that G^* is a (T, ϵ) pseudorandom generator.

Hence conclude the proof of Lemma 3.6, it suffices to prove Claim I. Choosing a random $G : \{0, 1\}^\ell \rightarrow \{0, 1\}^m$ amounts to choosing $L = 2^\ell$ random strings $y_0, \dots, y_{L-1} \in \{0, 1\}^m$ and letting $G(x) = y_x$ (identifying $\{0, 1\}^\ell$ and $[L]$ via the binary representation). Hence the claim amounts to showing that for every fixed function $P : \{0, 1\}^m \rightarrow \{0, 1\}$, if $L > 2^{C(\log T + \log \epsilon)}$ (which by setting $C > 4$, we can ensure is larger than $10T^2/\epsilon^2$) then the probability that

$$\left| \frac{1}{L} \sum_{i=0}^{L-1} P(y_i) - \mathbb{P}_{s \sim \{0, 1\}^m}[P(s) = 1] \right| > \epsilon \quad (3.8)$$

is at most 2^{-T^2} . Eq. (3.8) follows directly from the Chernoff bound.

If we let for every $i \in [L]$ the random variable X_i denote $P(y_i)$, then since y_0, \dots, y_{L-1} is chosen independently at random, these are independently and identically distributed random variables with mean $\mathbb{E}_{y \sim \{0, 1\}^m}[P(y)] = \mathbb{P}_{y \sim \{0, 1\}^m}[P(y) = 1]$ and hence the probability that they deviate from their expectation by ϵ is at most $2 \cdot 2^{-\epsilon^2 L / 2}$. ■

4

Pseudorandom functions

In the last lecture we saw the notion of *pseudorandom generators*, and introduced the **PRG conjecture** that there exists a pseudorandom generator mapping n bits to $n + 1$ bits. We have seen the *length extension* theorem that when given such a pseudorandom generator, we can create a generator mapping n bits to m bits for an arbitrarily large polynomial $m(n)$. But can we extend it even further? Say, to 2^n bits? Does this question even make sense? And why would we want to do that? This is the topic of this lecture.

At a first look, the notion of extending the output length of a pseudorandom generator to 2^n bits seems nonsensical. After all we want our generator to be *efficient* and just writing down the output will take exponential time. However, there is a way around this conundrum. While we can't efficiently write down the full output, we can require that it would be possible, given an index $i \in \{0, \dots, 2^n - 1\}$, to compute the i^{th} bit of the output in polynomial time.¹ That is, we require that the function $i \mapsto G(S)_i$ is efficiently computable and (by security of the pseudorandom generator) indistinguishable from a function that maps each index i to an independent random bit in $\{0, 1\}$. This is the notion of a *pseudorandom function generator* which is a bit subtle to define and construct, but turns out to have great many applications in cryptography.

Definition 4.1 — Pseudorandom Function Generator. An efficiently computable function F taking two inputs $s \in \{0, 1\}^n$ and $i \in \{0, \dots, 2^n - 1\}$ and outputting a single bit $F(s, i)$ is a *pseudorandom function (PRF) generator* if for every polynomial time adversary A outputting a single bit and polynomial $p(n)$, if n is

¹ In this course we will often index strings and numbers starting from zero rather than one, and so typically index the coordinates of a string $y \in \{0, 1\}^N$ as $0, \dots, N - 1$ rather than $1, \dots, N$. But we will not be religious about it and occasionally “lapse” into one-based indexing. In almost all cases, this makes no difference.

large enough then:

$$\left| \mathbb{E}_{s \in \{0,1\}^n} [A^{F(s,\cdot)}(1^n)] - \mathbb{E}_{H \leftarrow_R [2^n] \rightarrow \{0,1\}} [A^H(1^n)] \right| < 1/p(n) . \quad (4.1)$$

Some notes on notation are in order. The input 1^n is simply a string of n ones, and it is a typical cryptography convention to assume that such an input is always given to the adversary. This is simply because by “polynomial time” we really mean polynomial in n (which is our key size or security parameter). The notation $A^{F(s,\cdot)}$ means that A has *black box* (also known as *oracle*) access to the function that maps i to $F(s,i)$. That is, A can choose an index i , query the box and get $F(s,i)$, then choose a new index i' , query the box to get $F(s,i')$, and so on and so forth continuing for a polynomial number of queries. The notation $H \leftarrow_R \{0,1\}^n \rightarrow \{0,1\}$ means that H is a completely random function that maps every index i to an independent and random different bit. That means that the notation A^H in the equation above means that A has access to a completely random black box that returns a random bit for any new query made. Finally one last note: below we will identify the set $[2^n] = \{0, \dots, 2^n - 1\}$ with the set $\{0,1\}^n$ (there is a one to one mapping between those sets using the binary representation), and so we will treat i interchangeably as a number in $[2^n]$ or a string in $\{0,1\}^n$.

Informally, if F is a pseudorandom function generator, then if we choose a random string s , and consider the function f_s defined by $f_s(i) = F(s,i)$ then no efficient algorithm can distinguish between black box access to $f_s(\cdot)$ and black box access to a completely random function (see Fig. 4.1). Thus often instead of talking about a pseudorandom function generator we will refer to a *pseudorandom function collection* $\{f_s\}$ where by that we mean that the map $F(s,i) = f_s(i)$ is a pseudorandom function generator.

In the next lecture we will see the proof of following theorem (due to Goldreich, Goldwasser, and Micali)

Theorem 4.2 — PRFs from PRGs. Assuming the PRG conjecture, there exists a secure pseudorandom function generator.

But before we see the proof of Theorem 4.2, let us see why pseudorandom functions could be useful.

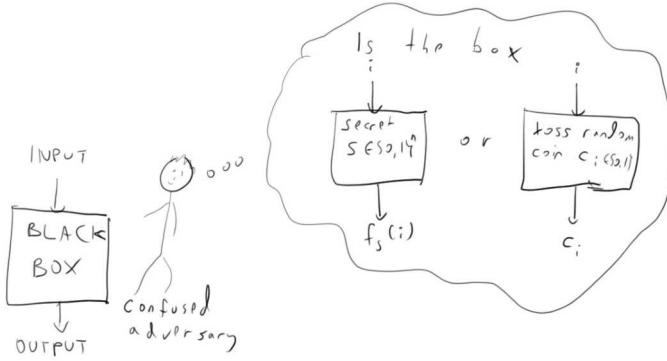


Figure 4.1: In a pseudorandom function, an adversary cannot tell whether they are given a black box that computes the function $i \mapsto F(s, i)$ for some secret s that was chosen at random and fixed, or whether the black box computes a completely random function that tosses a fresh random coin whenever it's given a new input i

4.1 One time passwords (e.g., Google Authenticator, RSA ID, etc..)

Until now we have talked about the task of *encryption*, or protecting the *secrecy* of messages. But the task of *authentication*, or protecting the *integrity* of messages is no less important. For example, consider the case that you receive a software update for your PC, phone, car, pacemaker, etc.. over an open connection such as an unencrypted Wi-Fi. Then the contents of that update are not secret, but it is of crucial importance that no malicious attacker had modified the code and that it was unchanged from the message sent out by the company. Similarly, when you log into your bank, you might be much more concerned about the possibility of someone impersonating you and cleaning out your account than you are about the secrecy of your information.

Let's start with a very simple scenario which I'll call **the login problem**. Alice and Bob share a key as before, but now Alice wants to simply prove her identity to Bob. What makes it challenging is that this time they need to tackle not the passive eavesdropping Eve but the active adversary **Mallory** who completely controls the communication channel between them and can modify (or *mall*) any message that they send out. Specifically for the identity proving case, we think of the following scenario. Each instance of such an **identification protocol** consists of some interaction between Alice and Bob that ends with Bob deciding whether to accept it as authentic or

reject as an impersonation attempt. Mallory's goal is to fool Bob into accepting her as Alice.

The most basic way to try to solve the login problem is simply using a *password*. That is, if we assume that Alice and Bob can share a key, we can treat this key as some secret password p that was selected at random from $\{0,1\}^n$ (and hence can only be guessed with probability 2^{-n}). Why doesn't Alice simply send p to Bob to prove to him her identity? A moment's thought shows that this would be a very bad idea. Since Mallory is controlling the communication line, she would learn p after the first identification attempt and then could impersonate Alice in future interactions. However, we seem to have just the tool to protect the secrecy of p —*encryption*. Suppose that Alice and Bob share a secret key k and an additional secret password p . Wouldn't a simple way to solve the login problem be for Alice to send to Bob an encryption of the password p ? After all, the security of the encryption should guarantee that Mallory can't learn p , right?



This would be a good time to stop reading and try to think for yourself whether using a secure encryption to encrypt p would guarantee security for the login problem. (No really, stop and think about it.)

The problem is that Mallory does not have to learn the password p in order to impersonate Alice. For example, she can simply record the message Alice c_1 sends to Bob in the first session and then *replay* it to Bob in the next session. Since the message is a valid encryption of p , then Bob would accept it from Mallory! (This is known as a *replay attack* and is a common concern one needs to protect against in cryptographic protocols.) One can try to put in countermeasures to defend against this particular attack, but its existence demonstrates that secrecy of the password does not guarantee security of the login protocol.

4.1.1 How do pseudorandom functions help in the login problem?

The idea is that they create what's known as a *one time password*. Alice and Bob will share an index $s \in \{0,1\}^n$ for the pseudorandom function generator $\{f_s\}$. When Alice wants to prove to Bob her identity, Bob will choose a random $i \leftarrow_R \{0,1\}^n$, and send i to Alice, and then Alice will send $f_s(i), f_s(i+1), \dots, f_s(i+\ell - 1)$ to Bob where ℓ is some parameter (you can think of $\ell = n$ for simplicity). Bob will check that indeed $y = f_s(i)$ and if so accept the session as authentic.

The formal protocol is as follows:

Protocol PRF-Login:

- Shared input: $s \in \{0,1\}^n$. Alice and Bob treat it as a seed for a pseudorandom function generator $\{f_s\}$.
- In every session Alice and Bob do the following:
 1. Bob chooses a random $i \leftarrow_R [2^n]$ and sends i to Alice.
 2. Alice sends y_1, \dots, y_ℓ to Bob where $y_j = f_s(i + j - 1)$.
 3. Bob checks that for every $j \in \{1, \dots, \ell\}$, $y_j = f_s(i + j - 1)$ and if so accepts the session; otherwise he rejects it.

As we will see it's not really crucial that the input i (which is known in crypto parlance as a *nonce*) is random. What is crucial is that it never repeats itself, to foil a replay attack. For this reason in many applications Alice and Bob compute i as a function of the current time (for example, the index of the current minute based on some agreed-upon starting point), and hence we can make it into a one message protocol. Also the parameter ℓ is sometimes chosen to be deliberately short so that it will be easy for people to type the values y_1, \dots, y_ℓ .



Figure 4.2: The Google Authenticator app is one popular example of a one-time password scheme using pseudorandom functions. Another example is RSA's SecurID token.

Why is this secure? The key to understanding schemes using pseudorandom functions is to imagine what would happen if instead of a *pseudo* random function, f_s would be an *actual* random function. In a truly random function, every one of the values $f_s(0), \dots, f_s(2^n - 1)$ is chosen independently and uniformly at random from $\{0,1\}$. One useful way to imagine this is using the concept of “lazy evaluation”. We can think of f_s as determined by tossing 2^n different coins for the

values $f(0), \dots, f(2^n - 1)$. Now consider the case where we don't actually toss the i^{th} coin until we need it. The crucial point is that if we have queried the function in $T \ll 2^n$ places, when Bob chooses a random $i \in [2^n]$ then it is *extremely unlikely* that any one of the set $\{i, i+1, \dots, i+\ell-1\}$ will be one of those locations that we previously queried. Thus, if the function was truly random, Mallory has *no information* on the value of the function in these coordinates, and would be able to predict it in all these locations with probability at most $2^{-\ell}$.



Please make sure you understand the informal reasoning above, since we will now translate this into a formal theorem and proof.

Theorem 4.3 — Login protocol via PRF. Suppose that $\{f_s\}$ is a secure pseudorandom function generator and Alice and Bob interact using Protocol PRF-Login for some polynomial number T of sessions (over a channel controlled by Mallory), and then Mallory interacts with Bob, where Bob follows the protocol's instructions but Mallory may use an arbitrary efficient computation. Then, the probability that Bob accepts the interaction after this interaction is at most $2^{-\ell} + \mu(n)$ where $\mu(\cdot)$ is some negligible function.

Proof. This proof, as so many others in this course, uses an argument via contradiction. We assume, towards the sake of contradiction, that there exists an adversary M (for Mallory) that can break the identification scheme PRF-Login with probability $2^{-\ell} + \epsilon$ after T interactions and construct an attacker A that can distinguish access to $\{f_s\}$ from access to a random function using $\text{poly}(T)$ time and with bias at least $\epsilon/2$.

How do we construct this adversary A ? The idea is as follows. First, we prove that if we ran the protocol PRF-Login using an *actual random* function, then M would not be able to succeed in impersonating with probability better than $2^{-\ell} + \text{negligible}$. Therefore, if M does do better, then we can use that to distinguish f_s from a random function. The adversary A gets some black box $F(\cdot)$ and will use it while internally simulating all the parties—Alice, Bob and Mallory (using M) in the $T + 1$ interactions of the PRF-Login protocol. Whenever any of the parties needs to evaluate $f_s(i)$, A will forward i to its black box $F(\cdot)$ and return the value $F(i)$. It will then output 1 if and only if M succeeds in impersonation in this internal simulation. The argument above showed that if $F(\cdot)$ is a truly random function then the probability A outputs 1 is at most $2^{-\ell} + \text{negligible}$ (and so in particular less than $2^{-\ell} + \epsilon/2$ while under our assumptions, if

$F(\cdot)$ is the function $i \mapsto f_s(i)$ for some fixed and random s , then this probability is at least $2^{-\ell} + \epsilon$. Thus A will distinguish between the two cases with bias at least $\epsilon/2$. We now turn to the formal proof:

Claim 1: Let PRF-Login^* be the hypothetical variant of the protocol PRF-Login where Alice and Bob share a completely random function $H : [2^n] \rightarrow \{0,1\}$. Then, no matter what Mallory does, the probability she can impersonate Alice after observing T interactions is at most $2^{-\ell} + (8\ell T)/2^n$.

(If PRF-Login^* is easier to prove secure than PRF-Login , you might wonder why we bother with PRF-Login in the first place and not simply use PRF-Login^* . The reason is that specifying a random function H requires specifying 2^n bits, and so that would be a *huge* shared key. So PRF-Login^* is not a protocol we can actually run but rather a hypothetical “mental experiment” that helps us in arguing about the security of PRF-Login .)

Proof of Claim 1: Let i_1, \dots, i_{2T} be the nonces chosen by Bob and received by Alice in the first T iterations. That is, i_1 is the nonce chosen by Bob in the first iteration while i_2 is the nonce that Alice received in the first iteration (if Mallory doesn’t modify it then $i_1 = i_2$), similarly i_3 is the nonce chosen by Bob in the second iteration while i_4 is the nonce received by Alice and so on and so forth. Let i be the nonce chosen in the $T + 1^{st}$ iteration in which Mallory tries to impersonate Alice. We claim that the probability that there exists some $j \in \{1, \dots, 2T\}$ such that $|i - i_j| < 2\ell$ is at most $(8\ell T)/2^n$. Indeed, let S be the union of all the intervals of the form $\{i_j - 2\ell + 1, \dots, i_j + 2\ell - 1\}$ for $1 \leq j \leq 2T$. Since it’s a union of $2T$ intervals each of length less than 4ℓ , S contains at most $8T\ell$ elements, but so the probability that $i \in S$ is $|S|/2^n \leq (8T\ell)/2^n$. Now, if there does *not* exist a j such that $|i - i_j| < 2\ell$ then it means in particular that all the queries to $H(\cdot)$ made by either Alice or Bob during the first T iterations are disjoint from the interval $\{i, i+1, \dots, i+\ell-1\}$. Since $H(\cdot)$ is a completely random function, the values $H(i), \dots, H(i+\ell-1)$ are chosen uniformly and independently from all the rest of the values of this function. Since Mallory’s message y to Bob in the $T + 1^{st}$ iteration depends only on what she observed in the past, the values $H(i), \dots, H(i+\ell-1)$ are *independent* from y , and hence under this condition that there is no overlap between this interval and prior queries, the probability that they equal y is $2^{-\ell}$. QED (Claim 1).

The proof of Claim 1 is not hard, but it is somewhat subtle, and it’s good to go over it again and make sure you are sure you understand it.

Now that we have Claim 1, the proof of the theorem follows as outlined above. We build an adversary A to the pseudorandom function generator from M by having A simulate “inside its belly” all the parties Alice, Bob and Mallory and output 1 if Mallory succeeds in impersonating. Since we assumed ϵ is non-negligible and T is polynomial, we can assume that $(8\ell T)/2^n < \epsilon/2$ and hence by Claim 1, if the black box is a random function then we are in the PRF-Login* setting and Mallory’s success will be at most $2^{-\ell} + \epsilon/2$ while if the black box is $f_s(\cdot)$ then we get exactly the same setting as PRF-Login and hence under our assumption the success will be at least $2^{-\ell} + \epsilon$. We conclude that the difference in probability of A outputting one between the random and pseudorandom case is at least $\epsilon/2$ thus contradicting the security of the pseudorandom function generator. ■

R

Increasing output length of PRFs In the course of constructing this one-time-password scheme from a PRF, we have actually proven a general statement that is useful on its own: that we can transform standard PRF which is a collection $\{f_s\}$ of functions mapping $\{0,1\}^n$ to $\{0,1\}$, into a PRF where the functions have a longer output ℓ (see the problem set for a formal statement of this result) Thus from now on whenever we are given a PRF, we will allow ourselves to assume that it has any output size that is convenient for us.

4.2 Message Authentication Codes

One time passwords are a tool allowing you to prove your *identity* to, say, your email server. But even after you did so, how can the server trust that future communication comes from you and not from some attacker that can interfere with the communication channel between you and the server (so called “man in the middle” attack). Similarly, one time passwords may allow a software company to prove their identity before they send you a software update, but how do you know that an attacker does not change some bits of this software update on route between their servers and your device?

This is where *Message Authentication Codes* (MACs) come into play—their role is to authenticate not merely the *identity* of the parties but also their *communication*. Once again we have **Alice** and **Bob**, and the adversary **Mallory** who can actively modify messages (in contrast to the passive eavesdropper **Eve**). Similar to the case to encryption,

Alice has a *message* m she wants to send to Bob, but now we are not concerned with Mallory *learning* the contents of the message. Rather, we want to make sure that Bob gets precisely the message m sent by Alice. Actually this is too much to ask for, since Mallory can always decide to block all communication, but we can ask that either Bob gets precisely m or he detects failure and accepts no message at all. Since we are in the *private key* setting, we assume that Alice and Bob share a key k that is unknown to Mallory.

What kind of security would we want? We clearly want Mallory not to be able to cause Bob to accept a message $m' \neq m$. But, like in the encryption setting, we want more than that. We would like Alice and Bob to be able to use the same key for *many* messages. So, Mallory might observe the interactions of Alice and Bob on messages m_1, \dots, m_T before trying to cause Bob to accept a message $m'_{T+1} \neq m_{T+1}$. In fact, to make our notion of security more robust, we will even allow Mallory to *choose* the messages m_1, \dots, m_T (this is known as a *chosen message* or *chosen plaintext* attack). The resulting formal definition is below:

Definition 4.4 — Message Authentication Codes (MAC). Let (S, V) (for *sign* and *verify*) be a pair of efficiently computable algorithms where S takes as input a key k and a message m , and produces a tag $\tau \in \{0,1\}^*$, while V takes as input a key k , a message m , and a tag τ , and produces a bit $b \in \{0,1\}$. We say that (S, V) is a *Message Authentication Code (MAC)* if:

- For every key k and message m , $V_k(m, S_k(m)) = 1$.
- For every polynomial-time adversary A and polynomial $p(n)$, it is with less than $1/p(n)$ probability over the choice of $k \leftarrow_R \{0,1\}^n$ that $A^{S_k(\cdot)}(1^n) = (m', \tau')$ such that m' is *not* one of the messages A queries and $V_k(m', \tau') = 1$.²

If Alice and Bob share the key k , then to send a message m to Bob, Alice will simply send over the pair (m, τ) where $\tau = S_k(m)$. If Bob receives a message (m', τ') , then he will accept m' if and only if $V_k(m', \tau') = 1$. Now, Mallory could observe t rounds of communication of the form $(m_i, S_k(m_i))$ for messages m_1, \dots, m_t of her choice, and now her goal is to try to create a new message m' that was *not* sent by Alice, but for which she can forge a valid tag τ' that will pass verification. Our notion of security guarantees that she'll only be able to do so with negligible probability.³

² Clearly if the adversary outputs a pair (m, τ) that it did query from its oracle then that pair will pass verification. This suggests the possibility of a *replay* attack whereby Mallory resends to Bob a message that Alice sent him in the past. As above, once can thwart this by insisting the every message m begins with a fresh nonce or a value derived from the current time.

³ A priori you might ask if we should not also give Mallory an oracle to $V_k(\cdot)$ as well. After all, in the course of those many interactions, Mallory could also send Bob many messages (m', τ') of her choice, and observe from his behavior whether or not these passed verification. It is a good exercise to show that adding such an oracle does not change the power of the definition, though we note that this is decidedly *not* the case in the analogous question for encryption.

R **Why can Mallory choose the messages?** The notion of a “chosen message attack” might seem a little “over the top”. After all, Alice is going to send to Bob the messages of *her* choice, rather than those chosen by her adversary Mallory. However, as cryptographers have learned time and again the hard way, it is better to be conservative in our security definitions and think of an attacker that has as much power as possible. First of all, we want a message authentication code that will work for *any* sequence of messages, and so it’s better to consider this “worst case” setting of allowing Mallory to choose them. Second, in many realistic settings an adversary could have some effect on the messages that are being sent by the parties. This has occurred time and again in cases ranging from web servers to German submarines in World War II, and we’ll return to this point when we talk about *chosen plaintext* and *chosen ciphertext* attacks on encryption schemes.

R **Strong unforgeability** Some texts (such as Boneh Shoup) define a stronger notion of unforgeability where the adversary cannot even produce new signatures for messages it *has* queried in the attack. That is, the adversary cannot produce a valid message-signature pair that it has not seen before. This stronger definition can be useful for some applications. It is fairly easy to transform MACs satisfying Definition 4.4 into MACs satisfying strong unforgeability. In particular, if the signing function is deterministic, and we use a *canonical verifier algorithm* where $V_k(m, \sigma) = 1$ iff $S_k(m) = \sigma$ then weak unforgeability automatically implies strong unforgeability since every message has a single signature that would pass verification (can you see why?).

4.3 MACs from PRFs

We now show how pseudorandom function generators yield message authentication codes. In fact, the construction is so immediate, that much of the more applied cryptographic literature does not distinguish between these two concepts, and uses the name “Message Authentication Codes” to refer to both MAC’s and PRF’s.

Theorem 4.5 — MAC Theorem. Under the PRF Conjecture, there exists a secure MAC.

Proof. Let $F(\cdot, \cdot)$ be a secure pseudorandom function generator with $n/2$ bits output (as mentioned in Remark 4.1.1, such PRF's can be constructed from one bit output PRF's). We define $S_k(m) = F(k, m)$ and $V_k(m, \tau)$ to output 1 iff $F_k(m) = \tau$. Suppose towards the sake of contradiction that there exists an adversary A that queries $S_k(\cdot)$ $\text{poly}(n)$ many times and outputs (m', τ') that she did *not* ask for and such that $F(k, m') = \tau'$. Now, if we had black box access to a completely random function $H(\cdot)$, then the value $H(m')$ would be completely random in $\{0, 1\}^{n/2}$ and independent of all prior queries. Hence the probability that this value would equal τ' is at most $2^{-n/2}$. That means that such an adversary can distinguish between an oracle to $F_k(\cdot)$ and an oracle to a random function H . ■

4.4 Input length extension for MACs and PRFs

So far we required the message to be signed m to be no longer than the key k (i.e., both n bits long). However, it is not hard to see that this requirement is not really needed. If our message is longer, we can divide into blocks m_1, \dots, m_t and sign each message (i, m_i) individually. The disadvantage here is that the size of the tag (i.e., MAC output) will grow with the size of the message. However, even this is not really needed. Because the tag has length $n/2$ for length n messages, we can sign the tags τ_1, \dots, τ_t and only output those. The verifier can repeat this computation to verify this. We can continue this way and so get tags of $O(n)$ length for arbitrarily long messages. Hence in the future, whenever we need to, we will assume that our PRFs and MACs can get inputs in $\{0, 1\}^*$ — i.e., arbitrarily length strings.

We note that this issue of length extension is actually quite a thorny and important one in practice. The above approach is not the most efficient way to achieve this, and there are several more practical variants in the literature (see Boneh-Shoup Sections 6.4-6.8). Also, one needs to be very careful on the exact way one chops the message into blocks and pads it to an integer multiple of the block size. Several attacks have been mounted on schemes that performed this incorrectly.

4.5 Aside: natural proofs

Pseudorandom functions play an important role in computational complexity, where they have been used as a way to give “barrier

results” for proving results such as $\mathbf{P} \neq \mathbf{NP}$.⁴ Specifically, the **Natural Proofs** barrier for proving circuit lower bounds says that if strong enough pseudorandom functions exist, then certain types of arguments are bound to fail. These are arguments which come up with a property *EASY* of a Boolean function $f : \{0,1\}^n \rightarrow \{0,1\}$ such that:

- If f can be computed by a polynomial sized circuit, then it has the property *EASY*.
- The property *EASY* fails to hold for a random function with high probability.
- Checking whether *EASY* holds can be done in time polynomial *in the truth table size of f* . That is, in $2^{O(n)}$ time.

A priori these technical conditions might not seem very “natural” but it turns out that many approaches for proving circuit lower bounds (for restricted families of circuits) have this form. The idea is that such approaches find a “non generic” property of easily computable function, such as finding some interesting correlations between the some input bits and the output. These are correlations that are unlikely to occur in random functions. The lower bound typically follows by exhibiting a function f_0 that does not have this property, and then using that to derive that f_0 cannot be efficiently computed by this particular restricted family of circuits.

The existence of strong enough pseudorandom functions can be shown to contradict the existence of such a property *EASY*, since a pseudorandom function can be computed by a polynomial sized circuit, but it cannot be distinguished from a random function. While a priori a pseudorandom function is only secure for polynomial time distinguishers, under certain assumptions it might be possible to create a pseudorandom function with a seed of size, say, n^5 , that would be secure with respect to adversaries running in time $2^{O(n^2)}$.

⁴ This discussion has more to do with computational complexity than cryptography, and so can be safely skipped without harming understanding of future material in this course.

5

Pseudorandom functions from pseudorandom generators

We have seen that PRF's (pseudorandom functions) are extremely useful, and we'll see some more applications of them later on. But are they perhaps too amazing to exist? Why would someone imagine that such a wonderful object is feasible? The answer is the following theorem:

Theorem 5.1 — The PRF Theorem. Suppose that the PRG Conjecture is true, then there exists a secure PRF collection $\{f_s\}_{s \in \{0,1\}^*}$ such that for every $s \in \{0,1\}^n$, f_s maps $\{0,1\}^n$ to $\{0,1\}^n$.

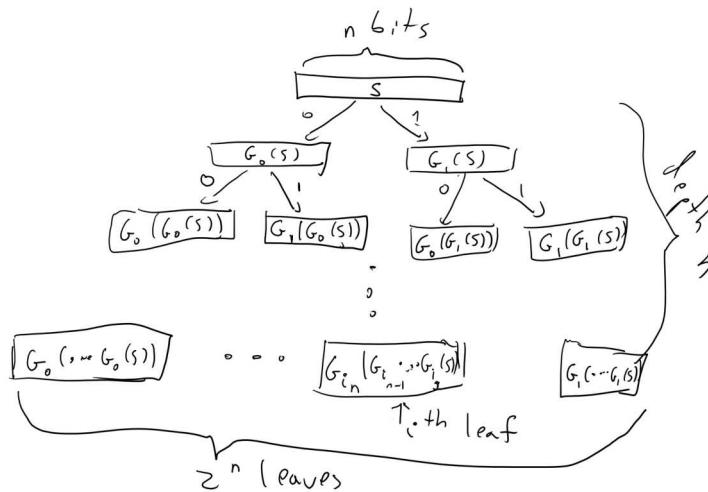


Figure 5.1: The construction of a pseudorandom function from a pseudorandom generator can be illustrated by a depth n binary tree. The root is labeled by the seed s and for every internal node v labeled by a strong $x \in \{0,1\}^n$, we label the two children of v by $G_0(x)$ and $G_1(x)$ respectively. The output of the function f_s on input i is the label of the i^{th} leaf.

Proof. If the PRG Conjecture is true then in particular by the length extension theorem there exists a PRG $G : \{0,1\}^n \rightarrow \{0,1\}^{2n}$ that

maps n bits into $2n$ bits. Let's denote $G(s) = G_0(s) \circ G_1(s)$ where \circ denotes concatenation. That is, $G_0(s)$ denotes the first n bits and $G_1(s)$ denotes the last n bits of $G(s)$.

For $i \in \{0,1\}^n$, we define $f_s(i)$ as

$$G_{i_n}(G_{i_{n-1}}(\dots G_{i_1}(s))). \quad (5.1)$$

This might be a bit hard to parse and is easier to understand by looking at Fig. 5.1.

By the definition above we can see that to evaluate $f_s(i)$ we need to evaluate the pseudorandom generator n times on inputs of length n , and so if the pseudorandom generator is efficiently computable then so is the pseudorandom function. Thus, “all” that’s left is to prove that the construction is secure and this is the heart of this proof.

I've mentioned before that the first step of writing a proof is convincing yourself that the statement is true, but there is actually an often more important zeroth step which is understanding what the statement actually *means*. In this case what we need to prove is the following:

Given an adversary A that can distinguish in time T a black box for $f_s(\cdot)$ from a black-box for a random function with advantage ϵ , we need to come up with an adversary D that can distinguish in time $poly(T)$ an input of the form $G(s)$ (where s is random in $\{0,1\}^n$) from an input of the form y where y is random in $\{0,1\}^{2n}$ with bias at least $\epsilon/poly(T)$.

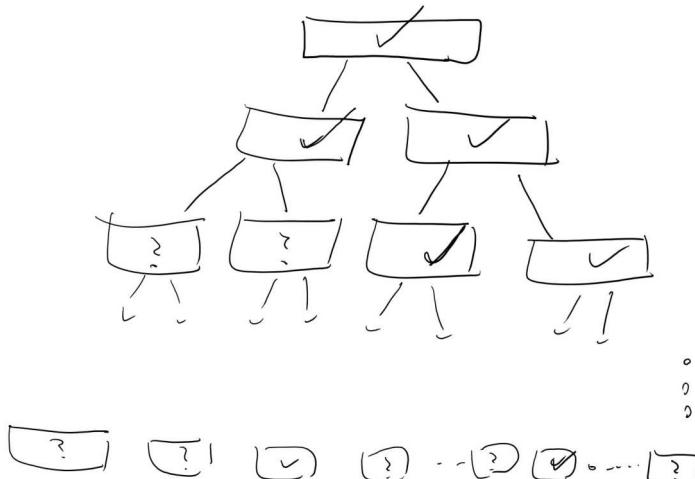


Figure 5.2: In the “lazy evaluation” implementation of the black box to the adversary, we label every node in the tree only when we need it. In this figure check marks correspond to nodes that have been labeled and question marks to nodes that are still unlabeled.

Let us consider the “lazy evaluation” implementation of the black box for A illustrated in Fig. 5.2. That is, at every point in time there are nodes in the full binary tree that are labeled and nodes which we haven’t yet labeled. When A makes a query i , this query corresponds to the path $i_1 \dots i_n$ in the tree. We look at the lowest (furthest away from the root) node v on this path which has been labeled by some value y , and then we continue labelling the path from v downwards until we reach i . In other words, we label the two children of v by $G_0(y)$ and $G_1(y)$, and then if the path i involves the first child then we label its children by $G_0(G_0(y))$ and $G_1(G_0(y))$, and so on and so forth (see Fig. 5.3).

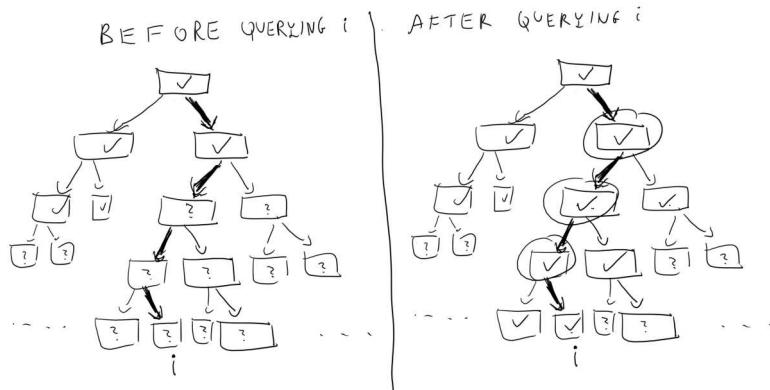


Figure 5.3: When the adversary queries i , the oracle takes the path from i to the root and computes the generator on the minimum number of internal nodes that is needed to obtain the label of the i^{th} leaf

A moment’s thought shows that this is just another (arguably cumbersome) way to describe the oracle that simply computes the map $i \mapsto f_s(i)$. And so the experiment of running A with this oracle produces precisely the same result as running A with access to $f_s(\cdot)$. Note that since A has running time at most T , the number of times our oracle will need to label an internal node is at most $T' \leq 2nT$ (since we label at most $2n$ nodes for every query i).

We now define the following T' hybrids: in the j^{th} hybrid, we run this experiment but in the first j times the oracle needs to label internal nodes, then instead of labelling the b^{th} child of v by $G_b(y)$ (where y is the label of v), the oracle simply labels it by a random string in $\{0,1\}^n$.

Note that the 0^{th} hybrid corresponds to the case where the oracle implements the function $i \mapsto f_s(i)$ will in the T'^{th} hybrid all labels are random and hence the oracle implements a random function. By the hybrid argument, if A can distinguish between the 0^{th} hybrid and the T'^{th} hybrid with bias ϵ then there must exist some j such that it distinguishes between the j^{th} hybrid and the $j + 1^{st}$ hybrid with bias

at least ϵ/T' . We will use this j and A to break the pseudorandom generator.

We can now describe our distinguisher D for the pseudorandom generator. On input a string $y \in \{0,1\}^{2n}$ D will run A and the j^{th} oracle inside its belly with one difference- when the time comes to label the j^{th} node, instead of doing this by applying the pseudorandom generator to the label of its parent v (which is what should happen in the j^{th} oracle) it uses its input y to label the two children of v .

Now, if y was completely random then we get exactly the distribution of the $j+1^{st}$ oracle, and hence in this case D simulates internally the $j+1^{st}$ hybrid. However, if $y = G(s)$ for a random $s \in \{0,1\}^n$ it might not be obvious if we get the distribution of the j^{th} oracle, since in that oracle the label for the children of v was supposed to be the result of applying the pseudorandom generator to the label of v and not to some other random string. However, because v was labeled *before* the j^{th} step then we know that it was actually labeled by a random string. Moreover, since we use lazy evaluation we know that step j is the *first* time where we actually use the value of the label of v . Hence, if at this point we *resampled* this label and used a completely independent random string then the distribution would be *identical*. Hence if $y = G(U_n)$ then D actually does simulate the distribution of the j^{th} hybrid in its belly, and thus if A had advantage ϵ in breaking the PRF $\{f_s\}$ then D will have advantage ϵ/T' in breaking the PRG G thus obtaining a contradiction.

This proof is ultimately not very hard but is rather confusing. I urge you to also look at the proof of this theorem as is written in Section 7.5 (pages 265-269) of the KL book. ■



PRF's in practice While this construction reassures us that we can rely on the existence of pseudorandom functions even on days where we remember to take our meds, this is not the construction people use when they need a PRF in practice because it is still somewhat inefficient, making n calls to the underlying pseudorandom generators. There are constructions (e.g., HMAC) based on hash functions that require stronger assumptions but can use as few as two calls to the underlying function. We will cover these constructions when we talk about hash functions and the random oracle model. One can also obtain practical constructions of PRFs from *block ciphers*, which we'll see later in this lecture.

5.1 Securely encrypting many messages - chosen plaintext security

Let's get back to our favorite task of *encryption*. We seemed to have nailed down the definition of secure encryption, or did we?



Try to think what kind of security guarantees are *not* provided by the notion of computational secrecy we saw in [Definition 2.4](#)

Our current definition requires talks about encrypting a *single* message, but this is not how we use encryption in the real world. Typically, Alice and Bob (or Amazon and Boaz) setup a shared key and then engage in many back and forth messages between one another. At first, we might think that this issue of a single long message vs. many short ones is merely a technicality. After all, if Alice wants to send a sequence of messages (m_1, m_2, \dots, m_t) to Bob, she can simply treat them as a single long message. Moreover, the way that *stream ciphers* work, Alice can compute the encryption for the first few bits of the message she decides what will be the next bits and so she can send the encryption of m_1 to Bob and later the encryption of m_2 . There is some truth to this sentiment, but there are issues with using stream ciphers for multiple messages. For Alice and Bob to encrypt messages in this way, they must maintain a *synchronized shared state*. If the message m_1 was dropped by the network, then Bob would not be able to decrypt correctly the encryption of m_2 .

There is another way in which treating many messages as a single tuple is unsatisfactory. In real life, Eve might be able to have some impact on *what* messages Alice encrypts. For example, the Katz-Lindell book describes several instances in world war II where Allied forces made particular military manouvers for the sole purpose of causing the axis forces to send encryptions of messages of the Allies' choosing. To consider a more modern example, today Google uses encryption for all of its search traffic including (for the most part) the *ads* that are displayed on the page. But this means that an attacker, by paying Google, can cause it to encrypt arbitrary text of their choosing. This kind of attack, where Eve *chooses* the message she wants to be encrypted is called a *chosen plaintext attack*. You might think that we are already covering this with our current definition that requires security for *every* pair of messages and so in particular this pair could chosen by Eve. However, in the case of multiple messages, we would want to allow Eve to be able to choose m_2 *after* she saw the

encryption of m_1 .

All that leads us to the following definition, which is a strengthening of our definition of computational security:

Definition 5.2 — Chosen Plaintext Attack (CPA) secure encryption. An encryption scheme (E, D) is *secure against chosen plaintext attack (CPA secure)* if for every polynomial time Eve, Eve wins with probability at most $1/2 + \text{negl}(n)$ in the game defined below:

1. The key k is chosen at random in $\{0,1\}^n$ and fixed.
2. Eve gets the length of the key 1^n as input.¹
3. Eve interacts with E for $t = \text{poly}(n)$ rounds as follows: in the i^{th} round, Eve chooses a message m_i and obtains $c_i = E_k(m_i)$.
4. Then Eve chooses two messages m_0, m_1 , and gets $c^* = E_k(m_b)$ for $b \leftarrow_R \{0,1\}$.
5. Eve *wins* if she outputs b .

¹ Giving Eve the key as a sequence of n 1's as opposed to in binary representation is a common notational convention in cryptography. It makes no difference except that it makes the input length for Eve of length n , which makes sense since we want to allow Eve to run in $\text{poly}(n)$ time.

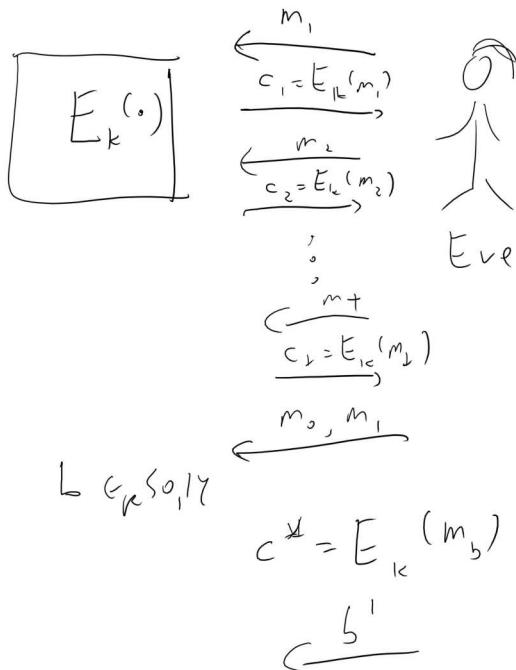


Figure 5.4: In the CPA game, Eve interacts with the encryption oracle and at the end chooses m_0, m_1 , gets an encryption $c^* = E_k(m_b)$ and outputs b' . She *wins* if $b' = b$

Definition 5.2 is illustrated in Fig. 5.4. Our previous notion of computational secrecy (i.e., Definition 2.4) corresponds to the case that we skip Step 3 above. Since Step 3 only gives the adversary more power (and hence is only more likely to win), CPA security (Definition

tion 5.2) is *stronger* than computational secrecy (Definition 2.4), in the sense that every CPA secure encryption (E, D) is also computationally secure. It turns out that CPA security is *strictly stronger*, in the sense that without modification, our stream ciphers cannot be CPA secure. In fact, we have a stronger, and initially somewhat surprising theorem:

Theorem 5.3 — CPA security requires randomization. There is no CPA secure (E, D) where E is *deterministic*.

Proof. The proof is very simple: Eve will only use a single round of interacting with E where she will ask for the encryption c_1 of 0^ℓ . In the second round, Eve will choose $m_0 = 0^\ell$ and $m_1 = 1^\ell$, and get $c^* = E_k(m_b)$ she will then output 0 if and only if $c^* = c_1$. ■



Figure 5.5: Insecurity of deterministic encryption

This proof is so simple that you might think it shows a problem with the definition, but it is actually a real problem with security. If you encrypt many messages and some of them repeat themselves, it is possible to get significant information by seeing the repetition pattern (que the XKCD cartoon again, see Fig. 5.5). To avoid this issue we need to use a *randomized* (or *probabilistic*) encryption, such that if we encrypt the same message twice we *won't* see two copies of the same ciphertext.² But how do we do that? Here pseudorandom functions come to the rescue:

² If the messages are guaranteed to have *high entropy* which roughly means that the probability that a message repeats itself is negligible, then it is possible to have a secure deterministic private-key encryption, and this is sometimes used in practice. (Though often some sort of randomization or padding is added to ensure this property, hence in effect creating a randomized encryption.) Deterministic encryptions can sometimes be useful for applications such as efficient queries on encrypted databases. See this lecture in Dan Boneh's coursera course.

Theorem 5.4 — CPA security from PRFs. Suppose that $\{f_s\}$ is a PRF collection where $f_s : \{0,1\}^n \rightarrow \{0,1\}^\ell$, then the following is a CPA secure encryption scheme: $E_s(m) = (r, f_s(r) \oplus m)$ and $D_s(r, z) = f_s(r) \oplus z$.

Proof. I leave to you to verify that $D_s(E_s(m)) = m$. We need to show the CPA security property. As is usual in PRF-based constructions, we first show that this scheme will be secure if f_s was an actually random function, and then use that to derive security.

Consider the game above when played with a completely random function and let r_i be the random string chosen by E in the i^{th} round and r^* the string chosen in the last round. We start with the following simple but crucial claim:

Claim: The probability that $r^* = r_i$ for some i is at most $T/2^n$.

Proof of claim: For any particular i , since r^* is chosen independently of r_i , the probability that $r^* = r_i$ is 2^{-n} . Hence the claim follows from the union bound. QED

Given this claim we know that with probability $1 - T/2^n$ (which is $1 - \text{negl}(n)$), the string r^* is distinct from any string that was chosen before. This means that by the lazy evaluation principle, if $f_s(\cdot)$ is a completely random function then the value $f_s(r^*)$ can be thought of as being chosen at random in the final round independently of anything that happened before. But then $f_s(r^*) \oplus m_b$ amounts to simply using the one-time pad to encrypt m_b . That is, the distributions $f_s(r^*) \oplus m_0$ and $f_s(r^*) \oplus m_1$ (where we think of r^*, m_0, m_1 as fixed and the randomness comes from the choice of the random function $f_s(\cdot)$) are both equal to the uniform distribution U_n over $\{0,1\}^n$ and hence Eve gets absolutely no information about b .

This shows that if $f_s(\cdot)$ was a random function then Eve would win the game with probability at most $1/2$. Now if we have some efficient Eve that wins the game with probability at least $1/2 + \epsilon$ then we can build an adversary A for the PRF that will run this entire game with black box access to $f_s(\cdot)$ and will output 1 if and only if Eve wins. By the argument above, there would be a difference of at least ϵ in the probability it outputs 1 when $f_s(\cdot)$ is random vs when it is pseudorandom, hence contradicting the security property of the PRF. ■

5.2 Pseudorandom permutations / block ciphers

Now that we have pseudorandom functions, we might get greedy and want such functions with even more magical properties. This is where the notion of *pseudorandom permutations* comes in.

Definition 5.5 — Pseudorandom permutations. Let $\ell : \mathbb{N} \rightarrow \mathbb{N}$ be some function that is polynomially bounded (i.e., there are some $0 < c < C$ such that $n^c < \ell(n) < n^C$ for every n). A collection of functions $\{f_s\}$ where $f_s : \{0,1\}^\ell \rightarrow \{0,1\}^\ell$ for $\ell = \ell(|s|)$ is called a *pseudorandom permutation (PRP) collection* if:

1. It is a pseudorandom function collection (i.e., the map $s, x \mapsto f_s(x)$ is efficiently computable and there is no efficient distinguisher between $f_s(\cdot)$ with a random s and a random function).
2. Every function f_s is a permutation of $\{0,1\}^\ell$ (i.e., a one to one and onto map).
3. There is an efficient algorithm that on input s, y returns $f_s^{-1}(y)$.

The parameter n is known as the *key length* of the pseudorandom permutation collection and the parameter $\ell = \ell(n)$ is known as the *input length* or *block length*. Often, $\ell = n$ and so in most cases you can safely ignore this distinction.



At first look [Definition 5.5](#) might seem not to make sense, since on one hand it requires the map $x \mapsto f_s(x)$ to be a permutation, but on the other hand it can be shown that with high probability a random map $H : \{0,1\}^\ell \rightarrow \{0,1\}^\ell$ will *not* be a permutation. How can then such a collection be pseudorandom? The key insight is that while a random map might not be a permutation, it is not possible to distinguish with a polynomial number of queries between a black box that computes a random function and a black box that computes a random permutation. Understanding why this is the case, and why this means that [Definition 5.5](#) is reasonable, is crucial to getting intuition to this notion, and so I suggest you pause now and make sure you understand these points.

As usual with a new concept, we want to know whether it is possible to achieve and is useful. The former is established by the following theorem:

Theorem 5.6 — PRP's from PRFs. If the PRF conjecture holds (and hence by [Theorem 5.1](#) also if the PRG conjecture holds) then there exists a pseudorandom permutation collection.

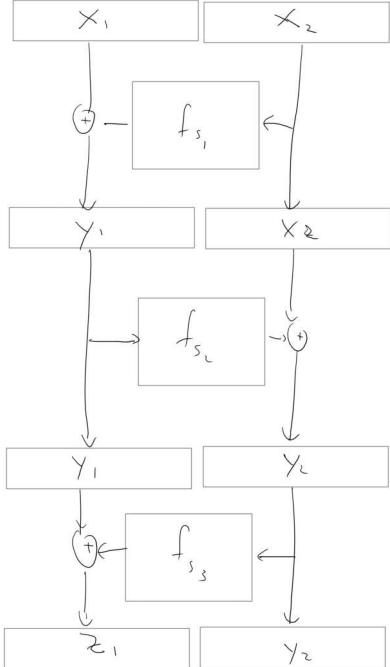


Figure 5.6: We build a PRP p on $2n$ bits from three PRFs $f_{s_1}, f_{s_2}, f_{s_3}$ on n bits by letting $p_{s_1, s_2, s_3}(x_1, x_2) = (z_1, y_2)$ where $y_1 = x_1 \oplus f_{s_1}(x_2)$, $y_2 = x_2 \oplus f_{s_2}(y_1)$ and $z_1 = f_{s_3}(y_2) \oplus y_1$.

We will not show the proof of this theorem here, but [Fig. 5.6](#) illustrates how the construction of a pseudorandom permutation from a pseudorandom function looks like. The construction (known as the Luby-Rackoff construction) uses several rounds of what is known as the **Feistel Transformation** that maps a function $f : \{0,1\}^n \rightarrow \{0,1\}^n$ into a permutation $g : \{0,1\}^{2n} \rightarrow \{0,1\}^{2n}$ using the map $(x, y) \mapsto (x, f(x) \oplus y)$. For an overview of the proof see Section 4.5 in Boneh Shoup or Section 7.6 in Katz-Lindell.

The more common name for a pseudorandom permutation is *block cipher* (though typically block ciphers are expected to meet additional security properties on top of being PRPs). The constructions for block ciphers used in practice don't follow the construction of [Theorem 5.6](#) (though they use some of the ideas) but have a more ad-hoc nature.

One of the first modern block ciphers was the **Data Encryption Standard (DES)** constructed by IBM in the 1970's. It is a fairly good cipher- to this day, as far as we know, it provides a pretty good number of security bits compared to its key. The trouble is that its

key is only 56 bits long, which is no longer outside the reach of modern computing power. (It turns out that subtle variants of DES are far less secure and fall prey to a technique known as **differential cryptanalysis**; the IBM designers of DES were aware of this technique but kept it secret at the behest of the NSA.)

Between 1997 and 2001, the U.S. national institute of standards (NIST) ran a competition to replace DES which resulted in the adoption of the block cipher Rijndael as the new **advanced encryption standard (AES)**. It has a block size (i.e., input length) of 128 bits and a key size (i.e., seed length) of 128, 196, or 256 bits.

The actual construction of AES (or DES for that matter) is not extremely illuminating, but let us say a few words about the general principle behind many block ciphers. They are typically constructed by repeating one after the other a number of very simple permutations (see Fig. 5.7). Each such iteration is called a *round*. If there are t rounds, then the key k is typically expanded into a longer string, which we think of as a t tuple of strings (k_1, \dots, k_t) via some pseudorandom generator known as the *key scheduling algorithm*. The i -th string in the tuple is known as the *round key* and is used in the i^{th} round. Each round is typically composed of several components: there is a “key mixing component” that performs some simple permutation based on the key (often as simply as XOR’ing the key), there is a “mixing component” that mixes the bits of the block so that bits that were initially nearby don’t stay close to one another, and then there is some non-linear component (often obtained by applying some simple non-linear functions known as “S boxes” to each small block of the input) that ensures that the overall cipher will not be an affine function. Each one of these operations is an easily reversible operations, and hence decrypting the cipher simply involves running the rounds backwards.

5.3 Encryption modes

How do we use a block cipher to actually encrypt traffic? Well we could use it as a PRF in the construction above, but in practice people use other ways.³ The most natural approach would be that to encrypt a message m , we simply use $p_s(m)$ where $\{p_s\}$ is the PRP/block cipher. This is known as the *electronic code book (ECB) mode* of a block cipher (see Fig. 5.8). Note that we can easily decrypt since we can compute $p_s^{-1}(m)$. However, this is a *deterministic* encryption and hence cannot be CPA secure. Moreover, this is actually a real problem of security on realistic inputs (see Fig. 5.9).

³ Partially this is because in the above construction we had to encode a plaintext of length n with a ciphertext of length $2n$ meaning an overhead of 100 percent in the communication.

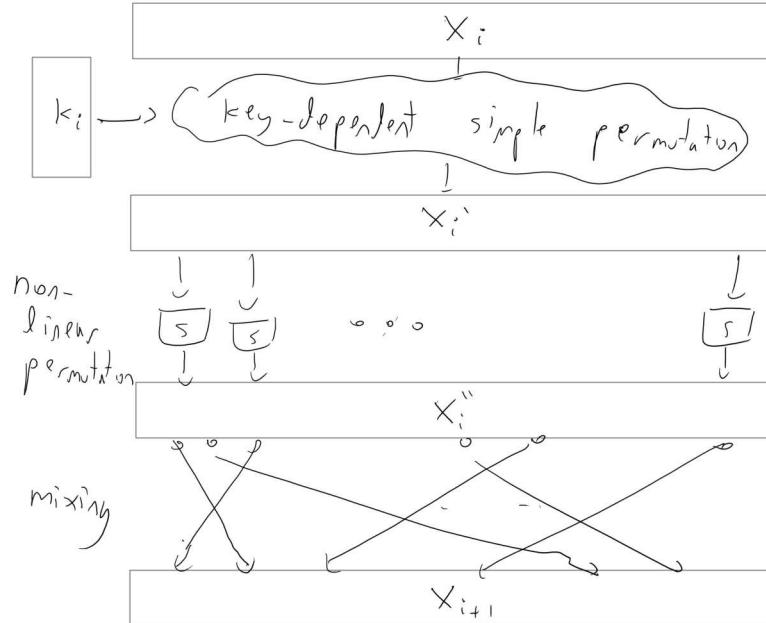


Figure 5.7: A typical round of a block cipher, k_i is the i^{th} round key, x_i is the block before the i^{th} round and x_{i+1} is the block at the end of this round.

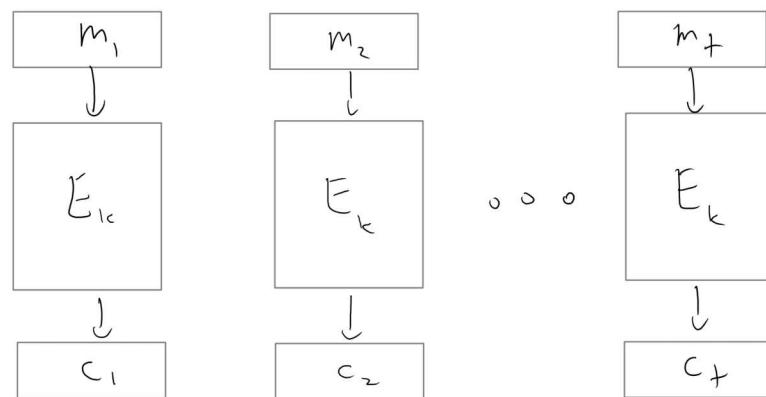


Figure 5.8: In the Electronic Codebook (ECB) mode every message is encrypted deterministically and independently

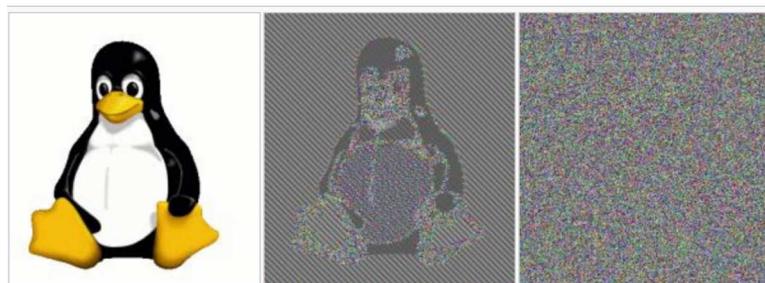


Figure 5.9: An encryption of the Linux penguin (left image) using ECB mode (middle image) vs CBC mode (right image). The ECB encryption is insecure as it reveals much structure about the original image. Image taken from Wikipedia.

A more secure way to use a block cipher to encrypt is the *cipher block chaining mode* where we XOR every message with the previous ciphertext (Fig. 5.10). For the first message we XOR a string known as the *initialization vector* or IV. Note that if we lose a block to traffic in the CBC mode then we are unable to decrypt the next block, but can recover from that point onwards. It turns out that using this mode with a random IV can yield CPA security, though one has to be careful in how you go about it, see the exercises.

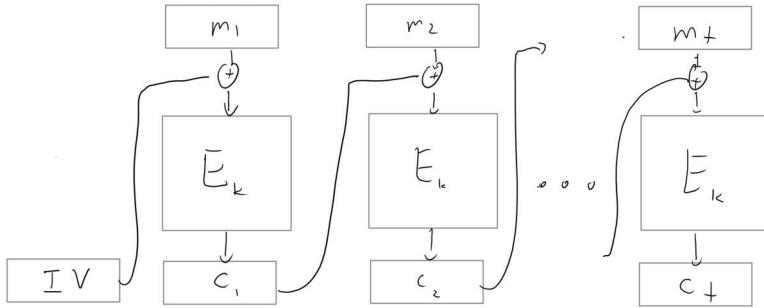


Figure 5.10: In the Cypher-Block-Chaining (CBC) the encryption of the previous message is XOR'ed into the current message prior to encrypting. The first message is XOR'ed with an *initialization vector* (IV) that if chosen randomly, ensures CPA security.

In the *output feedback mode* (OFB) we encrypt the all zero string using CBC mode to get a sequence (y_1, y_2, \dots) of pseudorandom outputs that we can use as a stream cipher. Perhaps the simplest mode is *counter mode* where we convert a block cipher to a stream cipher by using the stream $p_k(IV), p_k(IV + 1), p_k(IV + 2), \dots$ where IV is a random string in $\{0, 1\}^n$ which we identify with $[2^n]$ (and perform addition modulo 2^n). For a modern block cipher this should be no less secure than CBC or OFB and has advantages that we can easily compute it in parallel.

A fairly comprehensive study of the different modes of block ciphers is in [this document by Rogaway](#). His conclusion is that if we simply consider CPA security (as opposed to the stronger notions of *chosen ciphertext security* we'll see in the next lecture) then counter mode is the best choice, but CBC, OFB and CFB are widely implemented due to legacy reasons. ECB should not be used (except as a building block as part of a construction achieving stronger security).

6

Chosen Ciphertext Security

6.1 Short recap

Let's start by reviewing what we have learned so far:

- We can mathematically define security for encryption schemes. A natural definition is *perfect secrecy*: no matter what Eve does, she can't learn anything about the plaintext that she didn't know before. Unfortunately this requires the key to be as long as the message, thus placing a severe limitation on the usability of it.
- To get around this, we need to consider computational considerations. A basic object is a *pseudorandom generator* and we considered the *PRG Conjecture* which stipulates the existence of an efficiently computable function $G : \{0,1\}^n \rightarrow \{0,1\}^{n+1}$ such that $G(U_n) \approx U_{n+1}$ (where U_m denotes the uniform distribution on $\{0,1\}^m$ and \approx denotes computational indistinguishability).
- We showed that the PRG conjecture implies a pseudorandom generator of any polynomial output length which in particular via the stream cipher construction implies a computationally secure encryption with plaintext arbitrarily larger than the key. (The only restriction is that the plaintext is of polynomial size which is anyway needed if we want to actually be able to read and write it.)
- We then showed that the PRG conjecture actually implies a stronger object known as a *pseudorandom function (PRF) function collection*: this is a collection $\{f_s\}$ of functions such that if we choose s at random and fix it, and give an adversary a black box computing $i \mapsto f_s(i)$ then she can't tell the difference between this and a blackbox computing a random function.

- Pseudorandom functions turn out to be useful for *identification protocols*, *message authentication codes* and this strong notion of security of encryption known as *chosen plaintext attack (CPA) security* where we allow to encrypt *many messages of Eve's choice* and still require that the next message hides all information except for what Eve already knew before.

6.2 Going beyond CPA

It may seem that we have finally nailed down the security definition for encryption. After all, what could be stronger than allowing Eve unfettered access to the encryption function. Clearly an encryption satisfying this property will hide the contents of the message in all practical circumstances. Or will it?



Please stop and play an ominous sound track at this point.

6.2.1 Example: The Wired Equivalence Protocol (WEP)

The WEP is perhaps one of the most inaccurately named protocols of all times. It was invented in 1999 for the purpose of securing Wi-Fi networks so that they would have virtually the same level of security as wired networks, but already early on several security flaws were pointed out. In particular in 2001, Fluhrer, Mantin, and Shamir showed how the RC4 flaws we mentioned in prior lecture can be used to completely break WEP in less than one minute. Yet, the protocol lingered on and for many years after was still the most widely used WiFi encryption protocol as many routers had it as the default option. In 2007 the WEP was blamed for a hack stealing 45 million credit card numbers from T.J. Maxx. In 2012 (after 11 years of attacks!) it was estimated that it is still used in about a quarter of encrypted wireless networks, and in 2014 it was still the default option on many Verizon home routers. (I don't know of more recent surveys.) Here we will talk about a different flaw of WEP that is in fact shared by many other protocols, including the first versions of the secure socket layer (SSL) protocol that is used to protect all encrypted web traffic.

To avoid superfluous details we will consider a highly abstract (and somewhat inaccurate) version of WEP that still demonstrates

our main point. In this protocol Alice (the user) sends to Bob (the access point) an IP packet that she wants routed somewhere on the internet.

Thus we can think of the message Alice sends to Bob as a string $m \in \{0,1\}^\ell$ of the form $m = (m_1, m_2)$ where m_1 is the IP address this packet needs to be routed to and m_2 is the actual message that needs to be delivered. In the WEP protocol, the message that Alice sends to Bob has the form

$E_k(m \| CRC(m))$ (where $\|$ denotes concatenation and $CRC(m)$ is some cyclic redundancy code). The actual encryption WEP used was RC4, but for us it doesn't really matter. What does matter is that the encryption has the form $E_k(m') = pad \oplus m'$ where pad is computed as some function of the key. In particular the attack we will describe works even if we use our stronger CPA secure PRF-based scheme where $pad = f_k(r)$ for some random (or counter) r that is sent out separately.

Now the security of the encryption means that an adversary seeing the ciphertext $c = E_k(m \| CRC(m))$ will not be able to know m , but since this is traveling over the air, the adversary could "spoof" the signal and send a different ciphertext c' to Bob. In particular, if the adversary knows the IP address m_1 that Alice was using (e.g., for example, the adversary can guess that Alice is probably one of the billions of people that visit the website boazbarak.org on a regular basis) then she can XOR the ciphertext with a string of her choosing and hence convert the ciphertext $c = pad \oplus (m_1, m_2, CRC(m_1, m_2))$ into the ciphertext $c' = c \oplus x$ where $x = (x_1, x_2, x_3)$ is computed so that $x_1 \oplus m_1$ is equal to the adversary's own IP address!

So, the adversary doesn't need to decrypt the message- by spoofing the ciphertext she can ensure that Bob (who is an access point, and whose job is to decrypt and then deliver packets) simply delivers it unencrypted straight into her hands. One issue is that Eve modifies m_1 then it is unlikely that the CRC code will still check out, and hence Bob would reject the packet. However, **CRC 32** - the CRC algorithm used by WEP - is *linear* modulo 2, which means that for every pair of strings x_1, x_2 , $CRC(m_1 \oplus x_1, m_2 \oplus m_2) = CRC(m_1, m_2) \oplus CRC(x_1, x_2)$. This means that if the original ciphertext c was an encryption of the message $m = (m_1, m_2, CRC(m_1, m_2))$ then $c' = c \oplus (x_1, 0, CRC(x_1, 0))$ will be an encryption of the message $m' = (m_1 \oplus x_1, m_2, CRC(x_1 \oplus m_1, m_2))$. (Where 0 denotes a string of zeroes of the same length as m_2 , and hence $m_2 \oplus 0 = m_2$.) Therefore by XOR'ing c with $(x_1, 0, CRC(x_1, 0))$, the adversary Mallory can ensure that Bob will deliver the message m_2 to the IP address $m_1 \oplus x_1$ of her

choice (see Fig. 6.1).

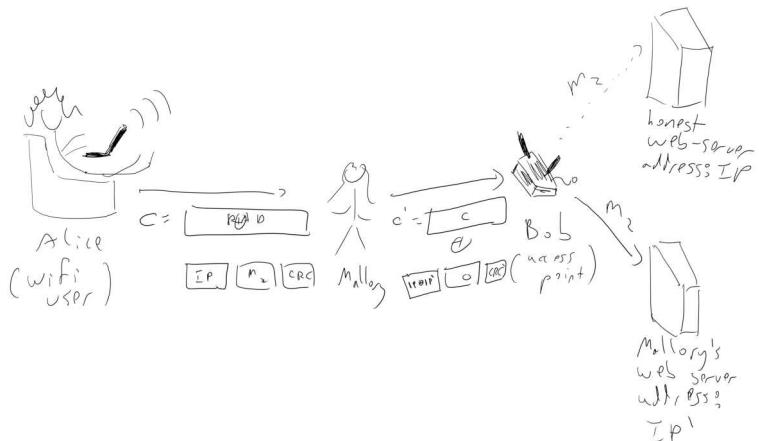


Figure 6.1: The attack on the WEP protocol allowing the adversary Mallory to read encrypted messages even when Alice uses a CPA secure encryption.

6.2.2 Chosen ciphertext security

This is not an isolated example but in fact an instance of a general pattern of many breaks in practical protocols. Some examples of protocols broken through similar means include [XML encryption](#), [IPSec](#) (see also [here](#)) as well as JavaServer Faces, Ruby on Rails, ASP.NET, and the Steam gaming client (see the Wikipedia page on [Padding Oracle Attacks](#)).

The point is that often our adversaries can be *active* and modify the communication between sender and receiver, which in effect gives them access not just to choose *plaintexts* of their choice to encrypt but even to have some impact on the *ciphertexts* that are decrypted. This motivates the following notion of security (see also Fig. 6.2):

Definition 6.1 — CCA security. An encryption scheme (E, D) is *chosen ciphertext attack (CCA) secure* if every efficient adversary *Mallory* wins in the following game with probability at most $1/2 + \text{negl}(n)$:

- * Mallory gets 1^n where n is the length of the key
- * For $\text{poly}(n)$ rounds, Mallory gets access to the functions $m \mapsto E_k(m)$ and $c \mapsto D_k(c)$.
- * Mallory chooses a pair of messages $\{m_0, m_1\}$, a secret b is chosen at random in $\{0, 1\}$, and Mallory gets $c^* = E_k(m_b)$.
- * Mallory now gets another $\text{poly}(n)$ rounds of access to the functions $m \mapsto E_k(m)$ and $c \mapsto D_k(c)$ except that she is not

allowed to query c^* to her second oracle.

* Mallory outputs b' and wins if $b' = b$.

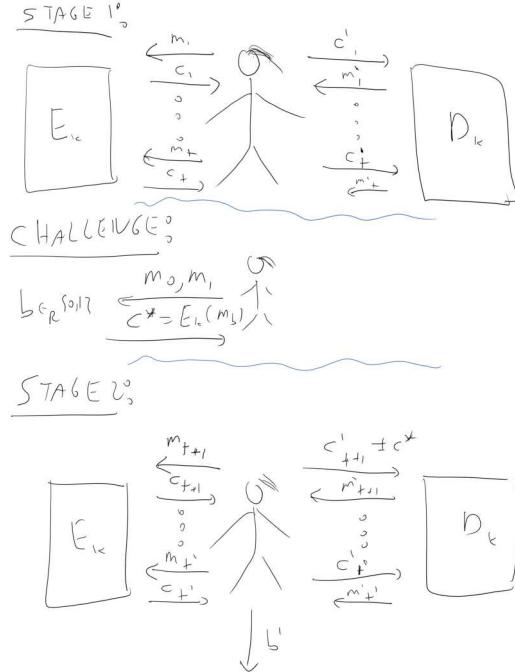


Figure 6.2: the CCA security game

This might seem a rather strange definition so let's try to digest it slowly. Most people, once they understand what the definition says, don't like it that much. There are two natural objections to it:

- **This definition seems to be too strong:** There is no way we would let Mallory play with a *decryption box* - that basically amounts to letting her break the encryption scheme. Sure, she could have some impact on the ciphertexts that Bob decrypts and observe some resulting side effects, but there is a long way from that to giving her oracle access to the decryption algorithm.

The response to this is that it is very hard to model what is the “realistic” information Mallory might get about the ciphertexts she might cause Bob to decrypt. The goal of a security definition is not to capture exactly the attack scenarios that occur in real life but rather to be *sufficiently conservative* so that these real life attacks could be modeled in our game. Therefore, having a too strong definition is not a bad thing (as long as it can be achieved!). The WEP example shows that the definition does capture a practical issue in security and similar attacks on practical protocols have been shown time and again (see for example the discussion of “padding attacks” in Section

3.7.2 of the Katz Lindell book.)

- **This definition seems to be too weak:** What justification do we have for not allowing Mallory to make the query c^* to the decryption box? After all she is an adversary so she could do whatever she wants. The answer is that the definition would be clearly impossible to achieve if Mallory could simply get the decryption of c^* and learn whether it was an encryption of m_0 or m_1 . So this restriction is the absolutely minimal one we could make without causing the notion to be obviously impossible. Perhaps surprisingly, it turns out that once we make this minimal restriction, we can in fact construct CCA-secure encryptions.

What does CCA has to do with WEP? The CCA security game is somewhat strange, and it might not be immediately clear whether it has anything to do with the attack we described on the WEP protocol. However, it turns out that using a CCA secure encryption *would* have prevented that attack. The key is the following claim:

Lemma 6.2 Suppose that (E, D) is a CCA secure encryption, then there is no efficient algorithm that given an encryption c of the plaintext (m_1, m_2) outputs a ciphertext c' that decrypts to (m'_1, m_2) where $m'_1 \neq m_1$.

In particular Lemma 6.2 rules out the attack of transforming c that encrypts a message starting with a some address IP to a ciphertext that starts with a different address IP' . Let us now sketch its proof.

Proof. We'll show that such if we had an adversary M' that violates the conclusion of the claim, then there is an adversary M that can win in the CCA game.

The proof is simple and relies on the crucial fact that the CCA game allows M to query the decryption box on *any* ciphertext of her choice, as long as it's not *exactly identical* to the challenge ciphertext c^* . In particular, if M' is able to morph an encryption c of m to some encryption c' of some different m' that agrees with m on some set of bits, then M can do the following: in the security game, use m_0 to be some random message and m_1 to be this plaintext m . Then, when receiving c^* , apply M' to it to obtain a ciphertext c' (note that if the plaintext differs then the ciphertext must differ also; can you see why?) ask the decryption box to decrypt it and output 1 if the resulting message agrees with m in the corresponding set of bits (otherwise output a random bit). If M' was successful with probability ϵ , then M would win in the CCA game with probability at least $1/2 + \epsilon/10$ or so. ■

P

The proof above is rather sketchy. However it is not very difficult and proving Lemma 6.2 on your own is an excellent way to ensure familiarity with the definition of CCA security.

6.3 Constructing CCA secure encryption

The definition of CCA seems extremely strong, so perhaps it is not surprising that it is useful, but can we actually construct it? The WEP attack shows that the CPA secure encryption we saw before (i.e., $E_k(m) = (r, f_k(r) \oplus m)$) is *not* CCA secure. We will see other examples of *non* CCA secure encryptions in the exercises. So, how *do* we construct such a scheme? The WEP attack actually already hints of the crux of CCA security. We want to ensure that Mallory is not able to modify the challenge ciphertext c^* to some related c' . Another way to say it is that we need to ensure the *integrity* of messages to achieve their *confidentiality* if we want to handle *active* adversaries that might modify messages on the channel. Since in in a great many practical scenarios, an adversary might be able to do so, this is an important message that deserves to be repeated:

To ensure confidentiality, you need integrity.

This is a lesson that has been time and again been shown and many protocols have been broken due to the mistaken belief that if we only care about *secrecy*, it is enough to use only *encryption* (and one that is only CPA secure) and there is no need for *authentication*.

Matthew Green writes this more provocatively as

Nearly all of the symmetric encryption modes you learned about in school, textbooks, and Wikipedia are (potentially) insecure.¹

exactly because these basic modes only ensure security for *passive* eavesdropping adversaries and do not ensure chosen ciphertext security which is the “gold standard” for online applications. (For symmetric encryption people often use the name “authenticated encryption” in practice rather than CCA security; those are not identical but are extremely related notions.)

All of this suggests that Message Authentication Codes might help

¹ I also like the part where Green says about a block cipher mode that “if OCB was your kid, he’d play three sports and be on his way to Harvard.” We will have an exercise about a simplified version of the GCM mode (which perhaps only plays a single sport and is on its way to . . .). You can read about OCB in Exercise 9.14 in the Boneh-Shoup book; it uses the notion of a “tweakable block cipher” which simply means that given a single key k , you actually get a set $\{p_{k,1}, \dots, p_{k,t}\}$ of permutations that are indistinguishable from t independent random permutation (the set $\{1, \dots, t\}$ is called the set of “tweaks” and we sometimes index it using strings instead of numbers).

us get CCA security. This turns out to be the case. But one needs to take some care exactly *how* to use MAC's to get CCA security. At this point, you might want to stop and think how you would do this...



You should stop here and try to think how you would implement a CCA secure encryption by combining MAC's with a CPA secure encryption.

 If you didn't stop before, then you should really stop and think now.

OK, so now that you had a chance to think about this on your own, we will describe one way that works to achieve CCA security from MACs. We will explore other approaches that may or may not work in the exercises.

Theorem 6.3 — CCA from CPA and MAC. Let (E, D) be CPA-secure encryption scheme and (S, V) be a CMA-secure MAC with n bit keys and a canonical verification algorithm.² Then the following encryption (E', D') with keys $2n$ bits is CCA secure:

- * $E'_{k_1, k_2}(m)$ is obtained by computing $c = E_{k_1}(m)$, $\sigma = S_{k_2}(c)$ and outputting (c, σ) .
- * $D'_{k_1, k_2}(c, \sigma)$ outputs nothing (e.g., an error message) if $V_{k_2}(c, \sigma) \neq 1$, and otherwise outputs $D_{k_1}(c)$.

Proof. Suppose, for the sake of contradiction, that there exists an adversary M' that wins the CCA game for the scheme (E', D') with probability at least $1/2 + \epsilon$. We consider the following two cases:

Case I: With probability at least $\epsilon/10$, at some point during the CCA game, M' sends to its decryption box a ciphertext (c, σ) that is not identical to one of the ciphertexts it previously obtained from its decryption box, and obtains from it a non-error response.

Case II: The event above happens with probability smaller than $\epsilon/10$.

We will derive a contradiction in either case. In the first case, we will use M' to obtain an adversary that breaks the MAC (S, V) , while in the second case, we will use M' to obtain an adversary that breaks the CPA-security of (E, D) .

Let's start with Case I: When this case holds, we will build an adversary F (for “forger”) for the MAC (S, V) , we can assume the adversary F has access to the both signing and verification algorithms as black boxes for some unknown key k_2 that is chosen at random and fixed.³ F will choose k_1 on its own, and will also choose at random a number i_0 from 1 to T , where T is the total number of queries that M' makes to the decryption box. F will run the entire CCA game with M' , using k_1 and its access to the black boxes to execute the decryption and decryption boxes, all the way until just before M' makes the i_0^{th} query (c, σ) to its decryption box. At that point, F will output (c, σ) . We claim that with probability at least $\epsilon/(10T)$, our forger will succeed in the CMA game in the sense that **(i)** the query (c, σ) will pass verification, and **(ii)** the message c was

² By a *canonical verification algorithm* we mean that $V_k(m, \sigma) = 1$ iff $S_k(m) = \sigma$.

³ Since we use a MAC with canonical verification, access to the signature algorithm implies access to the verification algorithm.

not previously queried before to the signing oracle.

Indeed, because we are in Case I, with probability $\epsilon/10$, in this game *some* query that M' makes will be one that was not asked before and hence was *not* queried by F to its signing oracle, and moreover, the returned message is not an error message, and hence the signature passes verification. Since i_0 is random, with probability $\epsilon/(10T)$ this query will be at the i_0^{th} round. Let us assume that this above event *GOOD* happened in which the i_0 -th query to the decryption box is a pair (c, σ) that both passes verification and the pair (c, σ) was not returned before by the encryption oracle. Since we pass (canonical) verification, we know that $\sigma = S_{k_2}(c)$, and because all encryption queries return pairs of the form $(c', S_{k_2}(\sigma'))$, this means that no such query returned c as its first element either. In other words, when the event *GOOD* happens the i_0 -the query contains a pair (c, σ) such that c was not queried before to the signature box, but (c, σ) passes verification. This is the definitoin of breaking (S, V) in a chosen message attack, and hence we obtain a contradiction to the CMA security of (S, V) .

Now for Case II: In this case, we will build an adversary *Eve* for CPA-game in the original scheme (E, D) . As you might expect, the adversary *Eve* will choose by herself the key k_2 for the MAC scheme, and attempt to play the CCA security game with M' . When M' makes *encryption queries* this should not be a problem- *Eve* can forward the plaintext m to its encryption oracle to get $c = E_{k_1}(m)$ and then compute $\sigma = S_{k_2}(c)$ since she knows the signing key k_2 .

However, what does *Eve* do when M' makes *decryption queries*? That is, suppose that M' sends a query of the form (c, σ) to its decryption box. To simulate the algorithm D' , *Eve* will need access to a *decryption box* for D , but she doesn't get such a box in the CPA game (This is a subtle point- please pause here and reflect on it until you are sure you understand it!)

To handle this issue *Eve* will follow the common approach of "winging it and hoping for the best". When M' sends a query of the form (c, σ) , *Eve* will first check if it happens to be the case that (c, σ) was returned before as an answer to an encryption query m . In this case *Eve* will breathe a sigh of relief and simply return m to M' as the answer. (This is obviously correct: if (c, σ) is the encryption of m then m is the decryption of (c, σ) .) However, if the query (c, σ) has not been returned before as an answer, then *Eve* is in a bit of a pickle. The way out of it is for her to simply return "error" and hope that everything will work out. The crucial observation is that because we are in case II things *will* work out. After all, the only way *Eve*

makes a mistake is if she returns an error message where the original decryption box would not have done so, but this happens with probability at most $\epsilon/10$. Hence, if M' has success $1/2 + \epsilon$ in the CCA game, then even if it's the case that M' always outputs the wrong answer when *Eve* makes this mistake, we will still get success at least $1/2 + 0.9\epsilon$. Since ϵ is non negligible, this would contradict the CPA security of (E, D) thereby concluding the proof of the theorem. ■

P This proof is emblematic of a general principle for proving CCA security. The idea is to show that the decryption box is completely “useless” for the adversary, since the only way to get a non error response from it is to feed it with a ciphertext that was received from the encryption box.

6.4 (*Simplified*) GCM encryption

The construction above works as a generic construction, but it is somewhat costly in the sense that we need to evaluate both the block cipher and the MAC. In particular, if messages have t blocks, then we would need to invoke two cryptographic operations (a block cipher encryption and a MAC computation) per block. The **GCM (Galois Counter Mode)** is a way around this. We are going to describe a simplified version of this mode. For simplicity, assume that the number of blocks t is fixed and known (though many of the annoying but important details in block cipher modes of operations involve dealing with padding to multiple of blocks and dealing with variable block size).

A **universal hash function collection** is a family of functions $\{h : \{0,1\}^\ell \rightarrow \{0,1\}^n\}$ such that for every $x \neq x' \in \{0,1\}^\ell$, the random variables $h(x)$ and $h(x')$ (taken over the choice of a random h from this family) are pairwise independent in $\{0,1\}^{2n}$. That is, for every two potential outputs $y, y' \in \{0,1\}^n$,

$$\mathbb{P}_h[h(x) = y \wedge h(x') = y'] = 2^{-2n} \quad (6.1)$$

Universal hash functions have rather efficient constructions, and in particular if we relax the definition to allow *almost universal* hash functions (where we replace the 2^{-2n} factor in the righthand side of Eq. (6.1) by a slightly bigger, though still negligible quantity) then the constructions become extremely efficient and the size of the description of h is only related to n , no matter how big ℓ is.⁴

⁴ In ϵ -almost universal hash functions we require that for every $y, y' \in \{0,1\}^n$, and $x \neq x' \in \{0,1\}^\ell$, the probability that $h(x) = h(x')$ is at most ϵ . It can be easily shown that the analysis below extends to ϵ almost universal hash functions as long as ϵ is negligible, but we will leave verifying this to the reader.

Our encryption scheme is defined as follow. The key is (k, h) where k is an index to a pseudorandom permutation $\{p_k\}$ and h is the key for a *universal hash function*.⁵ To encrypt a message $m = (m_1, \dots, m_t) \in \{0, 1\}^{nt}$ do the following:

- Choose IV at random in $[2^n]$.
- Let $z_i = E(k, IV + i)$ for $i = 1, \dots, t + 1$.
- Let $c_i = z_i \oplus m_i$.
- Let $c_{t+1} = h(c_1, \dots, c_t) \oplus z_{t+1}$.
- Output $(IV, c_1, \dots, c_{t+1})$.

The communication overhead includes one additional output block plus the IV (whose transmission can often be avoided or reduced, depending on the settings; see the notion of “nonce based encryption”). This is fairly minimal. The additional computational cost on top of t block-cipher evaluation is the application of $h(\cdot)$. For the particular choice of h used in Galois Counter Mode, this function h can be evaluated very efficiently- at a cost of a single multiplication in the Galois field of size 2^{128} per block (one can think of it as some very particular operation that maps two 128 bit strings to a single one, and can be carried out quite efficiently). We leave it as an (excellent!) exercise to prove that the resulting scheme is CCA secure.

⁵ In practice the key h is derived from the key k by applying the PRP to some particular input.

6.5 Padding, chopping and their pitfalls: the “buffer overflow” of cryptography

In this course we typically focus on the simplest case where messages have a *fixed size*. But in fact, in real life we often need to chop long messages into blocks, or pad messages so that their length becomes an integral multiple of the block size. Moreover, there are several subtle ways to get this wrong, and these have been used in several practical attacks.

Chopping into blocks: A block cipher a-priori provides a way to encrypt a message of length n , but we often have much longer messages and need to “chop” them into blocks. This is where the *block cipher modes* discussed in the previous lecture come in. However, the basic popular modes such as CBC and OFB do *not* provide security against chosen ciphertext attack, and in fact typically make it easy to *extend* a ciphertext with an additional block or to *remove* the last block from a ciphertext, both being operations which should not be feasible in a CCA secure encryption.

Padding: Oftentimes messages are not an integer multiple of the block size and hence need to be *padding*. The *padding* is typically a map that takes the last partial block of the message (i.e., a string m of length in $\{0, \dots, n-1\}$) and maps it into a full block (i.e., a string $m \in \{0,1\}^n$). The map needs to be invertible which in particular means that if the message is already an integer multiple of the block size we will need to add an extra block. (Since we have to map all the $1 + 2 + \dots + 2^{n-1}$ messages of length $1, \dots, n-1$ into the 2^n messages of length n in a one-to-one fashion.) One approach for doing so is to pad an $n' < n$ length message with the string $10^{n-n'-1}$. Sometimes people use a different padding which involves encoding the length of the pad.

6.6 Chosen ciphertext attack as implementing metaphors

The classical “metaphor” for an encryption is a sealed envelope, but as we have seen in the WEP, this metaphor can lead you astray. If you placed a message m in a sealed envelope, you should not be able to modify it to the message $m \oplus m'$ without opening the envelope, and yet this is exactly what happens in the canonical CPA secure encryption $E_k(m) = (r, f_k(r) \oplus m)$. CCA security comes much closer to realizing the metaphor, and hence is considered as the “gold standard” of secure encryption. This is important even if you do not intend to write poetry about encryption. *Formal verification* of computer programs is an area that is growing in importance given that computer programs become both more complex and more mission critical. Cryptographic protocols can fail in subtle ways, and even published proofs of security can turn out to have bugs in them. Hence there is a line of research dedicated to finding ways to *automatically* prove security of cryptographic protocols. Much of these line of research is based on simple models to describe protocols that are known as *Dolev Yao models*, based on the first paper that proposed such models. These models define an *algebraic* form of security, where rather than thinking of messages, keys, and ciphertexts as binary string, we think of them as abstract entities. There are certain rules for manipulating these symbols. For example, given a key k and a message m you can create the ciphertext $\{m\}_k$, which you can decrypt back to m using the same key. However the assumption is that any information that cannot be obtained by such manipulation is unknown.

Translating a proof of security in this algebra to proof for real world adversaries is highly non trivial. However, to have even a

fighting a chance, the encryption scheme needs to be as strong as possible, and in particular it turns out that security notions such as CCA play a crucial role.

7

Hash functions and random oracles

We have seen pseudorandom generators, functions and permutations, as well as Message Authentication codes, CPA and CCA secure encryptions. This week we will talk about cryptographic hash functions and some of their magical properties. We motivate this by the *bitcoin* cryptocurrency. As usual our discussion will be highly abstract and idealized, and any resemblance to real cryptocurrencies, living or dead, is purely coincidental.

7.1 The “bitcoin” problem

Using cryptography to create a *centralized* digital-currency is fairly straightforward, and indeed this is what is done by Visa, Mastercard etc.. The main challenge with bitcoin is that it is *decentralized*. There is no trusted server, there are no “user accounts”, no central authority to adjudicate claims. Rather we have a collection of anonymous and autonomous parties that somehow need to agree on what is a valid payment.

7.1.1 The currency problem

Before talking about cryptocurrencies, let’s talk about currencies in general.¹ At an abstract level, a *currency* requires two components:

- A scarce resource
- A mechanism for determining and transferring *ownership* of certain quantities of this resource.

The original currencies were based on **commodity money**. The scarce resource was some commodity having intrinsic value, such as

¹ I am not an economist by any stretch of the imagination, so please take the discussion below with a huge grain of salt. I would appreciate any comments on it.

gold or silver, or even salt or tea, and ownership based on physical possession. However, as commerce increased, carrying around (and protecting) the large quantity of the commodities became impractical, and societies shifted to **representative money**, where the currency is not the commodity itself but rather a certificate that provides the right to the commodity. Representative money requires trust in some central authority that would respect the certificate. The next step in the evolution of currencies was **fiat money**, which is a currency (like today's dollar, ever since the U.S. moved off the **gold standard**) that does not correspond to any commodity, but rather only relies on trust in a central authority. (Another example is the Roman coins, which though originally made of silver, have undergone a continuous process of **debasement** until they contained less than two percent of it.) One advantage (sometimes disadvantage) of a fiat currency is that it allows for more flexible monetary policy on parts of the central authority.

7.1.2 Bitcoin architecture

Bitcoin is a fiat currency without a central authority. A priori this seems like a contradiction in terms. If there is no trusted central authority, how can we ensure a scarce resource? who settles claims of ownership? and who sets monetary policy? Bitcoin (and other cryptocurrencies) is about the solution for these problems via cryptographic means.

The basic unit in the bitcoin system is a *coin*. Each coin has a unique identifier, and a current *owner*.² Transactions in the system have either the form of “mint coin with identifier ID and owner P ” or “transfer the coin ID from P to Q ”. All of these transactions are recorded in a public *ledger*.

Since there are no user accounts in bitcoin, the “entities” P and Q are not identifiers of any person or account. Rather P and Q are “computational puzzles”. A *computational puzzle* can be thought of as a string α that specifies some “problem” such that it’s easy to verify whether some other string β is a “solution” for α , but it is hard to find such a solution on your own. (Students with complexity background will recognize here the class **NP**.) So when we say “transfer the coin ID from P to Q ” we mean that whomever holds a solution for the puzzle Q is now the owner of the coin ID (and to verify the authenticity of this transfer, you provide a solution to the puzzle P .) More accurately, a transaction involving the coin ID is self-validating if it contains a solution to the puzzle that is associated with ID ac-

² This is one of the places where we simplify and deviate from the actual Bitcoin system. In the actual Bitcoin system, the atomic unit is known as a *satoshi* and one bitcoin (abbreviated BTC) is 10^8 satoshis. For reasons of efficiency, there is no individual identifier per satoshi and transactions can involve transfer and creation of multiple satoshis. However, conceptually we can think of atomic coins each of which has a unique identifier.

cording to the latest transaction in the ledger.

 Please re-read the previous paragraph, to make sure you follow the logic.

One example of a puzzle is that α can encode some large integer N , and a solution β will encode a pair of numbers A, B such that $N = A \cdot B$. Another more generic example (that you can keep in mind as a potential implementation for the puzzles we use here) is that α will be a string in $\{0, 1\}^{2n}$ while β will be a string in $\{0, 1\}^n$ such that $\alpha = G(\beta)$ where $G : \{0, 1\}^n \rightarrow \{0, 1\}^{2n}$ is some pseudorandom generator. The real Bitcoin system typically uses puzzles based on *digital signatures*, a concept we will learn about later in this course, but you can simply think of P as specifying some abstract puzzle and every person that can solve P can perform transactions on the coins owned by P .³ In particular if you lost the solution to the puzzle then you have no access to the coin, and if someone stole the solution from you, then you have no recourse or way to get your coin back. People have managed to **lose millions of dollars** in this way.

7.2 The bitcoin ledger

The main idea behind bitcoin is that there is a public *ledger* that contains an ordered list of all the transactions that were ever performed and are considered as valid in the system. Given such a ledger, it is easy to answer the question of who owns any particular coin. The main problem is how does a collection of anonymous parties without any central authority agree on this ledger? This is an instance of the *consensus* problem in distributed computing. This seems quite scary, as there are very strong negative results known for this problem; for example the famous **Fischer, Lynch Patterson (FLP) result** showed that if there is even one party that has a *benign* failure (i.e., it halts and stop responding) then it is impossible to guarantee consensus in an asynchronous network. Things are better if we assume synchronicity (i.e., a global clock and some bounds on the latency of messages) as well as that a majority or supermajority of the parties behave correctly. The central clock assumption is typically approximately maintained on the Internet, but the honest majority assumption seems quite suspicious. What does it mean a “majority of parties” in an anonymous network where a single person can create multiple “entities” and cause them to behave arbitrarily (known as “byzantine” faults in distributed parlance)? Also, why would we

³ There are reasons why Bitcoin uses digital signatures and not these puzzles. The main issue is that we want to bind the puzzle not just to the coin but also to the particular transaction, so that if you know the solution to the puzzle P corresponding to the coin ID and want to use that to transfer it to Q , it won't be possible for someone to take your solution and use that to transfer the coin to Q' before your transaction is added to the public ledger. We will come back to this issue after we learn about digital signatures.

assume that even one party would behave honestly- if there is no central authority and it is profitable to cheat then they everyone would cheat, wouldn't they?

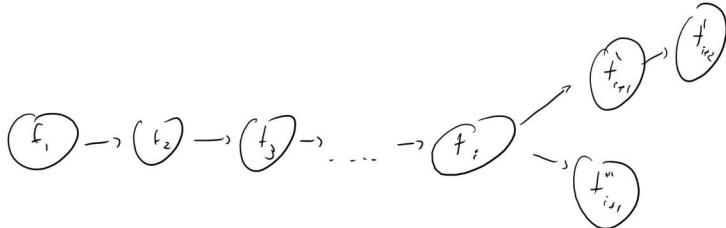


Figure 7.1: The bitcoin ledger consists of an ordered list of transactions. At any given point in time there might be several “forks” that continue the ledger, and different parties do not necessarily have to agree on them. However, the bitcoin architecture is designed to ensure that the parties corresponding to a majority of the computing power will reach consensus on a single ledger.

Perhaps the main idea behind bitcoin is that “majority” will correspond to a “majority of computing power”, or as the [original bitcoin paper](#) says, “one CPU one vote” (or perhaps more accurately, “one cycle one vote”). It might not be immediately clear how to implement this, but at least it means that creating fictitious new entities (sometimes known as a [Sybill attack](#) after the movie about multiple-personality disorder) cannot help. To implement it we turn to a cryptographic concept known as “proof of work” which was originally suggested by Dwork and Naor in 1991 as a way to combat mass marketing email.⁴

Consider a pseudorandom function $\{f_k\}$ mapping n bits to ℓ bits. On average, it will take a party Alice 2^ℓ queries to obtain an input x such that $f_k(x) = 0^\ell$. So, if we’re not too careful, we might think of such an input x as a *proof* that Alice spent 2^ℓ time.



Stop here and try to think if indeed it is the case that one cannot find an input x such that $f_k(x) = 0^\ell$ using much fewer than 2^ℓ steps.

⁴ This was a rather visionary paper in that it foresaw this issue before the term “spam” was introduced and indeed when email itself, let alone spam email, was hardly widespread.

The main question in using PRF’s for proofs of work is who is holding the key k for the pseudorandom function. If there is a trusted server holding the key, then sure, finding such an input x would take on average 2^ℓ queries, but the whole point of bitcoin is to *not* have a trusted server. If we give k to a party Alice, then can we guarantee that she can’t find a “shortcut” to find such an input without running 2^ℓ queries? The answer, in general, is **no**.

P Indeed, it is an excellent exercise to prove that (under the PRF conjecture) that there exists a PRF $\{f_k\}$ mapping n bits to n bits and an efficient algorithm A such that $A(k) = x$ such that $f_k(x) = 0^\ell$.

However, suppose that $\{f_k\}$ was somehow a “super-strong PRF” that would behave like a random function *even to a party that holds the key*. In this case, we can imagine that making a query to f_k corresponds to tossing ℓ independent random coins, and it would not be feasible to obtain x such that $f_k(x) = 0^\ell$ using much less than 2^ℓ cycles. Thus presenting such an input x can serve as a “proof of work” that you’ve spent 2^ℓ cycles or so. By adjusting ℓ we can obtain a proof of spending T cycles for a value T of our choice. Now if things would go as usual in this course then I would state a result like the following:

Theorem: Under the PRG conjecture, there exist super strong PRF.

Unfortunately such a result is *not* known to be true, and for a very good reason. Most natural ways to define “super strong PRF” will result in properties that can be shown to be *impossible to achieve*. Nevertheless, the intuition behind it still seems useful and so we have the following heuristic:

The random oracle heuristic (aka “Random oracle model”, Bellare-Rogaway 1993): If a “natural” protocol is secure when all parties have access to a random function $H : \{0,1\}^n \rightarrow \{0,1\}^\ell$, then it remains secure even when we give the parties the *description* of a cryptographic hash function with the same input and output lengths.

We don’t have a good characterization as to what makes a protocol “natural” and we do have fairly strong counterexamples to this heuristic (though they are arguably “unnatural”). That said, it still seems useful as a way to get intuition for security, and in particular to analyze bitcoin (and many other practical protocols) we do need to assume it, at least given current knowledge.

R **Important caveat on the random oracle model** The random oracle heuristic is very different from all the conjectures we considered before. It is **not** a formal

conjecture since we don't have any good way to define "natural" and we do have examples of protocols that are secure when all parties have access to a random function but are **insecure** whenever we replace this random function by **any** efficiently computable function (see the homework exercises).

We can now specify the "proof of work" protocol for bitcoin. Given some identifier $ID \in \{0,1\}^n$, an integer $T \ll 2^n$, and a hash function $H : \{0,1\}^{2n} \rightarrow \{0,1\}^n$, the proof of work corresponding to ID and T will be some $x \in \{0,1\}^*$ such that the first $\lceil \log T \rceil$ bits of $H(ID\|x)$ are zero.⁵

7.2.1 From proof of work to consensus on ledger

How does proof of work help us in achieving consensus? The idea is that every transaction t_i comes up with a proof of work of some T_i time with respect to some identifier that is unique to t_i . The *length* of a ledger (t_1, \dots, t_n) is the sum of the corresponding T_i 's which correspond to the total number of cycles invested in creating this ledger.

An honest party in the bitcoin network will accept the longest valid ledger it is aware of. (A ledger is *valid* if every transaction in it of the form "transfer the coin ID from P to Q " is self-certified by a solution of P , and the last transaction in the ledger involving ID either transferred or minted the coin ID to P). If a ledger L corresponds to the majority of the cycles that were available in this network then every honest party would accept it, as any alternative ledger would be necessarily shorter. (See Fig. 7.1.)

The question is then how do we get to that happy state given that many parties might be non-malicious but still *selfish* and might not want to volunteer their computing power for the goal of creating a consensus ledger. Bitcoin achieves this by giving some incentive, in the form of the ability to mint new coins, to any party that adds to the ledger. This means that if we are already in the situation where there is a consensus ledger L , then every party has an interest in continuing this ledger L , and not any alternative, as they want their minting transaction to be part of the new consensus ledger. In contrast if they "fork" the consensus ledger then their work may well be for vain. Thus one can hope that the consensus ledger will continue to grow. (This is a rather hand-wavy and imprecise argument, see [this paper](#) for a more in depth analysis; this is also related to the phenomenon known as [preferential attachment](#).)

⁵ The actual bitcoin protocol is slightly more general, where the proof is some x such that $H(ID\|x)$, when interpreted as a number in $[2^n]$, is at most T . There are also other issues about how exactly x is placed and ID is computed from past history that we ignore here.

Cost to mine, mining pools: Generally, if you know that completing a T -cycle proof will get you a single coin, then making a single query (which will succeed with probability $1/T$) is akin to buying a lottery ticket that costs you a single cycle and has probability $1/T$ to win a single coin. One difference over the actual lottery is that there is also some probability that you’re working on the wrong fork of the ledger, but this incentivizes people to avoid this as much as possible. Another, perhaps even more major difference, is that things are setup so that this is a *profitable* enterprise and the cost of a cycle is smaller than the value of $1/T$ coins. Just like in the lottery, people can and do gather in groups (known as “mining pools”) where they pool together all their computing resources, and then split the award if they win it. Joining a pool doesn’t change your expectation of winning but reduces the *variance*. In the extreme case, if everyone is in the same pool, then for every cycle you spend you get exactly $1/T$ coins. The way these pools work in practice is that someone that spent C cycles looking for an output with all zeroes, only has probability C/T of getting it, but is very likely to get an output that begins with $\log C$ zeroes. This output can serve as their own “proof of work” that they spent C cycles and they can send it to the pool management so they get an appropriate share of the reward.

The real bitcoin: There are several aspects in which the protocol described above differs from the real bitcoin protocol. Some of them were already discussed above: Bitcoin typically uses digital signatures for puzzles (though it has a more general scripting language to specify them), and transactions involve a number of satoshis (and the user interface typically displayes currency in units of BTC which are 10^8 satoshis). The Bitcoin protocol also has a formula designed to factor in the decrease in dollar cost per cycle so that bitcoins become more expensive to mine with time. There is also a fee mechanism apart from the mining to incentivize parties to add to the ledger. (The issue of incentives in bitcoin is quite subtle and not fully resolved, and it is possible that parties’ behavior will change with time.) The ledger does not grow by a single transaction at a time but rather by a *block* of transactions, and there is also some timing synchronization mechanism (which is needed, as per the consensus impossibility results). There are other differences as well; see the [Bonneau et al paper](#) as well as the [Tschorisch and Scheuermann survey](#) for more.

7.3 Collision resistance hash functions and creating short “unique” identifiers

Another issue we “brushed under the carpet” is how do we come up with these unique identifiers per transaction. We want each transaction t_i to be *bound* to the ledger state (t_1, \dots, t_{i-1}) , and so the ID of t_i should contain also the ID’s all the prior transactions. But yet we want this ID to be only n bits long. Ideally, we could solve this if we had a *one to one* mapping H from $\{0,1\}^N$ to $\{0,1\}^n$ for some very large $N \gg n$. Then the ID corresponding to the task of appending t_i to (t_1, \dots, t_{i-1}) would simply be $H(t_1 \parallel \dots \parallel t_i)$. The only problem is that this is of course clearly impossible- 2^N is *much* bigger than 2^n and there is no one to one map from a large set to a smaller set. Luckily we are in the magical world of crypto where the impossible is routine and the unimaginable is occasional. So, we can actually find a function H that is “essentially” one to one.

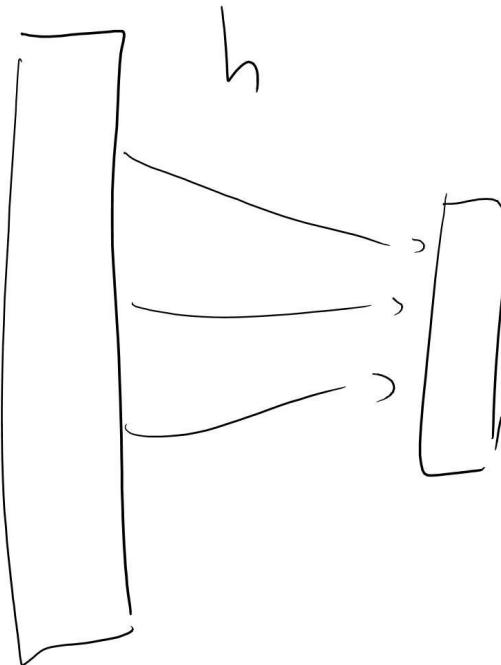


Figure 7.2: A collision-resistant hash function is a map that from a large universe to a small one that is “practically one to one” in the sense that collisions for the function do exist but are hard to find.

The main idea is the following simple result, which can be thought of as one side of the so called “**birthday paradox**”:

Lemma 7.1 If H is a random function from some domain S to $\{0,1\}^n$, then the probability that after T queries an attacker finds $x \neq x'$ such that $H(x) = H(x')$ is at most $T^2/2^n$.

Proof. Let us think of H in the “lazy evaluation” mode where for every query the adversary makes, we choose a random answer in $\{0,1\}^n$ at the time it is made. (We can assume the adversary never makes the same query twice since a repeat query can be simulated by repeating the same answer.) For $i < j$ in $[T]$ let $E_{i,j}$ be the event that $H(x_i) = H(x_j)$. Since $H(x_j)$ is chosen at random and independently from the prior choice of $H(x_i)$, the probability of $E_{i,j}$ is 2^{-n} . Thus the probability of the union of $E_{i,j}$ over all i, j 's is less than $T^2/2^n$, but this probability is exactly what we needed to calculate. ■

This means that a random function H is *collision resistant* in the sense that it is hard for an efficient adversary to find two inputs that collide. Thus the random oracle heuristic would suggest that a cryptographic hash function can be used to obtain the following object:

Definition 7.2 — Collision resistant hash functions. A collection $\{h_k\}$ of functions where $h_k : \{0,1\}^* \rightarrow \{0,1\}^n$ for $k \in \{0,1\}^n$ is a *collision resistant hash function (CRH) collection* if the map $(k, x) \mapsto h_k(x)$ is efficiently computable and for every efficient adversary A , the probability over k that $A(k) = (x, x')$ such that $x \neq x'$ and $h_k(x) = h_k(x')$ is negligible.⁶

Once more we do *not* know a theorem saying that under the PRG conjecture there exists a collision resistant hash function collection, even though this property is considered as one of the desiderata for cryptographic hash functions. However, we do know how to obtain collections satisfying this condition under various assumptions that we will see later in the course such as the learning with error problem and the factoring and discrete logarithm problems. Furthermore if we consider the weaker notion of security under a *second preimage attack* (also known as being a “universal one way hash function” or UOWHF) then it is known how to derive such a function from the PRG assumption.

⁶ Note that the other side of the birthday bound shows that you can always find a collision in h_k using roughly $2^{n/2}$ queries. For this reason we typically need to double the output length of hash functions compared to the key size of other cryptographic primitives (e.g., 256 bits as opposed to 128 bits).



CRH vs PRF A collection $\{h_k\}$ of collision resistant hash functions is an incomparable object to a collection $\{f_s\}$ of pseudorandom functions with the same input and output lengths. On one hand, the condition of being collision-resistant does not imply that h_k is indistinguishable from random. For example, it is possible to construct a valid collision resistant hash function where the first output bit always equals zero (and hence is easily distinguishable from a random function). On the other hand, unlike

Definition 4.1, the adversary of **Definition 7.2** is not merely given a “black box” to compute the hash function, but rather the key to the hash function. This is a much stronger attack model, and so a PRF does not have to be collision resistant. (Constructing a PRF that is not collision resistant is a nice and recommended exercise.)

7.4 Practical constructions of cryptographic hash functions

While we discussed hash functions as *keyed* collections, in practice people often think of a hash function as being a *fixed keyless function*. However, this is because most practical constructions involve some hardwired standardized constants (often known as IV) that can be thought of as a choice of the key.

Practical constructions of cryptographic hash functions start with a basic block which is known as a *compression function* h : $\{0,1\}^{2n} \rightarrow \{0,1\}^n$. The function $H : \{0,1\}^* \rightarrow \{0,1\}^n$ is defined as $H(m_1, \dots, m_t) = h(h(h(m_1, \text{IV}), m_2), \dots, m_t)$ when the message is composed of t blocks (and we can pad it otherwise). See [Fig. 7.3](#). This construction is known as the Merkle-Damgard construction and we know that it does preserve collision resistance:

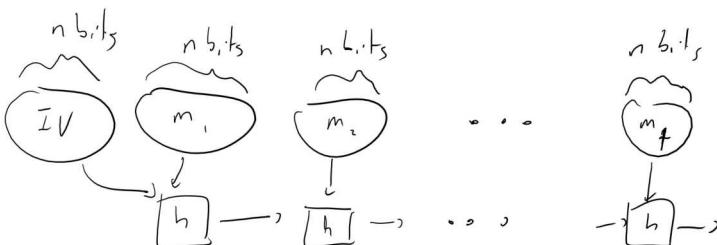


Figure 7.3: The Merkle-Damgard construction converts a compression function $h : \{0,1\}^{2n} \rightarrow \{0,1\}^n$ into a hash function that maps strings of arbitrary length into $\{0,1\}^n$. The transformation preserves collision resistance but does not yield a PRF even if h was pseudorandom. Hence for many applications it should not be used directly but rather composed with a transformation such as HMAC.

Theorem 7.3 — Merkle-Damgard preserves collision resistance. Let H be constructed from h as above. Then given two messages $m \neq m' \in \{0,1\}^{tn}$ such that $H(m) = H(m')$ we can efficiently find two messages $x \neq x' \in \{0,1\}^{2n}$ such that $h(x) = h(x')$.

Proof. The intuition behind the proof is that if h was invertible then we could invert H by simply going backwards. Thus in principle if a collision for H exists then so does a collision for h . Now of course

this is a vacuous statement since both h and H shrink their inputs and hence clearly have collisions. But we want to show a *constructive* proof for this statement that will allow us to transform a collision in H to a collision in h . This is very simple. We look at the computation of $H(m)$ and $H(m')$ and at the first block in which the inputs differ but the output is the same (there must be such a block). This block will yield a collision for h . ■

7.4.1 Practical random-ish functions

In practice we want much more than collision resistance from our hash functions. In particular we often would like them to be PRF's as well. Unfortunately, the Merkle-Damgård construction is *not* a PRF even when IV is random and secret. This is because we can perform a *length extension attack* on it. Even if we don't know IV , given $y = H_{IV}(m_1, \dots, m_t)$ and a block m_{t+1} we can compute $y' = h(y, m_{t+1})$ which equals $H_{IV}(m_1, \dots, m_{t+1})$.

One fix for this is to use a different IV' in the *end* of the encryption. That is, we define:

$$H_{IV, IV'}(m_1, \dots, m_t) = h(IV', H_{IV}(m_1, \dots, m_t))$$

A variant of this construction (where IV' is obtained as some simple function of IV) is known as HMAC and it can be shown to be a pseudorandom function under some pseudorandomness assumptions on the compression function h . It is very widely implemented. In many cases where I say "use a cryptographic hash function" in this course I actually mean to use an HMAC like construction that can be conjectured to give at least a PRF if not stronger "random oracle"-like properties.

The simplest implementation for a compression function is to take a *block cipher* with an n bit key and an n bit message and then simply define $h(x_1, \dots, x_{2n}) = E_{x_{n+1}, \dots, x_{2n}}(x_1, \dots, x_n)$. A more common variant is known as Davies-Meyer where we also XOR the output with x_{n+1}, \dots, x_{2n} . In practice people often use *tailor made* block ciphers that are designed for some efficiency or security concerns.

7.4.2 Some history

Almost all practically used hash functions are based on the Merkle-Damgård paradigm. Hash functions are designed to be extremely efficient⁷ which also means that they are often at the "edge of insecurity" and indeed have fallen over the edge.

⁷ For example, the Boneh-Shoup book quotes processing times of up to 255MB/sec on a 1.83 Ghz Intel Core 2 processor, which is more than enough to handle not just Harvard's network but even Lamar College's.

In 1990 Ron Rivest proposed MD4, which was already shown weaknesses in 1991, and a full collision has been found in 1995. Even faster attacks have been since found and MD4 is considered completely insecure.

In response to these weaknesses, Rivest designed MD5 in 1991. A weakness was shown for it in 1996 and a full collision was shown in 2004. Hence it is now also considered insecure.

In 1993 the National Institute of Standards proposed a standard for a hash function known as the *Secure Hash Algorithm (SHA)*, which has quite a few similarities with the MD4 and MD5 functions. This function is known as SHA-0, and the standard was replaced in 1995 with SHA-1 that includes an extra “mixing” (i.e., bit rotation) operation. At the time no explanation was given for this change but SHA-0 was later found to be insecure. In 2002 a variant with longer output, known as SHA-256, was added (as well as some others). In 2005, following the MD5 collision, significant weaknesses were shown in SHA-1. In 2017, a **full SHA-1 collision was found**. Today SHA-1 is considered insecure and SHA-256 is recommended.

Given the weaknesses in MD-5 and SHA-1, NIST started in 2006 a competition for a new hashing standard, based on functions that seem sufficiently different from the MD5/SHA-0/SHA-1 family. (SHA-256 is unbroken but it seems too close for comfort to those other systems.) The hash function Keccak was selected as the new standard **SHA-3** in August of 2015.

7.4.3 The NSA and hash functions.

The NSA is the world’s largest employer of mathematicians, and is very heavily invested in cryptographic research. It seems quite possible that they devote far more resources to analyzing symmetric primitives such as block ciphers and hash functions than the open research community. Indeed, the history above suggests that the NSA has consistently discovered attacks on hash functions before the cryptographic community (and the same holds for the differential cryptanalysis technique for block ciphers). That said, despite the “mythic” powers that are sometimes ascribed to the NSA, this history suggests that they are ahead of the open community but not so much ahead, discovering attacks on hash functions about 5 years or so ahead.

There are a few ways we can get “insider views” to the NSA’s thinking. Some such insights can be obtained from the Snowden

documents. The [Flame malware](#) has been discovered in Iran in 2012 after operating since at least 2010. It used an MD5 collision to achieve its goals. Such a collision was known in the open literature since 2008, but Flame used a different variant that was unknown in the literature. For this reason it is suspected that it was designed by a western intelligence agency.

Another insight into NSA's thoughts can be found in pages 12-19 of NSA's internal [Cryptolog magazine](#) which has been recently declassified; one can find there a rather entertaining and opinionated (or obnoxious, depending on your point of view) review of the CRYPTO 1992 conference. In page 14 the author remarks that certain weaknesses of MD5 demonstrated in the conference are unlikely to be extended to the full version, which suggests that the NSA (or at least the author) was not aware of the MD5 collisions at the time.

7.4.4 *Cryptographic vs non-cryptographic hash functions:*

Hash functions are of course also widely used for *non-cryptographic* applications such as building hash tables and load balancing. For these applications people often use *linear* hash functions known as *cyclic redundancy codes (CRC)*. Note however that even in those seemingly non-cryptographic applications, an adversary might cause significant slowdown to the system if he can generate many collisions. This can and [has](#) been used to obtain denial of service attacks. As a rule of thumb, if the inputs to your system might be generated by someone who does not have your best interests at heart, you're better off using a cryptographic hash function.

8

Key derivation, protecting passwords, slow hashes, Merkle trees

Last lecture we saw the notion of cryptographic hash functions which are functions that behave like a random function, even in settings (unlike that of standard PRFs) where the adversary has access to the key that allows them to evaluate the hash function. Hash functions have found a variety of uses in cryptography, and in this lecture we survey some of their other applications. In some of these cases, we only need the relatively mild and well-defined property of *collision resistance* while in others we only know how to analyze security under the stronger (and not precisely well defined) *random oracle heuristic*.

8.1 Keys from passwords

We have seen great cryptographic tools, including PRFs, MACs, and CCA secure encryptions, that Alice and Bob can use when they share a cryptographic key of 128 bits or so. But unfortunately, many of the current users of cryptography are *humans* which, generally speaking, have extremely faulty memory capacity for storing large numbers. There are $62^8 \approx 2^{48}$ ways to select a password of 8 upper and lower case letters + numbers, but some letter/numbers combinations end up being chosen much more frequently than others. Due to several large scale hacks, very large databases of passwords have been made public, and one **estimate** is that 90 percent of the passwords chosen by users are contained in a list of about $10,000 \approx 2^{14}$ strings.

If we choose a password at random from some set D then the *entropy* of the password is simply $\log_2 |D|$. However, estimating the entropy of real life passwords is rather difficult. For example,

suppose that I use the winning Massachussets Mega-Lottery numbers as my password. A priory, my password consists of 5 numbers between 1 till 75 and so its entropy is $\log_2(75^5) \approx 31$. However, if an attacker *knew* that I did this, the entropy might be something like $\log(520) \approx 9$ (since there were only 520 such numbers selected in the last 10 years). Moreover, if they knew exactly what draw I based my password on, then they would it exactly and hence the entropy (from their point of view) would be zero. This is worthwhile to emphasize:

The entropy of a secret is always measured with respect to the attacker's point of view.

The exact security of passwords is of course a matter of intense practical interest, but we will simply model the password as being chosen at random from some set $D \subseteq \{0, 1\}^n$ (which is sometimes called the “dictionary”). The set D is known to the attacker, but she has no information on the particular choice of the password.

Much of the challenge for using passwords securely relies on the distinction between *offline* and *online* attacks. If each guess for a password requires interacting *online* with a server, as is the case when typing a PIN number in the ATM, then even a weak password (such as a 4 digit PIN that at best provides 13 bits of entropy) can yield meaningful security guarantees, as typically an alarm would be raised after five or so failed attempts.

However, if the adversary has the ability to check *offline* whether a password is correct then the number of guesses they can try can be as high as the number of computing cycles at their disposal, which can easily run into the billions and so break passwords of 30 or more bits of entropy. (This is an issue we'll return to after we learn about *public key cryptography* when we'll talk about *password authenticated key exchange*.)

Consider a password manager application. In such an application, a user typically chooses a *master password* p_{master} which she can then use to access all her other passwords p_1, \dots, p_t . To enable her to do so without requiring online access to a server, the master password p_{master} is used to *encrypt* the other passwords. However to do that, we need to derive a key k_{master} from the password.



A natural approach is to simply let the key be the password. For example, if the password p is a string of at most 16 bytes, then we can simply treat it as a 128 bit key and use it for encryption. Stop and think

why this would *not* be a good idea. In particular think of an example of a secure encryption (E, D) and a distribution P over $\{0, 1\}^n$ of entropy at least $n/2$ such that if the key k is chosen at random from P then the encryption will be completely insecure.

A classical approach is to simply use a cryptographic hash function $H : \{0, 1\}^* \rightarrow \{0, 1\}^n$, and let $k_{\text{master}} = H(p_{\text{master}})$. If we think of H as a random oracle and p_{master} as chosen randomly from D , then as long as an attacker makes $\ll |D|$ queries to the oracle, they are unlikely to make the query p_{master} and hence the value k_{master} will be completely random from their point of view.

However, since $|D|$ is not too large, it might not be so hard to perform such $|D|$ queries. For this reason, people typically use a *deliberately slow hash* function as a *key derivation function*. The rationale is that the honest user only needs to evaluate H once, and so could afford for it to take a while, while the adversary would need to evaluate it $|D|$ times. For example, if $|D|$ is about 100,000 and the honest user is willing to spend 1 cent of computation resources every time they need to derive k_{master} from p_{master} , then we could set $H(\cdot)$ so that it costs 1 cent to evaluate it and hence on average it will cost the adversary 1,000 dollars to recover it.

There are several approaches for trying to make H deliberately “slow” or “costly” to evaluate but the most popular and simplest one is to simply let H be obtained by iterating many times a basic hash function such as SHA-256. That is, $H(x) = h(h(h(\dots h(x))))$ where h is some standard (“fast”) cryptographic hash function and the number of iterations is tailored to be the largest one that the honest users can tolerate.¹

In fact, typically we will set $k_{\text{master}} = H(p_{\text{master}} \| r)$ where r is a long random but *public* string known as a “salt” (see Fig. 8.1). Including such a “salt” can be important to foiling an adversary’s attempts to amortize the computation costs, see the exercises.

Even when we don’t use one password to encrypt others, it is generally considered the best practice to *never* store a password in the clear but always in this “slow hashed and salted” form, so if the passwords file falls to the hands of an adversary it will be expensive to recover them.

¹ Since CPU speeds can vary quite radically and attackers might even use special-purpose hardware to evaluate iterated hash functions quickly, Abadi, Burrows, Manasse, and Wobber suggested in 2003 to use *memory bound* functions as an alternative approach, where these are functions $H(\cdot)$ designed so that evaluating them will consume at least T bits of memory for some large T . See also the followup paper of Dwork, Goldberg and Naor. This approach has also been used in some practical key derivation functions such as `scrypt` and `Argon2`.

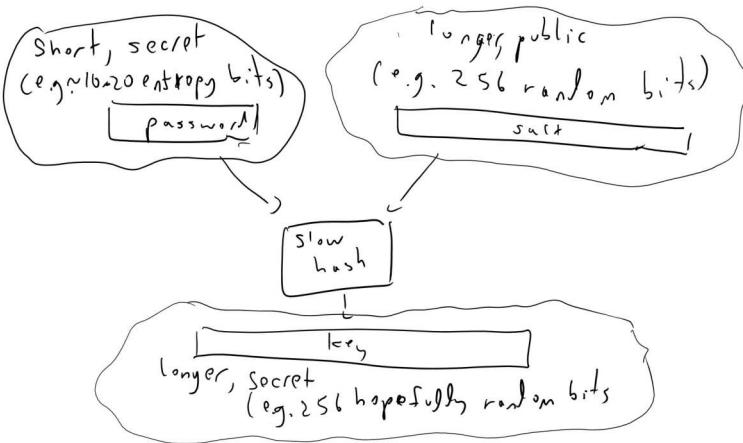


Figure 8.1: To obtain a key from a password we will typically use a “slow” hash function to map the password and a unique-to-user public “salt” value to a cryptographic key. Even with such a procedure, the resulting key cannot be considered as secure and unpredictable as a key that was chosen truly at random, especially if we are in a setting where an adversary can launch an *offline* attack to guess all possibilities.

8.2 Merkle trees and verifying storage.

Suppose that you outsource to the cloud storing your huge data file $x \in \{0,1\}^N$. You now need the i^{th} bit of that file and ask the cloud for x_i . How can you tell that you actually received the correct bit?

Ralph Merkle came up in 1979 with a clever solution, which is known as “Merkle hash trees”. The idea is the following (see Fig. 8.2): suppose we have a collision-resistant hash function $h : \{0,1\}^{2n} \rightarrow \{0,1\}^n$, and think of the string x as composed of t blocks of size n . We then hash every pair of consecutive blocks to transform x into a string x_1 of $t/2$ blocks, and continue in this way for $\log t$ steps until we get a single block $y \in \{0,1\}^n$. (Assume here t is a power of two for simplicity, though it doesn’t make much difference.)

Alice who sends x to the cloud Bob will keep the short block y . Whenever Alice queries the value i she will ask for a *certificate* that x_i is indeed the right value. This certificate will consist of the block that contains i , as well as all of the $2 \log t$ blocks that were used in the hash from this block to the root. The security of this scheme follows from the following simple theorem:

Theorem 8.1 — Merkle Tree security. Suppose that π is a valid certificate that $x_i = b$, then either this statement is true, or one can efficiently extract from π and x two inputs $z \neq z'$ in $\{0,1\}^{2n}$ such that $h(z) = h(z')$.

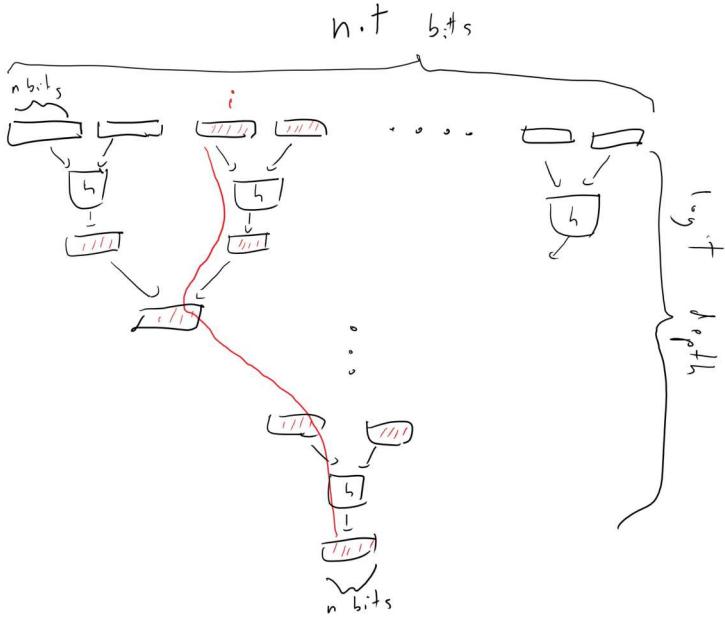


Figure 8.2: In the Merkle Tree construction we map a long string x into a block $y \in \{0, 1\}^n$ that is a “digest” of the long string x . As in a collision resistant hash we can imagine that this map is “one to one” in the sense that it won’t be possible to find $x' \neq x$ with the same digest. Moreover, we can efficiently certify that a certain bit of x is equal to some value without sending out all of x but rather the $\log t$ blocks that are on the path between i to the root together with their “siblings” used in the hash function, for a total of at most $2 \log t$ blocks.

Proof. The certificate π consists of a sequence of $\log t$ pairs of size- n blocks that are obtained by following the path on the tree from the i^{th} coordinate of x to the final root y . The last pair of blocks is the a preimage of y under h , while each pair on this list is a preimage of one of the blocks in the next pair. If $x_i \neq b$, then the first pair of blocks cannot be identical to the pair of blocks of x that contains the i^{th} coordinate. However, since we know the final root y is identical, if we compare the corresponding path in x to π , we will see that at some point there must be an input z in the path from x and a distinct input z' in π that hash to the same output. ■

8.3 Proofs of Retrievability

The above provides a way to ensure Alice that the value retrieved from a cloud storage is correct, but how can Alice be sure that the cloud server still stores the values that she *did not* ask about?

A priori, you might think that she obviously can’t. If Bob is lazy, or short on storage, he could decide to store only some small fraction of x that he thinks Alice is more likely to query for. As long as Bob

wasn't unlucky and Alice doesn't ask these queries, then it seems Bob could get away with this. In a *proof of retrievability*, first proposed by Juels and Kalisky in 2007, Alice would be able to get convinced that Bob does in fact store her data.

First, note that Alice can guarantee that Bob stores at least 99 percent of her data, by periodically asking him to provide answers (with proofs!) of the value of x at 100 or so random locations. The idea is that if Bob dropped more than 1 percent of the bits, then he'd be very likely to be caught "red handed" and get a question from Alice about a location he did not retain.

Now, if we used some redundancy to store x such as the RAID format, where it is composed of some small number c parts and we can recover any bit of the original data as long as at most one of the parts were lost, then we might hope that even if 1% of x was in fact lost by Bob, we could still recover the whole string. This is not a fool-proof guarantee since it could possibly be that the data lost by Bob was not confined to a single part. To handle this case one needs to consider generalizations of RAID known as "local reconstruction codes" or "locally decodable codes". The paper by [Dodis, Vadhan and Wichs](#) is a good source for this; see also [these slides by Seny Kamara](#) for a more recent overview of the theory and implementations.

8.4 Entropy extraction

As we've seen time and again, randomness is crucial to cryptography. But how do we get these random bits we need? If we only have a small number n of random bits (e.g., $n = 128$ or so) then we can expand them to as large a number as we want using a pseudorandom generator, but where do we get those initial n bits from?

The approach used in practice is known as "harvesting entropy". The idea is that we make great many measurements x_1, \dots, x_m of events that are considered "unpredictable" to some extent, including mouse movements, hard-disk and network latency, sources of noise etc... and accumulate them in an entropy "pool" which would simply be some memory array. When we estimate that we have accumulated more than 128 bits of randomness, then we hash this array into a 128 bit string which we'll use as a seed for a pseudorandom generator (see Fig. 8.3).² Because entropy needs to be measured *from the point of view of the attacker*, this "entropy estimation" routine is a bit of a "black art" and there isn't a very principled way to perform it.

² The reason that people use entropy "pools" rather than simply adding the entropy to the generator's state as it comes along is that the latter alternative might be insecure. Suppose that initial state of the generator was known to the adversary and now the entropy is "trickling in" one bit at a time while we continuously use the generator to produce outputs that can be observed by the adversary. Every time a new bit of entropy is added, the adversary now has uncertainty between two potential states of the generator, but once an output is produced this eliminates this uncertainty. In contrast, if we wait until we accumulate, say, 128 bits of entropy, then now the adversary will have 2^{128} possible state options to consider, and it could be computationally infeasible to cull them using further observation.

In practice people try to be very conservative (e.g., assume that there is only one bit of entropy for 64 bits of measurements or so) and hope for the best, which often works but sometimes also **spectacularly fails**, especially in embedded systems that do not have access to many of these sources.

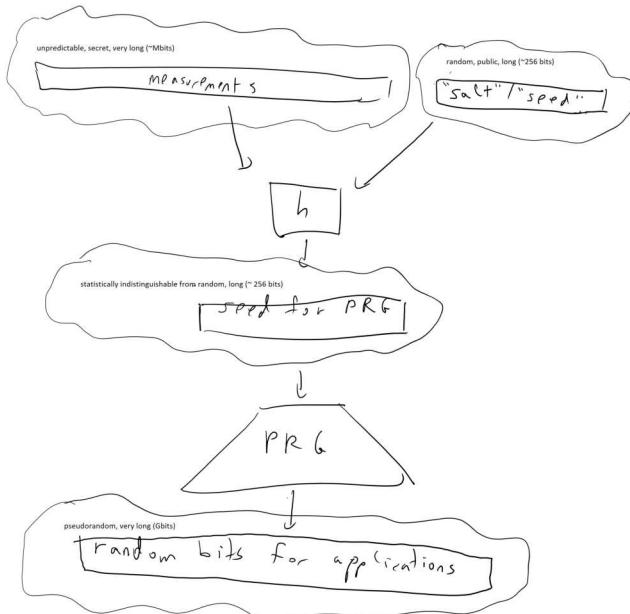


Figure 8.3: To obtain pseudorandom bits for cryptographic applications we hash down measurements which contain some *entropy* in them to a shorter string that is hopefully truly uniformly random or at least statistically close to it, and then expand this to get as many pseudorandom bits as we need using a pseudorandom generator.

How do hash functions figure into this? The idea is that if an input x has n bits of entropy then $h(x)$ would still have the same bits of entropy, as long as its output is larger than n . In practice people use the notion of “entropy” in a rather loose sense, but we will try to be more precise below.

The *entropy* of a distribution D is meant to capture the amount of “uncertainty” you have over the distribution. The canonical example is when D is the uniform distribution over $\{0,1\}^n$, in which case it has n bits of entropy. If you learn a single bit of D then you reduce the entropy by one bit. For example, if you learn that the 17th bit is equal to 0, then the new conditional distribution D' is the uniform distribution over all strings in $x \in \{0,1\}^n$ such that $x_{17} = 0$ and has $n - 1$ bits of entropy. Entropy is invariant under permutations of the sample space, and only depends on the vector of probabilities, and thus for every set S all notions of entropy will give $\log_2 |S|$ bits of entropy for the uniform distribution over S . A distribution that is uniform over some set S is known as a *flat* distribution.

Where different notions of entropy begin to differ is when the distributions are not flat. The *Shannon entropy* follows the principle that “original uncertainty = knowledge learned + new uncertainty”. That is, it obeys the *chain rule* which is that if a random variable (X, Y) has n bits of entropy, and X has k bits of entropy, then after learning X on average Y will have $n - k$ bits of entropy. That is,

$$H_{\text{Shannon}}(X) + H_{\text{Shannon}}(Y|X) = H_{\text{Shannon}}(X, Y)$$

Where the entropy of a conditional distribution $Y|X$ is simply $\mathbb{E}_{x \leftarrow_R X} H_{\text{Shannon}}(Y|X = x)$ where $Y|X = x$ is the distribution on Y obtained by conditioning on the event that $X = x$.

If (p_1, \dots, p_m) is a vector of probabilities summing up to 1 and let us assume they are rounded so that for every i , $p_i = k_i/2^n$ for some integer k_i . We can then split the set $\{0, 1\}^n$ into m disjoint sets S_1, \dots, S_m where $|S_i| = k_i$, and consider the probability distribution (X, Y) where Y is uniform over $\{0, 1\}^n$, and X is equal to i whenever $Y \in S_i$. Therefore, by the principles above we know that $H_{\text{Shannon}}(X, Y) = n$ (since X is completely determined Y and hence (X, Y) is uniform over a set of 2^n elements), and $H(Y|X) = \log k_i$. Thus the chain rule tells us that $H_{\text{Shannon}}(X) = n - \mathbb{E}[Y|X] = n - \sum_{i=1}^m p_i k_i = \sum_{i=1}^m p_i \log(p_i)$ since $p_i = k_i/2^n$ and hence $\log(p_i) = \log(k_i) - n$.

The Shannon entropy has many attractive properties, but it turns out that for cryptographic applications, the notion of *min entropy* is more appropriate. For a distribution X the *min-entropy* is simply defined as $H_\infty(X) = \min_x \log(1/\mathbb{P}[X = x])$.³ Note that if X is flat then $H_\infty(X) = H_{\text{Shannon}}(X)$ and that $H_\infty(X) \leq H_{\text{Shannon}}(X)$ for all X . We can now formally define the notion of an extractor:

Definition 8.2 — Randomness extractor. A function $h : \{0, 1\}^{\ell+n} \rightarrow \{0, 1\}^n$ is a *randomness extractor* (“extractor” for short) if for every distribution X over $\{0, 1\}^\ell$ with min entropy at least $2n$, if we pick s to be a random “salt”, the distribution $h(X)$ is computationally indistinguishable from the uniform distribution.⁴

The idea is that we apply the hash function to our measurements in $\{0, 1\}^\ell$ then if those measurements had at least k bits of entropy (with some extra “security margin”) then the output $h(X)$ will be as good as random. Since the “salt” value s is not secret, it can be chosen once at random and hardwired into the description of the function. (Indeed in practice people often do not explicitly use such a “salt”, but the hash function description contain some parameters IV that play a similar role.)

³ The notation $H_\infty(\cdot)$ for min entropy comes from the fact that one can define a *family* of entropy like functions, containing a function for every non-negative number p based on the p -norm of the probability distribution. That is, the Rényi entropy of order p is defined as $H_p(X) = (1-p)^{-1} \log(\sum_x \mathbb{P}[X = x]^p)$. The min entropy can be thought of as the limit of H_p when p tends to infinity while the Shannon entropy is the limit as p tends to 1. The entropy $H_2(\cdot)$ is related to the *collision probability* of X and is often used as well. The min entropy is the smallest among all the entropies and hence it is the most *conservative* (and so appropriate for usage in cryptography). For *flat sources*, which are uniform over a certain subset, all entropies coincide.

⁴ The pseudorandomness literature studies the notion of extractors much more generally and consider all possible variations for parameters such as the entropy requirement, the salt (more commonly known as seed) size, the distance from uniformity, and more. The type of notion we consider here is known in that literature as a “strong seeded extractor”. See [Vadhan’s monograph](#) for an in-depth treatment of this topic.

Theorem 8.3 — Random function is an extractor. Suppose that $h : \{0,1\}^{\ell+n} \rightarrow \{0,1\}^n$ is chosen at random, and $\ell < n^{100}$. Then with high probability h is an extractor.

Proof. Let h be chosen as above, and let X be some distribution over $\{0,1\}^\ell$ with $\max_x \{P[X = x]\} \leq 2^{-2n}$. Now, for every $s \in \{0,1\}^n$ let h_s be the function that maps $x \in \{0,1\}^\ell$ to $h(s||x)$, and let $Y_s = h_s(X)$. We want to prove that Y_s is pseudorandom. We will use the following claim:

Claim: Let $Col(Y_s)$ be the probability that two independent samples from Y_s are identical. Then with probability at least 0.99, $Col(Y_s) < 2^{-n} + 100 \cdot 2^{-2n}$.

Proof of claim: $E_s Col(Y_s) = \sum_s 2^{-n} \sum_{x,x'} P[X = x] P[X = x'] \sum_{y \in \{0,1\}^n} P[h(s,x) = y] P[h(s,x') = y]$. Let's separate this to the contribution when $x = x'$ and when they differ. The contribution from the first term is $\sum_s 2^{-n} \sum_x P[X = x]^2$ which is simply $Col(X) = \sum P[X = x]^2 \leq 2^{-2n}$ since $P[X = x] \leq 2^{-2n}$. In the second term, the events that $h(s,x) = y$ and $h(s,x') = y$ are independent, and hence the contribution here is at most $\sum_{x,x'} P[X = x] P[X = x'] 2^{-n}$. The claim follows from Markov.

Now suppose that T is some efficiently computable function from $\{0,1\}^n$ to $\{0,1\}$, then by Cauchy-Schwarz $|E[T(U_n)] - E[T(Y_s)]| = |\sum_{y \in \{0,1\}^n} T(y)[2^{-n} - P[Y_s = y]]| \leq \sqrt{\sum_y T(y)^2 \cdot \sum_y (2^{-n} - P[Y_s = y])^2}$ but opening up $\sum_y (2^{-n} - P[Y_s = y])^2$ we get $2^{-n} - 2 \cdot 2^{-n} \sum_y P[Y_s = y] + \sum_y P[Y_s = y]^2$ or $Col(Y_s) - 2^{-n}$ which is at most the negligible quantity $100 \cdot 2^{-2n}$. ■



Statistical randomness This proof actually proves a much stronger statement. First, note that we did not at all use the fact that T is efficiently computable and hence the distribution $h_s(X)$ will not be merely pseudorandom but actually *statistically indistinguishable* from truly random distribution. Second, we didn't use the fact that h is completely random but rather what we needed was merely *pairwise independence*: that for every $x \neq x'$ and y , $P_s[h_s(x) = h_s(x') = y] = 2^{-2n}$. There are efficient constructions of functions $h(\cdot)$ with this property, though in practice people still often use cryptographic hash function for this purpose.

8.4.1 *Forward and backward secrecy*

A cryptographic tool such as encryption is clearly insecure if the adversary learns the private key, and similarly the output of a pseudorandom generator is insecure if the adversary learns the seed. So, it might seem as if it's "game over" once this happens. However, there is still some hope. For example, if the adversary learns it at time t but didn't know it before then, then one could hope that she does not learn the information that was exchanged up to time $t - 1$. This property is known as "forward secrecy". It had recently gained interest as means to protect against powerful "attackers" such as the NSA that may record the communication transcripts in the hope of deciphering them in some future after it had learned the secret key. In the context of pseudorandom generators, one could hope for both forward and backward secrecy. Forward secrecy means that the state of the generator is updated at every point in time in a way that learning the state at time t does not help in recovering past state, and "backward secrecy" means that we can recover from the adversary knowing our internal state by updating the generator with fresh entropy. See [this paper of me and Halevi](#) for some discussions of this issue, as well as [this later work by Dodis et al.](#)

9

Public key cryptography

People have been dreaming about heavier than air flight since at least the days of Leonardo Da Vinci (not to mention Icarus from the greek mythology). Jules Verne wrote with rather insightful details about going to the moon in 1865. But, as far as I know, in all the thousands of years people have been using secret writing, until about 50 years ago no one has considered the possibility of communicating securely without first exchanging a shared secret key. However, in the late 1960's and early 1970's, several people started to question this "common wisdom".

Perhaps the most surprising of these visionaries was an undergraduate student at Berkeley named Ralph Merkle. In the fall of 1974 he wrote a [project proposal](#) for his computer security course that while "it might seem intuitively obvious that if two people have never had the opportunity to prearrange an encryption method, then they will be unable to communicate securely over an insecure channel... I believe it is false". The project proposal was rejected by his professor as "not good enough". Merkle later submitted a paper to the communication of the ACM where he apologized for the lack of references since he was unable to find any mention of the problem in the scientific literature, and the only source where he saw the problem even *raised* was in a science fiction story. The paper was rejected with the comment that "Experience shows that it is extremely dangerous to transmit key information in the clear." Merkle showed that one can design a protocol where Alice and Bob can use T invocations of a hash function to exchange a key, but an adversary (in the random oracle model, though he of course didn't use this name) would need roughly T^2 invocations to break it. He conjectured that it may be possible to obtain such protocols where breaking is *exponentially harder* than using them, but could not think of any concrete way to doing so.

We only found out much later that in the late 1960's, a few years before Merkle, James Ellis of the British Intelligence agency GCHQ was [having similar thoughts](#). His curiosity was spurred by an old World-War II manuscript from Bell labs that suggested the following way that two people could communicate securely over a phone line. Alice would inject noise to the line, Bob would relay his messages, and then Alice would subtract the noise to get the signal. The idea is that an adversary over the line sees only the sum of Alice's and Bob's signals, and doesn't know what came from what. This got James Ellis thinking whether it would be possible to achieve something like that digitally. As he later recollects, in 1970 he realized that in principle this should be possible, since he could think of an hypothetical black box B that on input a "handle" α and plaintext p would give a "ciphertext" c and that there would be a secret key β corresponding to α , such that feeding β and c to the box would recover p . However, Ellis had no idea how to actually instantiate this box. He and others kept giving this question as a puzzle to bright new recruits until one of them, Clifford Cocks, came up in 1973 with a candidate solution loosely based on the factoring problem; in 1974 another GCHQ recruit, Malcolm Williamson, came up with a solution using modular exponentiation.

But among all those thinking of public key cryptography, probably the people who saw the furthest were two researchers at Stanford, Whit Diffie and Martin Hellman. They realized that with the advent of electronic communication, cryptography would find new applications beyond the military domain of spies and submarines. And they understood that in this new world of many users and point to point communication, cryptography would need to scale up. They envisioned an object which we now call "trapdoor permutation" though they called it "one way trapdoor function" or sometimes simply "public key encryption". This is a collection of permutations $\{p_k\}$ where p_k is a permutation over (say) $\{0, 1\}^{|k|}$, and the map $(x, k) \mapsto p_k(x)$ is efficiently computable *but* the reverse map $(k, y) \mapsto p_k^{-1}(y)$ is computationally hard. Yet, there is also some secret key $s(k)$ (i.e., the "trapdoor") such that using $s(k)$ it is possible to efficiently compute p_k^{-1} . Their idea was that using such a trapdoor permutation, Alice who knows $s(k)$ would be able to publish k on some public file such that everyone who wants to send her a message x could do so by computing $p_k(x)$. (While today we know, due to the work of Goldwasser and Micali, that such a deterministic encryption is not a good idea, at the time Diffie and Hellman had amazing intuitions but didn't really have proper definitions of security.) But they didn't stop there. They realized that protecting the *integrity* of

communication is no less important than protecting its *secrecy*. Thus they imagined that Alice could “run encryption in reverse” in order to certify or *sign* messages. That is, given some message m , Alice would send the value $x = p_k^{-1}(h(m))$ (for a hash function h) as a way to certify that she endorses m , and every person who knows k could verify this by checking that $p_k(x) = h(m)$.

However, Diffie and Hellman were in a position not unlike physicists who predicted that a certain particle should exist but without any experimental verification. Luckily they met **Ralph Merkle**, and his ideas about a probabilistic *key exchange protocol*, together with a suggestion from their Stanford colleague **John Gill**, inspired them to come up with what today is known as the *Diffie-Hellman Key Exchange* (which unbeknownst to them was found two years earlier at GCHQ by Malcolm Williamson). They published their paper “[New Directions in Cryptography](#)” in 1976, and it is considered to have brought about the birth of modern cryptography. However, they still didn’t find their elusive trapdoor function. This was done the next year by Rivest, Shamir and Adleman who came up with the RSA trapdoor function, which through the framework of Diffie and Hellman yielded not just encryption but also signatures (this was essentially the same function discovered earlier by Clifford Cocks at GCHQ, though as far as I can tell Cocks, Ellis and Williamson did not realize the application to digital signatures). From this point on began a flurry of advances in cryptography which hasn’t really died down till this day.

9.1 Private key crypto recap

Before we embark on the wonderful journey to *public key cryptography*, let’s briefly look back and see what we learned about *private key cryptography*. This material is mostly covered in Chapters 1 to 9 of the Katz Lindell (KL) book and Part I (Chapters 1-9) of the Boneh Shoup (BS) book. Now would be a good time for you to read the corresponding proofs in one or both of these books. It is often helpful to see the same proof presented in a slightly different way. Below is a review of some of the various reductions we saw in class, with pointers to the corresponding sections in these books.

- Pseudorandom generators (PRG) length extension (from $n + 1$ output PRG to $\text{poly}(n)$ output PRG): KL 7.4.2, BS 3.4.2
- PRG’s to pseudorandom functions (PRF’s): KL 7.5, BS 4.6
- PRF’s to Chosen Plaintext Attack (CPA) secure encryption: KL

3.5.2, BS 5.5

- PRF's to secure Message Authentication Codes (MAC's): KL 4.3, BS 6.3
- MAC's + CPA secure encryption to chosen ciphertext attack (CCA) secure encryption: BS 4.5.4, BS 9.4
- Pseudorandom permutation (PRP's) to CPA secure encryption / block cipher modes: KL 3.5.2, KL 3.6.2, BS 4.1, 4.4, 5.4
- Hash function applications: fingerprinting, Merkle trees, passwords: KL 5.6, BS Chapter 8
- Coin tossing over the phone: we saw a construction in class that used a *commitment scheme* built out of a pseudorandom generator. This is shown in BS 3.12, KL 5.6.5 shows an alternative construction using random oracles.
- PRP's from PRF's: we only sketched the construction which can be found in KL 7.6 or BS 4.5

One major point we did *not* talk about in this course was *one way functions*. The definition of a one way function is quite simple:

Definition 9.1 — One Way Functions. A function $f : \{0,1\}^* \rightarrow \{0,1\}^*$ is a *one way function* if it is efficiently computable and for every n and a $\text{poly}(n)$ time adversary A , the probability over $x \leftarrow_R \{0,1\}^n$ that $A(f(x))$ outputs x' such that $f(x') = f(x)$ is negligible.

The “OWF conjecture” is the conjecture that one way functions exist. It turns out to be a necessary and sufficient condition for much of private key cryptography. That is, the following theorem is known (by combining works of many people):

Theorem 9.2 — One way functions and private key cryptography. The following are equivalent:

- * One way functions exist
- * Pseudorandom generators (with non-trivial stretch) exist
- * Pseudorandom functions exist
- * CPA secure private key encryptions exist
- * CCA secure private key encryptions exist
- * Message Authentication Codes exist
- * Commitment schemes exist

The key result in the proof of this theorem is the result of Hastad, Impagliazzo, Levin and Luby that if one way functions exist then pseudorandom generators exist. If you are interested in finding out more, Sections 7.2-7.4 in the KL book cover a special case of this theorem for the case that the one way function is a *permutation* on $\{0, 1\}^n$ for every n . This proof has been considerably simplified and quantitatively improved in works of Haitner, Holenstein, Reingold, Vadhan, Wee and Zheng. See [this talk of Salil Vadhan](#) for more on this. See also [these lecture notes](#) from a Princeton seminar I gave on this topic (though the proof has been simplified since then by the above works).



Attacks on private key cryptosystems Another topic we did not discuss in depth is attacks on private key cryptosystems. These attacks often work by “opening the black box” and looking at the internal operation of block ciphers or hash functions. One then often assigns variables to various internal registers, and then we look to finding collections of inputs that would satisfy some non-trivial relation between those variables. This is a rather vague description, but you can read KL Section 6.2.6 on *linear* and *differential* cryptanalysis and BS Sections 3.7-3.9 and 4.3 for more information. See also [this course of Adi Shamir](#). There is also the fascinating area of *side channel* attacks on both public and private key crypto.



Digital Signatures We will discuss in this lecture *Digital signatures*, which are the public key analog of message authentication codes. Surprisingly, despite being a “public key” object, it is possible to base digital signatures on one-way functions (this is obtained using ideas of Lamport, Merkle, Goldwasser-Goldreich-Micali, Naor-Yung, and Rompel). However these constructions are not very efficient (and this may be inherent) and so in practice people use digital signatures that are built using similar techniques to those used for public key encryption.

9.2 Public Key Encryptions: Definition

We now discuss how we define security for public key encryption. As mentioned above, it took quite a while for cryptographers to arrive at

the “right” definition, but in the interest of time we will skip ahead to what by now is the standard basic notion (see also Fig. 9.1):

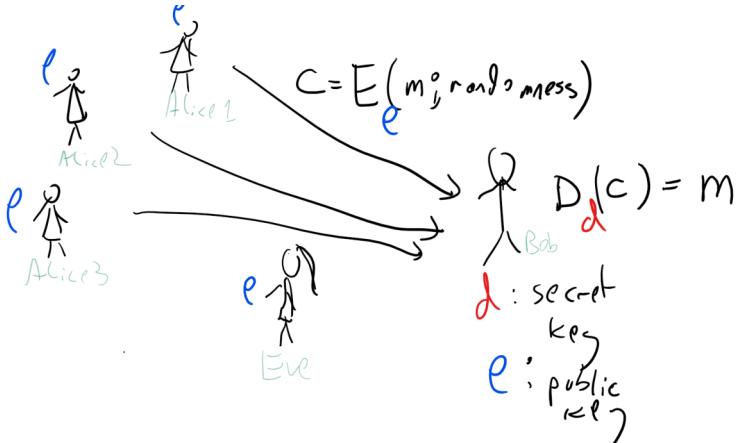


Figure 9.1: In a public key encryption, the receiver Bob generates a pair of keys (e, d) . The *encryption key* e is used for encryption, and the *decryption key* is used for decryption. We call it a public key system since the security of the scheme does not rely on the adversary Eve not knowing the encryption key. Hence Bob can publicize the key e to a great many potential receivers, and still ensure confidentiality of the messages he receives.

Definition 9.3 — Public key encryption. A triple of efficient algorithms (G, E, D) is a *public key encryption scheme* if it satisfies the following:

* G is a probabilistic algorithm known as the *key generation algorithm* that on input 1^n outputs a distribution over pair of keys (e, d) .

* E is the *encryption algorithm* that takes a pair of inputs e, m with $m \in \{0, 1\}^n$ and outputs $c = E_e(m)$

* D is the *decryption algorithm* that takes a pair of inputs d, c and outputs $m' = D_d(c)$.

* For every $m \in \{0, 1\}^n$, with probability $1 - negl(n)$ over the choice of (e, d) output from $G(1^n)$ and the coins of E, D , $D_d(E_e(m)) = m$.

We say that (G, E, D) is *CPA secure* every efficient adversary A wins the following game with probability at most $1/2 + negl(n)$:

- $(e, d) \leftarrow_R G(1^n)$
- A is given e and outputs a pair of messages $m_0, m_1 \in \{0, 1\}^n$.
- A is given $c = E_e(m_b)$ for $b \leftarrow_R \{0, 1\}$.
- A outputs $b' \in \{0, 1\}$ and *wins* if $b' = b$.

P Despite it being a “chosen plaintext attack”, we don’t explicitly give A access to the encryption oracle in the public key setting. Make sure you understand why giving it such access would not give it more power.

One metaphor for a public key encryption is a “self-locking lock” where you don’t need the key to *lock it* (but rather you simply push the shackle until it clicks and lock) but you do need the key to *unlock* it. So, if Alice generates $(e, d) = G(1^n)$ then e serves as the “lock” that can be used to *encrypt* messages for Alice while only d can be used to *decrypt* the messages. Another way to think about it is that e is a “hobbled key” that can be used for only some of the functions of d .

9.2.1 The obfuscation paradigm

Why would someone imagine that such a magical object could exist? The writing of both James Ellis as well as Diffie and Hellman suggests that their thought process was roughly as follows. You imagine a “magic black box” B such that if all parties have access to B then we could get a public key encryption scheme. Now if public key encryption was impossible it would mean that for every possible program P that computes the functionality of B , if we distribute the code of P to all parties then we don’t get a secure encryption scheme. That means that *no matter what program P the adversary gets*, she will always be able to get some information out of that code that helps break the encryption, even though she wouldn’t have been able to break it if P was a black box. Now intuitively understanding arbitrary code is a very hard problem, so Diffie and Hellman imagined that it might be possible to take this ideal B and compile it to some sufficiently low level assembly language so that it would behave as a “virtual black box”. In particular, if you took, say, the encoding procedure $m \mapsto p_k(m)$ of a block cipher with a particular key k , and ran it through an optimizing compiler you might hope that while it would be possible to perform this map using the resulting executable, it will be hard to extract k from it, and hence could treat this code as a “public key”. This suggests the following approach for getting an encryption scheme:

“Obfuscation based public key encryption”:

Ingredients: (i) A pseudorandom permutation collection $\{p_k\}_{k \in \{0,1\}^*}$ where for every $k \in \{0,1\}^n$, $p_k : \{0,1\}^n \rightarrow \{0,1\}^n$, (ii) An “ob-

fuscating compiler” polynomial-time computable $O : \{0,1\}^* \rightarrow \{0,1\}^*$ such that for every circuit C , $O(C)$ is a circuit that computes the same function as C

- *Key Generation:* The private key is $k \leftarrow_R \{0,1\}^n$, the public key is $E = O(C_k)$ where C_k is the circuit that maps $x \in \{0,1\}^n$ to $p_k(x)$.
- *Encryption:* To encrypt $m \in \{0,1\}^n$ with public key E , choose $IV \leftarrow_R \{0,1\}^n$ and output $(IV, E(x \oplus IV))$.
- *Decryption:* To decrypt (IV, y) with key k , output $IV \oplus p_k^{-1}(y)$.

Diffie and Hellman couldn’t really find a way to make this work, but it convinced them this notion of public key is not *inherently impossible*. This concept of compiling a program into a functionally equivalent but “inscrutable” form is known as *software obfuscation*. It had turned out to be quite a tricky object to both define formally and achieve, but it serves as a very good intuition as to what can be achieved, even if, as the random oracle, this intuition can sometimes be too optimistic. (Indeed, if software obfuscation was possible then we could obtain a “random oracle like” hash function by taking the code of a function f_k chosen from a PRF family and compiling it through an obfuscating compiler.)

We will not formally define obfuscators yet, but on intuitive level it would be a compiler that takes a program P and maps into a program P' such that:

- P' is not much slower/bigger than P (e.g., as a Boolean circuit it would be at most polynomially larger)
- P' is functionally equivalent to P , i.e., $P'(x) = P(x)$ for every input x .¹
- P' is “inscrutable” in the sense that seeing the code of P' is not more informative than getting *black box access* to P .

Let me stress again that there is no known construction of obfuscators achieving something similar to this definition. In fact, the most natural formalization of this definition is *impossible* to achieve (as we might see later in this course). Only very recently (exciting!) progress was finally made towards obfuscators-like notions strong enough to achieve these and other applications, and there are some significant caveats (see my survey on this topic).

¹ For simplicity, assume that the program P is *side effect free* and hence it simply computes some function, say from $\{0,1\}^n$ to $\{0,1\}^\ell$ for some n, ℓ .

However, when trying to stretch your imagination to consider the amazing possibilities that could be achieved in cryptography, it is not a bad heuristic to first ask yourself what could be possible if only everyone involved had access to a magic black box. It certainly worked well for Diffie and Hellman.

9.3 Some concrete candidates:

We would have loved to prove a theorem of the form:

"Theorem": If the PRG conjecture is true then there exists a CPA-secure public key encryption.

This would have meant that we do not need to assume anything more than the already minimal notion of pseudorandom generators (or equivalently, one way functions) to obtain public key cryptography. Unfortunately, no such result is known (and this may be inherent). The kind of results we know have the following form:

Theorem: If problem X is hard then there exists a CPA-secure public key encryption.

Where X is some problem that people have tried to solve and couldn't. Thus we have various *candidates* for public key encryption and we fervently hope that at least one of them is actually secure. The **dirty little secret** of cryptography is that we actually don't have that many candidates. We really have only two well studied families.² One is the "group theoretic" family that relies on the difficulty of the discrete logarithm (over modular arithmetic or elliptic curves) or the integer factoring problem. The other is the "coding/lattice theoretic" family that relies on the difficulty of solving noisy linear equations or related problems such as finding short vectors in a *lattice* and solving instances of the "knapsack" problem. Moreover, problems from the first family are known to be *efficiently solvable* in a computational model known as "quantum computing". If large scale physical devices that simulate this model, known as *quantum computers*, then they could break all cryptosystems relying on these problems and we'll be down to only having a *single* family of candidate public key encryption schemes.

We will start by describing cryptosystems based on the first family (which was discovered before the other, as well as much more widely implemented), and in future lectures talk about the second family.

² There have been some other more exotic suggestions for public key encryption (including some by *yours truly* as well as suggestions such as the *isogeny star problem*, though see also *this*), but they have not yet received wide scrutiny.

9.3.1 Diffie-Hellman Encryption (aka El-Gamal)

The Diffie-Hellman public key system is built on the presumed difficulty of the *discrete logarithm problem*:

For any number p , let \mathbb{Z}_p be the set of numbers $\{0, \dots, p - 1\}$ where addition and multiplication are done modulo p . We will think of numbers p that are of magnitude roughly 2^n , so they can be described with about n bits. We can clearly multiply and add such numbers modulo p in $\text{poly}(n)$ time. If $g \in \mathbb{Z}_p$ and a is any natural number, we can define g^a to be simply $g \cdot g \cdots g$ (a times). A priori one might think that it would take $a \cdot \text{poly}(n)$ time to compute g^a , which might be exponential if a itself is roughly 2^n . However, we can compute this in $\text{poly}((\log a) \cdot n)$ time using the *repeated squaring trick*. The idea is that if $a = 2^\ell$ then we can compute g^a in ℓ by squaring g ℓ times, and a general a can be decomposed into powers of two using the binary representation.

The *discrete logarithm* problem is the problem of computing, given $g, h \in \mathbb{Z}_p$, a number a such that $g^a = h$. If such a solution a exists then there is always also a solution of size at most p (can you see why?) and so the solution can be represented using n bits. However, currently the best known algorithm for computing the discrete logarithm run in time roughly $2^{n^{1/3}}$ ³ which currently becomes prohibitively expensive when p is a prime of length about 2048 bits.³

John Gill suggested to Diffie and Hellman that modular exponentiation can be a good source for the kind of “easy-to-compute but hard-to-invert” functions they were looking for. Diffie and Hellman based a public key encryption scheme as follows:

- The *key generation algorithm*, on input n , samples a prime number p of n bits description (i.e., between 2^{n-1} to 2^n), a number $g \leftarrow_R \mathbb{Z}_p$ and $a \leftarrow_R \{0, \dots, p - 1\}$. We also sample a hash function $H : \{0, 1\}^n \rightarrow \{0, 1\}^\ell$. The public key e is (p, g, g^a, H) while the secret key d is a .⁴
- The *encryption algorithm*, on input a message $m \in \{0, 1\}^\ell$ and a public key $e = (p, g, h, H)$ will choose a random $b \leftarrow_R \{0, \dots, p - 1\}$, and output $(g^b, H(h^b) \oplus m)$.
- The *decryption algorithm*, on input a ciphertext (f, y) and the secret key, will output $H(f^a) \oplus y$.

The correctness of the decryption algorithm follows from the fact that $(g^a)^b = (g^b)^a = g^{ab}$ and hence $H(h^b)$ computed by the encryption algorithm is the same as the value $H(f^a)$ computed by

³ The running time of the best known algorithms for computing the discrete logarithm modulo n bit primes is $2^{f(n)2^{n/3}}$ where $f(n)$ is a function that depends polylogarithmically on n . If $f(n)$ would equal 1 then we'd need numbers of $128^3 \approx 2 \cdot 10^6$ bits to get 128 bits of security, but because $f(n)$ is larger than one, the current estimates are that we need to let $n = 3072$ bit key to get 128 bits of security. Still the existence of such a non-trivial algorithm means that we need much larger keys than those used for private key systems to get the same level of security. In particular, to double the estimated security to 256 bits, NIST recommends that we multiply the RSA keysize give-fold to 15,360. (The same document also says that SHA-256 gives 256 bits of security as a pseudorandom generator but only 128 bits when used to hash documents for digital signatures; can you see why?)

⁴ Formally the secret key should contain all the information in the public key plus the extra secret information, but we omit the public information for simplicity of notation.

the decryption algorithm. A simple relation between the discrete logarithm and the Diffie-Hellman system is the following:

Lemma 9.4 If there is a polynomial time algorithm for the discrete logarithm problem then the Diffie-Hellman system is *insecure*.

Proof. Using a discrete logarithm algorithm, we can compute the private key a from the parameters p, g, g^a present in the public key, and clearly once we know the private key we can decrypt any message of our choice. ■

Unfortunately, no such result is known in the other direction. However in the random oracle model, we can prove that this protocol is secure assuming the task of computing g^{ab} from g^a and g^b (which is now known as the *Diffie-Hellman problem*) is hard.⁵

Computational Diffie-Hellman Assumption: Let G be a group elements of which can be described in n bits, with an associative and commutative multiplication operation that can be computed in $\text{poly}(n)$ time. The *Computational Diffie-Hellman (CDH)* assumption holds with respect to the group G if for every generator (see below) g of G and efficient algorithm A , the probability that on input g, g^a, g^b, A outputs the element g^{ab} is negligible as a function of n .⁶

⁵ One can get security results for this protocol without a random oracle if we assume a stronger variant known as the *Decisional Diffie-Hellman (DDH)* assumption.

In particular we can make the following conjecture:

Computational Diffie-Hellman Conjecture for mod prime groups: For a random n -bit prime and random $g \in \mathbb{Z}_p$, the CDH holds with respect to the group $G = \{g^a \bmod p \mid a \in \mathbb{Z}\}$.

That is, for every polynomial $q : \mathbb{N} \rightarrow \mathbb{N}$, if n is large enough, then with probability at least

$1 - 1/q(n)$ over the choice of a uniform prime $p \in [2^n]$ and $g \in \mathbb{Z}_p$, for every circuit A of size at most $q(n)$, the probability that $A(g, p, g^a, g^b)$ outputs h such that $g^{ab} = h \bmod p$ is at most $1/q(n)$ where the probability is taken over a, b chosen at random in \mathbb{Z}_p .⁷

⁶ Formally, since it is an asymptotic statement, the CDH assumption needs to be defined with a *sequence of groups*. However, to make notation simpler we will ignore this issue, and use it only for groups (such as the numbers modulo some n bit primes) where we can easily increase the “security parameter” n .



Please take your time to re-read the following conjecture until you are sure you understand what it means. Victor Shoup's excellent and online available book [A Computational Introduction to Number](#)

⁷ In practice people often take g to be a generator of a group significantly smaller in size than p , which enables a, b to be smaller numbers and hence multiplication to be more efficient. We ignore this optimization in our discussions.

Theory and Algebra has an in depth treatment of groups, generators, and the discrete log and Diffie-Hellman problem. See also Chapters 10.4 and 10.5 in the Boneh Shoup book, and Chapters 8.3 and 11.4 in the Katz-Lindell book.

Theorem 9.5 — Diffie-Hellman security in Random Oracle Model. Suppose that the Computational Diffie-Hellman Conjecture for mod prime groups is true. Then, the Diffie-Hellman public key encryption is CPA secure in the random oracle model.

Proof. For CPA security we need to prove that (for fixed \mathbb{G} of size p and random oracle H) the following two distributions are computationally indistinguishable for every two strings $m, m' \in \{0, 1\}^\ell$:

- $(g^a, g^b, H(g^{ab}) \oplus m)$ for a, b chosen uniformly and independently in \mathbb{Z}_p
- $(g^a, g^b, H(g^{ab}) \oplus m')$ chosen uniformly and independently in \mathbb{Z}_p .

(can you see why this implies CPA security? you should pause here and verify this!)

We make the following claim:

CLAIM: For a fixed \mathbb{G} of size p , generator g for \mathbb{G} , and given random oracle H , if there is a size T distinguisher A with ϵ advantage between the distribution $(g^a, g^b, H(g^{ab}))$ and the distribution (g^a, g^b, U_ℓ) (where a, b are chosen uniformly and independently in \mathbb{Z}_p) then there is a size $\text{poly}(T)$ algorithm A' to solve the Diffie-Hellman problem with respect to \mathbb{G}, g with success at least ϵ . That is, for random $a, b \in \mathbb{Z}_p$, $A'(g, g^a, g^b) = g^{ab}$ with probability at least $\epsilon/(2T)$.

Proof of claim: The proof is simple. We claim that under the assumptions above, a makes the query g^{ab} to its oracle H with probability at least $\epsilon/2$ since otherwise, by the “lazy evaluation” paradigm, we can assume that $H(g^{ab})$ is chosen independently at random after A 's attack is completed and hence (conditioned on the adversary not making that query), the value $H(g^{ab})$ is indistinguishable from a uniform output. Therefore, on input g, g^a, g^b, A' can simulate A and simply output one of the at most T queries that A makes to H at random, and will be successful with probability at least $\epsilon/(2T)$.

Now given the claim, we can complete the proof of security via the following hybrids. Define the following “hybrid” distributions (where in all cases a, b are chosen uniformly and independently in \mathbb{Z}_p):

- $H_0: (g^a, g^b, H(g^{ab}) \oplus m)$
- $H_1: (g^a, g^b, U_\ell \oplus m)$
- $H_2: (g^a, g^b, U_\ell \oplus m')$
- $H_3: (g^a, g^b, H(g^{ab}) \oplus m)$

The claim implies that $H_0 \approx H_1$. Indeed otherwise we could transform a distinguisher T between H_0 and H_1 to a distinguisher T' violating the claim by letting $T'(h, h', z) = T(h, h', z \oplus m)$.

The distributions H_1 and H_2 are *identical* by the same argument as the security of the one time pad (since $U_\ell \oplus m$ is identical to U_ℓ).

The distributions H_2 and H_3 are computationally indistinguishable by the same argument that $H_0 \approx H_1$.

Together these imply that $H_0 \approx H_3$ which yields the CPA security of the scheme. ■

R **Elliptic curve cryptography** As mentioned, the Diffie-Hellman systems can be run with many variants of Abelian groups. Of course, for some of those groups the discrete logarithm problem might be easy, and so they would be inappropriate to use for this system. One variant that has been proposed is **elliptic curve cryptography**. This is a group consisting of points of the form $(x, y, z) \in \mathbb{Z}_p^3$ that satisfy a certain equation, and multiplication can be defined according in a certain way. The main advantage of elliptic curve cryptography is that the best known algorithms run in time $2^{\approx n}$ as opposed to $2^{\approx n^{1/3}}$ which allows for much shorter keys. Unfortunately, elliptic curve cryptography is just as susceptible to quantum algorithms as the discrete logarithm problem over \mathbb{Z}_p .

R **Encryption vs Key Exchange and El Gamal** In most of the cryptography literature the protocol above is called the *Diffie-Hellman Key Exchange* protocol, and when considered as a public key system it is sometimes known as *ElGamal encryption*.⁸ The reason for this mostly stems from the early confusion on what are the right security definitions. Diffie and Hellman thought of encryption as a *deterministic* process and so they called their scheme a “key exchange protocol”. The work of Goldwasser and Micali showed that encryption must be probabilistic for security. Also, because of efficiency considera-

tions, these days public key encryption is mostly used as a mechanism to exchange a key for a private key encryption, that is then used for the bulk of the communication. Together this means that there is not much point in distinguishing between a two message key exchange algorithm and a public key encryption.

9.3.2 Sampling random primes

To sample a random n bit prime one can sample a random number $0 \leq p < 2^n$ and then test if p is prime. If it is not prime, then we can sample a new random number again. To make this work we need to show two properties:

Efficient testing: That there is a $\text{poly}(n)$ time algorithm to test whether an n bit number is a prime. It turns out that there are such **known algorithms**. *Randomized* algorithm have been known since the 1970's. Moreover in a 2002 breakthrough, **Manindra Agrawal, Neeraj Kayal, and Nitin Saxena** (a professor and two undergraduate students from the Indian Institute of Technology Kanpur) came up with the first deterministic polynomial time algorithm for testing primality.

Prime density: That the probability that a random n bit number is prime is at least $1/\text{poly}(n)$. This probability is in fact $1/\ln(2^n) = \Omega(1/n)$ by the **Prime Number Theorem**. However, for the sake of completeness, we sketch below a simple argument showing the probability is at least $\Omega(1/n^2)$.

Lemma 9.6 The number of primes between 1 and N is $\Omega(N/\log N)$.

Proof. Recall that the *least common multiple* (*LCM*) of two or more a_1, \dots, a_t is the smallest number that is a multiple of all of the a_i 's. One way to compute the LCM of a_1, \dots, a_t is to take the prime factorizations of all the a_i 's, and then the LCM is the product that all the primes that appear in these factorizations, with the highest power that they appear in. Let k be the number of primes between 1 and N . The lemma will follow from the following two claims:

CLAIM 1: $\text{LCM}(1, \dots, N) \leq N^k$.

CLAIM 2: If N is odd then $\text{LCM}(1, \dots, N) \geq 2^{N-1}$.

The two claim immediately imply the result since they imply that $2^N \leq N^k$, and taking logs we get that $N - 2 \leq k \log N$ or $k \geq (N - 2)/\log N$. (We can assume that N is odd without loss of

⁸ ElGamal's actual contribution was to design a *signature scheme* based on the Diffie-Hellman problem, a variant of which is the Digital Signature Algorithm (DSA) described below.

generality, since changing from N to $N + 1$ can change the number of primes by at most one.) Thus all that is left is to prove the two claims.

Proof of CLAIM 1: Let p_1, \dots, p_k be all the prime numbers between 1 and N , and let e_i be the largest integer such that $p_i^{e_i} \leq N$ and $L = p_1^{e_1} \cdots p_k^{e_k}$. Since L is the product of k terms, each of size at most N , $L \leq N^k$. But we claim that every number $1 \leq a \leq N$ divides L . Indeed, every prime p in the prime factorization of a is one of the p_i 's, and since $a \leq N$, the power in which p appears in a is at most e_i . By the definition of the least common multiple, this means that $\text{LCM}(1, \dots, N) \leq L$. QED (CLAIM 1)

Proof of CLAIM 2: Consider the integral $I = \int_0^1 x^{(N-1)/2} (1-x)^{(N-1)/2} dx$. This is clearly some positive number and so $I > 0$. On one hand, for every x between zero and one, $x(1-x) \leq 1/4$ and hence I is at most $4^{-(N-1)/2} = 2^{-N+1}$. On the other hand, the polynomial $x^{N/2}(1-x)^{N/2}$ is some polynomial of degree at most $N-1$ with integer coefficients, and so $I = \sum_{k=1}^{N-1} C_k \int_0^1 x^k dx$ for some integer coefficients C_1, \dots, C_{N-1} . Since $\int_0^1 x^k = \frac{1}{k+1}$, we see that I is a sum of fractions with integer numerators and with denominators that are at most N . Since all the denominators are at most N and $I > 0$, it follows that $I \geq \frac{1}{\text{LCM}(1, \dots, N)}$, and so

$$2^{-N+1} \geq I \geq \frac{1}{\text{LCM}(1, \dots, N)} \quad (9.1)$$

which implies $\text{LCM}(1, \dots, N) \leq 2^{N-1}$. QED (CLAIM 2 and hence lemma) ■

9.3.3 A little bit of group theory.

If you haven't seen group theory, it might be useful for you to do a quick review. We will not use much group theory and mostly use the theory of finite commutative (also known as Abelian) groups (in fact often *cyclic*) which are such a baby version that it might not be considered true "group theory" by many group theorists. Shoup's [excellent book](#) contains everything we need to know (and much more than that). What you need to remember is the following:

- A *finite commutative group* G is a finite set together with a multiplication operation that satisfies $a \cdot b = b \cdot a$ and $(a \cdot b) \cdot c = (a \cdot b) \cdot c$.
- G has a special element known as 1, where $g1 = 1g = g$ for every $g \in G$ and for every $g \in G$ there exists an element $g^{-1} \in G$ such that $gg^{-1} = 1$.
- For every $g \in G$, the *order* of g , denoted $\text{order}(g)$, is the smallest positive integer a such that $g^a = 1$.

The following basic facts are all not too hard to prove and would be useful exercises:

- For every $g \in G$, the map $a \mapsto g^a$ is a k to 1 map from $\{0, \dots, |G| - 1\}$ to G where $k = |G|/\text{order}(g)$. See footnote for hint⁹
- As a corollary, the order of g is always a divisor of $|G|$. This is a special case of a more general phenomenon: the set $\{g^a \mid a \in \mathbb{Z}\}$ is a subset of the group G that is closed under multiplication, and such subsets are known as *subgroups* of G . It is not hard to show (using the same approach as above) that for every group G and subgroup H , the size of H divides the size of G . This is known as **Lagrange's Theorem** in group theory.
- An element g of G is called a *generator* if $\text{order}(g) = |G|$. A group is called *cyclic* if it has a generator. If G is cyclic then there is a (not necessarily efficiently computable) *isomorphism* $\phi : G \rightarrow \mathbb{Z}_{|G|}$ which is a one-to-one and onto map satisfying $\phi(g \cdot h) = \phi(g) + \phi(h)$ for every $g, h \in G$.

When using a group G for the Diffie-Hellman protocol, we want the property that g is a *generator* of the group, which also means that the map $a \mapsto g^a$ is a one to one mapping from $\{0, \dots, |G| - 1\}$ to G . This can be efficiently tested if we know the order of the group and its factorization, since it will occur if and only if $g^a \neq 1$ for every $a < |G|$ (can you see why this holds?) and we know that if $g^a = 1$ then a must divide $|G|$ (and this?).

It is not hard to show that a random element $g \in G$ will be a generator with non-trivial probability (for similar reasons that a random number is prime with non-trivial probability) and hence an approach to getting such a generator is to simply choose g at random and test that $g^a \neq 1$ for all of the fewer than $\log |G|$ numbers that are obtained by taking $|G|/q$ where q is a factor of $|G|$.



Try to stop here and verify all the facts on groups mentioned above.

9.3.4 Digital Signatures

Public key encryption solves the *confidentiality* problem but we still need to solve the *authenticity* or *integrity* problem, which might be even more important in practice. That is, suppose Alice wants to endorse a message m that *everyone* can verify but only she can sign.

⁹ For every $f \in G$, you can show a one to one and onto mapping between the set $\{a : g^a = 1\}$ and the set $\{b : g^b = f\}$ by choosing some element b from the latter set and looking at the map $a \mapsto a + b \bmod |G|$.

This of course is extremely widely used in many settings, including software updates, web pages, financial transactions, and more.

Definition 9.7 — Digital Signatures. A triple of algorithms (G, S, V) is a chosen-message-attack secure *digital signature scheme* if it satisfies the following:

- On input 1^n , the probabilistic *key generation* algorithm G outputs a pair (s, v) of keys, where s is the private *signing key* and v is the public *verification* key.
- On input a message m and the signing key s , the signing algorithm S outputs a string $\sigma = S_s(m)$ such that with probability $1 - \text{negl}(n)$, $V_v(m, S_s(m)) = 1$.
- Every efficient adversary A wins the following game with at most negligible probability:
 1. The keys (s, v) are chosen by the key generation algorithm.
 2. The adversary gets the inputs $1^n, v$, and black box access to the signing algorithm $S_s(\cdot)$.
 3. The adversary *wins* if they output a pair (m^*, σ^*) such that m^* was *not* queried before to the signing algorithm and $V_v(m^*, \sigma^*) = 1$.



Strong unforgeability Just like for MACs (see Definition 4.4), our definition of security for digital signatures with respect to a chosen message attack does not preclude the ability of the adversary of producing a new signature for the same message that it has seen a signature of. Just like in MACs, people sometimes consider the notion of *strong unforgeability* which requires that it would not be possible for the adversary to produce a new message-signature pair (even if the message itself was queried before). Some signature schemes (such as the full domain hash and the DSA scheme) satisfy this stronger notion while others do not. However, just like MACs, it is possible to transform any signature with standard security into a signature that satisfies this stronger unforgeability condition.

9.3.5 The Digital Signature Algorithm (DSA)

The Diffie-Hellman protocol can be turned into a signature scheme. This was first done by ElGamal, and a variant of his scheme was developed by the NSA and standardized by NIST as the Digital Signature Algorithm (DSA) standard. When based on an elliptic curve this is known as ECDSA. The starting point is the following generic idea of how to turn an encryption scheme into an *identification protocol*.

If Alice published a public encryption key e , then one natural approach for Alice to prove her identity to Bob is as follows. Bob will send an encryption $c = E_e(x)$ of some random message $x \leftarrow_R \{0, 1\}^n$ to Alice, and Alice will send $x' = D_d(c)$ back. If $x = x'$ then she has proven that she can decrypt ciphertexts encrypted with e , and so Bob can be assured that she is the rightful owner of the public key e .

However, this falls short of a signature scheme in two aspects:

- This is only an identification protocol and does not allow Alice to endorse a particular message m .
- This is an *interactive* protocol, and so Alice cannot generate a static signature based on m that can be verified by any party without further interaction.

The first issue is not so significant, since we can always have the ciphertext be an encryption of $x = H(m)$ where H is some hash function presumed to behave as a random oracle. (We do *not* want to simply run this protocol with $x = m$. Can you see why?)

The second issue is more serious. We could imagine Alice trying to run this protocol on her own by generating the ciphertext and then decrypting it, and then sending over the transcript to Bob. But this does not really prove that she knows the corresponding private key. After all, even without knowing d , any party can generate a ciphertext c and its corresponding decryption. The idea behind the DSA protocol is that we require Alice to generate a ciphertext c and its decryption satisfying some additional extra conditions, which would prove that Alice truly knew the secret key.

DSA Signatures: The DSA signature algorithm works as follows: (See also Section 12.5.2 in the KL book)

- *Key generation:* Pick generator g for \mathbb{G} and $a \in \{0, \dots, |\mathbb{G}| - 1\}$ and let $h = g^a$. Pick $H : \{0, 1\}^\ell \rightarrow \mathbb{G}$ and $F : \mathbb{G} \rightarrow \mathbb{G}$ to be some functions that can be thought of as “hash functions”.¹⁰ The public key is (g, h) (as well as the functions H, F) and secret key is a .

¹⁰ It is a bit cumbersome, but not so hard, to transform functions that map strings to strings to functions whose domain or range are group elements. As noted in the KL book, in the actual DSA protocol F is *not* a cryptographic hash function but rather some very simple function that is still assumed to be “good enough” for security.

- *Signature:* To sign a message m with the key a , pick b at random, and let $f = g^b$, and then let $\sigma = b^{-1}[H(m) + a \cdot F(f)]$ where all computation is done modulo $|\mathbb{G}|$. The signature is (f, σ) .
- *Verification:* To verify a signature (f, σ) on a message m , check that $s \neq 0$ and $f^\sigma = g^{H(m)}h^{F(f)}$.

P You should pause here and verify that this is indeed a valid signature scheme, in the sense that for every m , $V_s(m, S_s(m)) = 1$.

Very roughly speaking, the idea behind security is that on one hand s does not reveal information about b and a because this is “masked” by the “random” value $H(m)$. On the other hand, if an adversary is able to come up with valid signatures then at least if we treated H and F as oracles, then if the signature passes verification then (by taking log to the base of g) the answers x, y of these oracles will satisfy $bs = x + ay$ which means that sufficiently many such equations should be enough to recover the discrete log a .

P Before seeing the actual proof, it is a very good exercise to try to see how to convert the intuition above into a formal proof.

Theorem 9.8 — Random-Oracle Model Security of DSA signatures. Suppose that the discrete logarithm assumption holds for the group \mathbb{G} . Then the DSA signature with \mathbb{G} is secure when H, F are modeled as random oracles.

Proof. Suppose, towards the sake of contradiction, that there was a T -time adversary A that succeeds with probability ϵ in a chosen message attack against the DSA scheme. We will show that there is an adversary that can compute the discrete logarithm with running time and probability polynomially related to T and ϵ respectively.

Recall that in a chosen message attack in the random oracle model, the adversary interacts with a signature oracle, and oracles that compute the functions F and H . For starters, we consider the following experiment CMA' where in the chosen message attack we replace the signature box with the following “fake signature oracle” and “fake function F oracle”. On input a message m , the fake box will choose σ, r at random in $\{0, \dots, p-1\}$ (where $p = |\mathbb{G}|$), and compute

$$f = (g^{H(m)}h^r)^{\sigma^{-1}} \pmod{p} \quad (9.2)$$

and output h . We will then record the value $F(f) = r$ and answer r on future queries to F . If we've already answered before $F(f)$ to be a different value then we halt the experiment and output an error. We claim that the adversary's chance of succeeding in CMA' is computationally indistinguishable from its chance of succeeding in the original CMA experiment. Indeed, since we choose the value $r = F(f)$ at random, as long as we don't repeat a value f that was queried before, the function F is completely random. But since the adversary makes at most T queries, and each f is chosen according to Eq. (9.2), which yields a random element the group \mathbb{G} (which has size roughly 2^n), the probability that f is repeated is at most $T/|\mathbb{G}|$ which is negligible. Now we computed σ in the fake box as a random value, but we can also compute σ as equalling $b^{-1}(H(m) + ar) \pmod p$, where $b = \log_g f \pmod \mathbb{G}$ is uniform as well, and so the distribution of the signature (f, σ) is identical to the distribution by a real box.

Note that we can simulate the result of the experiment CMA' without access to the value a such that $h = g^a$. We now transform an algorithm A' that manages to forge a signature in the CMA' experiment into an algorithm that given \mathbb{G}, g, g^a manages to recover a .

We let (m^*, f^*, σ^*) be the message and signature that the adversary A' outputs at the end of a successful attack. We can assume without loss of generality that f^* is queried to the F oracle at some point during the attack. (For example, by modifying A' to make this query just before she outputs the final signature.) So, we split into two cases:

Case I: The value $F(f^*)$ is first queried by the signature box.

Case II: The value $F(f^*)$ is first queried by the adversary.

If Case I happens with non negligible probability, then we know that the value f^* is queried when producing the signature (f^*, σ) for some message $m \neq m^*$, and so we know the following two equations hold:

$$g^{H(m)} h^{F(f^*)} = (f^*)^\sigma \quad (9.3)$$

and

$$g^{H(m^*)} h^{F(f^*)} = (f^*)^{\sigma^*} \quad (9.4)$$

Taking logs we get the following equations on $a = \log_g h$ and $b = \log_g f^*$:

$$H(m) + aF(f^*) = b\sigma \quad (9.5)$$

and

$$H(m^*) + aF(f^*) = b\sigma^* \quad (9.6)$$

or

$$b = (H(m^*) - H(m))(\sigma - \sigma^*)^{-1} \pmod{p} \quad (9.7)$$

since all of the values $H(m^*), H(m), \sigma, \sigma^*$ are known, this means we can compute b , and hence also recover the unknown value a .

If Case II happens, then we split it into two cases as well. **Case IIa** is that this happens and $F(f^*)$ is queried before $H(m^*)$ is queried, and **Case IIb** is that this happens and $F(f^*)$ is queried after $H(m^*)$ is queried.

We start by considering the setting that **Case IIa** happens with non-negligible probability ϵ . By the averaging argument there are some $t' < t \in \{1, \dots, T\}$ such that with probability at least ϵ/T^2 , f^* is queried by the adversary at the t' -th query and m^* is queried by the adversary at its t -th query. We run the CMA' experiment *twice*, using the same randomness up until the $t-1$ -th query and independent randomness from then onwards. With probability at least $(\epsilon/T^2)^2$, both experiments will result in a successful forge, and since f^* was queried before at stage $t' < t$, we get the following equations

$$H_1(m^*) + aF(f^*) = b\sigma \quad (9.8)$$

and

$$H_2(m^*) + aF(f^*) = b\sigma^* \quad (9.9)$$

where $H_1(m^*)$ and $H_2(m^*)$ are the answers of H to the query m^* in the first and second time we run the experiment. (The answers of F to f^* are the same since this happens before the t -th step). As before, we can use this to recover $a = \log_g h$.

If **Case IIb** happens with non-negligible probability $\epsilon > 0$. Then again by the averaging argument there are some $t < t' \in \{1, \dots, T\}$ such that with probability at least ϵ/T^2 , m^* is queried by the adversary at the t -th query and f^* is queried by the adversary at its t' -th query. We run the CMA' experiment *twice*, using the same randomness up until the $t'-1$ -th query and independent randomness from then onwards. This time we will get the two equations

$$H(m^*) + aF_1(f^*) = b\sigma \quad (9.10)$$

and

$$H(m^*) + aF_2(f^*) = b\sigma^* \quad (9.11)$$

where $F_1(f^*)$ and $F_2(f^*)$ are our two answers in the first and second experiment, and now we can use this to learn $a = b(\sigma - \sigma^*)(F_1(f^*) - F_2(f^*))^{-1}$.

The bottom line is that we obtain a probabilistic polynomial time algorithm that on input \mathbb{G}, g, g^a recovers a with non-negligible probability, hence violating the assumption that the discrete log problem is hard for the group \mathbb{G} . ■



Non-random oracle model security In this lecture both our encryption scheme and digital signature schemes were not proven secure under a well stated computational assumption, but rather used the random oracle model heuristic. However, it is known how to obtain schemes that do not rely on this heuristic, and we will see such schemes later on in this course.

9.4 Putting everything together - security in practice.

Let us discuss briefly how public key cryptography is used to secure web traffic through the SSL/TLS protocol that we all use when we use `https://` URLs. The security this achieves is quite amazing. No matter what wired or wireless network you are using, no matter what country you are in, as long as your device (e.g., phone/laptop/etc..) and the server you are talking to (e.g., Google, Amazon, Microsoft etc.) is functioning properly, you can communicate securely without any party in the middle able to either learn or modify the contents of your interaction.¹¹

In the web setting, there are *servers* who have public keys, and *users* who generally don't have such keys. Ideally, as a user, you should already know the public keys of all the entities you communicate with e.g., `amazon.com`, `google.com`, etc. However, how are you going to learn those public keys? The traditional answer was that because they are *public* these keys are much easier to communicate and the servers could even post them as ads on the *New York Times*. Of course these days everyone reads the *Times* through `nytimes.com` and so this seems like a chicken-and-egg type of problem.

The solution goes back again to the quote of Archimedes of "Give me a fulcrum, and I shall move the world". The idea is that trust can be *transitive*. Suppose you have a Mac. Then you have already trusted Apple with quite a bit of your personal information, and so you might be fine if this Mac came pre-installed with the Apple public key which you trust to be authentic. Now, suppose that you want to communicate with `Amazon.com`. Now, *you* might not know the correct

¹¹ They are able to know that such an interaction took place and the amount of bits exchanged. Preventing these kind of attacks is more subtle and approaches for solutions are known as *steganography* and *anonymous routing*.

public key for Amazon, but *Apple* surely does. So Apple can supply Amazon with a signed message to the effect of

```
"I Apple certify that the public key of Amazon.com is 30
82 01 0a 02 82 01 01 00 94 9f 2e fd 07 63 33
53 b1 be e5 d4 21 9d 86 43 70 0e b5 7c 45 bb
ab d1 ff 1f b1 48 7b a3 4f be c7 9d 0f 5c 0b
f1 dc 13 15 b0 10 e3 e3 b6 21 0b 40 b0 a3 ca
af cc bf 69 fb 99 b8 7b 22 32 bc 1b 17 72 5b
e5 e5 77 2b bd 65 d0 03 00 10 e7 09 04 e5 f2
f5 36 e3 1b 0a 09 fd 4e 1b 5a 1e d7 da 3c 20
18 93 92 e3 a1 bd 0d 03 7c b6 4f 3a a4 e5 e5
ed 19 97 f1 dc ec 9e 9f 0a 5e 2c ae f1 3a e5
5a d4 ca f6 06 cf 24 37 34 d6 fa c4 4c 7e 0e
12 08 a5 c9 dc cd a0 84 89 35 1b ca c6 9e 3c
65 04 32 36 c7 21 07 f4 55 32 75 62 a6 b3 d6
ba e4 63 dc 01 3a 09 18 f5 c7 49 bc 36 37 52
60 23 c2 10 82 7a 60 ec 9d 21 a6 b4 da 44 d7
52 ac c4 2e 3d fe 89 93 d1 ba 7e dc 25 55 46
50 56 3e e0 f0 8e c3 0a aa 68 70 af ec 90 25
2b 56 f6 fb f7 49 15 60 50 c8 b4 c4 78 7a 6b
97 ec cd 27 2e 88 98 92 db 02 03 01 00 01"
```

Such a message is known as a *certificate*, and it allows you to extend your trust in Apple to a trust in Amazon. Now when your browser communicates with amazon, it can request this message, and if it is not present not continue with the interaction or at least display some warning. Clearly a person in the middle can stop this message from travelling and hence not allow the interaction to continue, but they cannot *spoof* the message and send a certificate for their own public key, unless they know Apple's secret key. (In today's actual implementation, for various business and other reasons, the trusted keys that come pre-installed in browsers and devices do not belong to Apple or Microsoft but rather to particular companies such as *Verisign* known as *certificate authorities*. The security of these certificate authorities' private key is crucial to the security of the whole protocol, and it has been [attacked before](#).)

Using certificates, we can assume that Bob the user has the public verification key v of Alice the server. Now Alice can send Bob also a public *encryption* key e , which is authenticated by v and hence guaranteed to be correct.¹² Once Bob knows Alice's public key they are in business- he can use that to send an encryption of some private key k which they can then use for all the rest of their communication.

This is in a very high level the SSL/TLS protocol, but there are many details inside it including the exact security notions needed

¹² If this key is *ephemeral*- generated on the spot for this interaction and deleted afterward- then this has the benefit of ensuring the *forward secrecy* property that even if some entity that is in the habit of recording all communication later finds out Alice's private verification key, then it still will not be able to decrypt the information. In applied crypto circles this property is somewhat misnamed as "perfect forward secrecy" and associated with the Diffie-Hellman key exchange (or its elliptic curves variants), since in those protocols there is not much additional overhead for implementing it (see [this blog post](#)). The importance of forward security was emphasized by the discovery of the [Heartbleed](#) vulnerability (see [this paper](#)) that allowed via a buffer-overflow attack in OpenSSL to learn the private key of the server.

from the encryption, how the two parties negotiate *which* cryptographic algorithm to use, and more. All these issues can and have been used for attacks on this protocol. For two recent discussions see [this blog post](#) and [this website](#).



Figure 9.2: When you connect to a webpage protected by SSL/TLS, the Browser displays information on the certificate's authenticity

Example: Here is the list of certificate authorities that were trusted by default (as of spring 2016) by Mozilla products: Actalis, Amazon, AS Sertifitseerimiskeskuse (SK), Atos, Autoridad de Certificacion FirmaProfesional, Bypass, CA Disig a.s., Camerfirma, CerticA;mara S.A., Certigna, Certinomis, certSIGN, China Financial Certification Authority (CFCA), China Internet Network Information Center (CNNIC), Chunghwa Telecom Corporation, Comodo, ComSign, Consorci AdministraciA;3 Oberta de Catalunya (Consorci AOC, CATCert), Cybertrust Japan / JCSI, D-TRUST, Deutscher Sparkassen Verlag GmbH (S-TRUST, DSV-Gruppe), DigiCert, DocuSign (OpenTrust/Keynectis), e-tugra, EDICOM, Entrust, GlobalSign, GoDaddy, Government of France (ANSSI, DCSSI), Government of Hong Kong (SAR), Hongkong Post, Certizen, Government of Japan, Ministry of Internal Affairs and Communications, Government of Spain, Autoritat de CertificaciA;3 de la Comunitat Valenciana (ACCV), Government of Taiwan, Government Root Certification Authority (GRCA), Government of The Netherlands, PKIoverheid, Government of Turkey, Kamu Sertifikasyon Merkezi (Kamu SM), HARICA, IdenTrust, Izenpe S.A., Microsec e-SzignA;3 CA, NetLock Ltd., PROCERT, QuoVadis, RSA the Security Division of EMC, SECOM Trust Systems Co. Ltd., Start Commercial (StartCom)

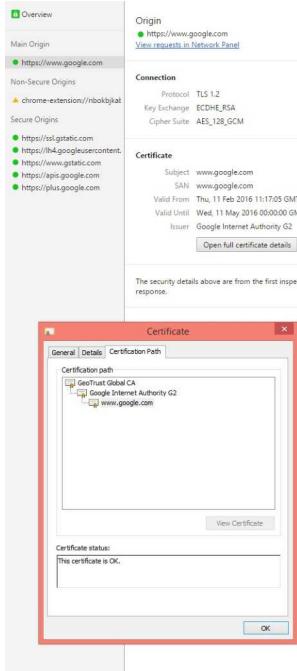


Figure 9.3: The cipher and certificate used by “Google.com”. Note that Google has a 2048bit RSA signature key which it then uses to authenticate an elliptic curve based Diffie-Hellman key exchange protocol to create session keys for the block cipher AES with 128 bit key in Galois Counter Mode.

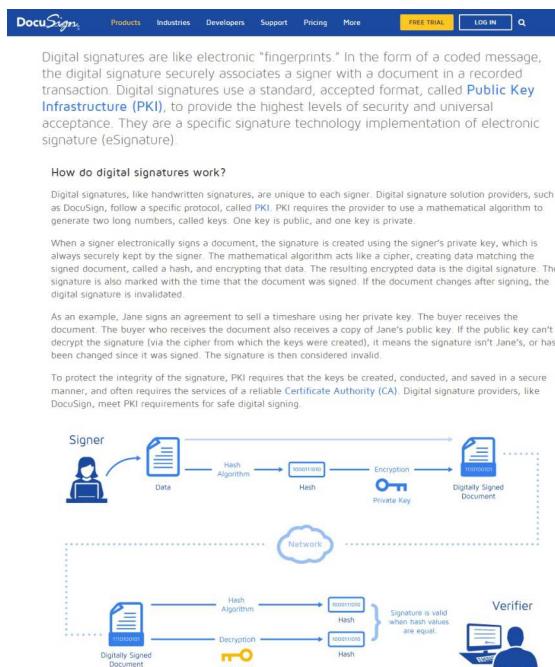


Figure 9.4: Digital signatures and other forms of electronic signatures are legally binding in many jurisdictions. This is some material from the website of the electronic signing company DocuSign

Ltd., Swisscom (Switzerland) Ltd, SwissSign AG, Symantec / GeoTrust, Symantec / Thawte, Symantec / VeriSign, T-Systems International GmbH (Deutsche Telekom), Taiwan-CA Inc. (TWCA), TeliaSonera, Trend Micro, Trustis, Trustwave, Turk-Trust, Unizeto Certum, Visa, Web.com, Wells Fargo Bank N.A., WiSeKey, WoSign CA Limited

9.5 Appendix: An alternative proof of the density of primes

I record here an alternative way to show that the fraction of primes in $[2^n]$ is $\Omega(1/n)$.¹³

Lemma 9.9 The probability that a random n bit number is prime is at least $\Omega(1/n)$.

¹³ It might be that the two ways are more or less the same, as if we open up the polynomial $(1-x)^k x^k$ we get the binomial coefficients.

Proof. Let $N = 2^n$. We need to show that the number of primes between 1 and N is at least $\Omega(N/\log N)$. Consider the number $\binom{2N}{N} = \frac{2N!}{N!N!}$. By Stirling's formula we know that $\log \binom{2N}{N} = (1 - o(1))2N$ and in particular $N \leq \log \binom{2N}{N} \leq 2N$. Also, by the formula using factorials, all the prime factors of $\binom{2N}{N}$ are between 0 and $2N$, and each factor P cannot appear more than $k = \lfloor \frac{\log 2N}{\log P} \rfloor$ times. Indeed, for every N , the number of times P appears in the factorization of $N!$ is $\sum_i \lfloor \frac{N}{P^i} \rfloor$, since we get $\lfloor \frac{N}{P} \rfloor$ times a factor P in the factorizations of $\{1, \dots, N\}$, $\lfloor \frac{N}{P^2} \rfloor$ times a factor of the form P^2 , etc... Thus the number of times P appears in the factorization of $\binom{2N}{N} = \frac{(2N)!}{N!N!}$ is equal to $\sum_i \lfloor \frac{2N}{P^i} \rfloor - 2 \lfloor \frac{N}{P} \rfloor$: a sum of at most k elements (since $P^{k+1} > 2N$) each of which is either 0 or 1.

Thus, $\binom{2N}{N} \leq \prod_{\substack{1 \leq P \leq 2N \\ P \text{ prime}}} P^{\lfloor \frac{\log 2N}{\log P} \rfloor}$. Taking logs we get that

$$N \leq \log \binom{2N}{N} \tag{9.12}$$

$$\leq \sum_{P \text{ prime} \in [2n]} \lfloor \frac{\log 2N}{\log P} \rfloor \log P \tag{9.13}$$

$$\leq \sum_{P \text{ prime} \in [2n]} \log 2N \tag{9.14}$$

establishing that the number of primes in $[1, N]$ is $\Omega(\frac{N}{\log N})$. ■

10

Concrete candidates for public key crypto

In the previous lecture we talked about *public key cryptography* and saw the Diffie Hellman system and the DSA signature scheme. In this lecture, we will see the RSA trapdoor function and how to use it for both encryptions and signatures.

10.1 Some number theory.

(See [Shoup's excellent and freely available book](#) for extensive coverage of these and many other topics.)

For every number m , we define \mathbb{Z}_m to be the set $\{0, \dots, m - 1\}$ with the addition and multiplication operations modulo m . When two elements are in \mathbb{Z}_n then we will always assume that all operations are done modulo m unless stated otherwise. We let $\mathbb{Z}_m^* = \{a \in \mathbb{Z}_m : \gcd(a, m) = 1\}$. Note that m is prime if and only if $|\mathbb{Z}_m^*| = m - 1$. For every $a \in \mathbb{Z}_m^*$ we can find using the extended gcd algorithm an element b (typically denoted as a^{-1}) such that $ab = 1$ (can you see why?). The set \mathbb{Z}_m^* is an abelian group with the multiplication operation, and hence by the observations of the previous lecture, $a^{|\mathbb{Z}_m^*|} = 1$ for every $a \in \mathbb{Z}_m^*$. In the case that m is prime, this result is known as “Fermat’s Little Theorem” and is typically stated as $a^{p-1} = 1 \pmod{p}$ for every $a \neq 0$.



Note on n bits vs a number n One aspect that is often confusing in number-theoretic based cryptography, is that one needs to always keep track whether we are talking about “big” numbers or “small” numbers. In many cases in crypto, we use n to talk about our key size or security parameter, in which case we think of n as a “small” number of size 100 – 1000 or

so. However, when we work with \mathbb{Z}_m^* we often think of m as a “big” number having about 100 – 1000 digits; that is m would be roughly 2^{100} to 2^{1000} or so. I will try to reserve the notation n for “small” numbers but may sometimes forget to do so, and other descriptions of RSA etc.. often use n for “big” numbers. It is important that whenever you see a number x , you make sure you have a sense whether it is a “small” number (in which case $\text{poly}(x)$ time is considered efficient) or whether it is a “large” number (in which case only $\text{poly}(\log(x))$ time would be considered efficient).

R

The number m vs the message m In much of this course we use m to denote a string which is our plaintext message to be encrypted or authenticated. In the context of integer factoring, it is convenient to use $m = pq$ as the composite number that is to be factored. To keep things interesting (or more honestly, because I keep running out of letters) in this lecture we will have both usages of m (though hopefully not in the same theorem or definition!). When we talk about factoring, RSA, and Rabin, then we will use m as the composite number, while in the context of the abstract trapdoor-permutation based encryption and signatures we will use m for the message. When you see an instance of m , make sure you understand what is its usage.

10.1.1 Primality testing

One procedure we often need is to find a prime of n bits. The typical way people do it is by choosing a random n -bit number p , and testing whether it is prime. We showed in the previous lecture that a random n bit number is prime with probability at least $\Omega(1/n^2)$ (in fact the probability is $\frac{1+o(1)}{\ln n}$ by the **Prime Number Theorem**). We now discuss how we can test for primality.

Theorem 10.1 — Primality Testing. There is an $\text{poly}(n)$ -time algorithm to test whether a given n -bit number is prime or composite.

Theorem 10.1 was first shown in 1970’s by Solovay, Strassen, Miller and Rabin via a *probabilistic* algorithm (that can make a mistake with probability exponentially small in the number of coins it uses), and in a 2002 breakthrough Agrawal, Kayal, and Saxena gave a *deterministic* polynomial time algorithm for the same problem.

Lemma 10.2 There is a probabilistic polynomial time algorithm A that on input a number m , if m is prime A outputs YES with probability 1 and if A is not even a “pseudoprime” it outputs NO with probability at least $1/2$. (The definition of “pseudo-prime” will be clarified in the proof below.)

Proof. The algorithm is very simple and is based on Fermat’s Little Theorem: on input m , pick a random $a \in \{2, \dots, m-1\}$, and if $\gcd(a, m) \neq 1$ or $a^{m-1} \neq 1 \pmod{m}$ return NO and otherwise return YES.

By Fermat’s little theorem, the algorithm will always return YES on a prime m . We define a “pseudoprime” to be a non-prime number m such that $a^{m-1} = 1 \pmod{m}$ for all a such that $\gcd(a, m) = 1$.

If n is *not* a pseudoprime then the set $S = \{a \in \mathbb{Z}_m^* : a^{m-1} = 1\}$ is a strict subset of \mathbb{Z}_m^* . But it is easy to see that S is a group and hence $|S|$ must divide $|\mathbb{Z}_n^*|$ and hence in particular it must be the case that $|S| < |\mathbb{Z}_n^*|/2$ and so with probability at least $1/2$ the algorithm will output NO. ■

Lemma 10.2 its own might not seem very meaningful since it’s not clear how many pseudoprimes are there. However, it turns out these pseudoprimes, also known as “Carmichael numbers”, are much less prevalent than the primes, specifically, there are about $N/2^{-\Theta(\log N / \log \log N)}$ pseudoprimes between 1 and N . If we choose a random number $m \in [2^n]$ and output it if and only if the algorithm of **Lemma 10.2** algorithm outputs YES (otherwise resampling), then the probability we make a mistake and output a pseudoprime is equal to the ratio of the set of pseudoprimes in $[2^n]$ to the set of primes in $[2^n]$. Since there are $\Omega(2^n/n)$ primes in $[2^n]$, this ratio is $\frac{n}{2^{-\Omega(n/\log n)}}$ which is a negligible quantity. Moreover, as mentioned above, there are better algorithms that succeed for *all* numbers.

In contrast to *testing* if a number is prime or composite, there is no known efficient algorithm to actually *find* the factorization of a composite number. The best known algorithms run in time roughly $2^{\tilde{O}(n^{1/3})}$ where n is the number of bits.

10.1.2 Fields

If p is a prime then \mathbb{Z}_p is a *field* which means it is closed under addition and multiplication and has 0 and 1 elements. One property of a field is the following:

Theorem 10.3 — Fundamental Theorem of Algebra, mod p version. If f is a nonzero polynomial of degree d over \mathbb{Z}_p then there are at most d distinct inputs x such that $f(x) = 0$.

(If you're curious why, you can see that the task of, given x_1, \dots, x_{d+1} finding the coefficients for a polynomial vanishing on the x_i 's amounts to solving a linear system in $d + 1$ variables with $d + 1$ equations that are independent due to the non-singularity of the Vandermonde matrix.)

In particular every $x \in \mathbb{Z}_p$ has at most two *square roots* (numbers s such that $s^2 = x \pmod p$). In fact, just like over the reals, every $x \in \mathbb{Z}_p$ either has no square roots or exactly two square roots of the form $\pm s$.

We can efficiently find square roots modulo a prime. In fact, the following result is known:

Theorem 10.4 — Finding roots. There is a probabilistic $\text{poly}(\log p, d)$ time algorithm to find the roots of a degree d polynomial over \mathbb{Z}_p .

This is a special case of the problem of factoring polynomials over finite fields, shown in 1967 by Berlekamp and on which much other work has been done; see Chapter 20 in [Shoup](#)).

10.1.3 Chinese remainder theorem

Suppose that $m = pq$ is a product of two primes. In this case \mathbb{Z}_m^* does not contain *all* the numbers from 1 to $m - 1$. Indeed, all the numbers of the form $p, 2p, 3p, \dots, (q - 1)p$ and $q, 2q, \dots, (p - 1)q$ will have non-trivial g.c.d. with m . There are exactly $q - 1 + p - 1$ such numbers (because p and q are prime all the numbers of the forms above are distinct). Hence $|\mathbb{Z}_m^*| = m - 1 - (p - 1) - (q - 1) = pq - p - q + 1 = (p - 1)(q - 1)$.

Note that $|\mathbb{Z}_m^*| = |\mathbb{Z}_p^*| \cdot |\mathbb{Z}_q^*|$. It turns out this is no accident:

Theorem 10.5 — Chinese Remainder Theorem (CRT). If $m = pq$ then there is an isomorphism $\varphi : \mathbb{Z}_m^* \rightarrow \mathbb{Z}_p^* \times \mathbb{Z}_q^*$. That is, φ is one to one and onto and maps $x \in \mathbb{Z}_m^*$ into a pair $(\varphi_1(x), \varphi_2(x)) \in \mathbb{Z}_p^* \times \mathbb{Z}_q^*$ such that for every $x, y \in \mathbb{Z}_m^*$:

$$* \quad \varphi_1(x + y) = \varphi_1(x) + \varphi_1(y) \pmod p$$

- * $\varphi_2(x + y) = \varphi_2(x) + \varphi_2(y) \pmod{q}$
- * $\varphi_1(x \cdot y) = \varphi_1(x) \cdot \varphi_1(y) \pmod{p}$
- * $\varphi_2(x \cdot y) = \varphi_2(x) \cdot \varphi_2(y) \pmod{q}$

Proof. φ simply maps $x \in \mathbb{Z}_m^*$ to the pair $(x \pmod{p}, x \pmod{q})$.

Verifying that it satisfies all desired properties is a good exercise.

QED ■

In particular, for every polynomial $f()$ and $x \in \mathbb{Z}_m^*$, $f(x) = 0 \pmod{m}$ iff $f(x) = 0 \pmod{p}$ and $f(x) = 0 \pmod{q}$. Therefore finding the roots of a polynomial $f()$ modulo a composite m is easy *if you know m's factorization*. However, if you don't know the factorization then this is hard. In particular, extracting square roots is as hard as finding out the factors:

Theorem 10.6 — Square root extraction implies factoring. Suppose and there is an efficient algorithm A such that for every $m \in \mathbb{N}$ and $a \in \mathbb{Z}_m^*$, $A(m, a^2 \pmod{m}) = b$ such that $a^2 = b^2 \pmod{m}$. Then, there is an efficient algorithm to recover p, q from m .

Proof. Suppose that there is such an algorithm A . Using the CRT we can define $f : \mathbb{Z}_p^* \times \mathbb{Z}_q^* \rightarrow \mathbb{Z}_p^* \times \mathbb{Z}_q^*$ as $f(x, y) = \varphi(A(\varphi^{-1}(x^2, y^2)))$ for all $x \in \mathbb{Z}_p^*$ and $y \in \mathbb{Z}_q^*$. Now, for any x, y let $(x', y') = f(x, y)$. Since $x^2 = x'^2 \pmod{p}$ and $y^2 = y'^2 \pmod{q}$ we know that $x' \in \{\pm x\}$ and $y' \in \{\pm y\}$. Since flipping signs doesn't change the value of $(x', y') = f(x, y)$, by flipping one or both of the signs of x or y we can ensure that $x' = x$ and $y' = -y$. Hence $(x, y) - (x', y') = (0, 2y)$. In other words, if $c = \varphi^{-1}(x - x', y - y')$ then $c = 0 \pmod{p}$ but $c \neq 0 \pmod{q}$ which in particular means that the greatest common divisor of c and m is q . So, by taking $\gcd(A(\varphi^{-1}(x, y)), m)$ we will find q , from which we can find $p = m/q$.

This almost works, but there is a question of how can we find $\varphi^{-1}(x, y)$, given that we don't know p and q ? The crucial observation is that we don't need to. We can simply pick a value a at random in $\{1, \dots, m\}$. With very high probability (namely $(p - 1 + q - 1)/pq$) a will be in \mathbb{Z}_m^* , and so we can imagine this process as equivalent to the process of taking a random $x \in \mathbb{Z}_p^*$, a random $y \in \mathbb{Z}_q^*$ and then flipping the signs of x and y randomly and taking $a = \varphi(x, y)$. By the arguments above with probability at least $1/4$, it will hold that $\gcd(a - A(a^2), m)$ will equal q . ■

Note that this argument generalizes to work even if the algorithm

A is an *average case* algorithm that only succeeds in finding a square root for a significant fraction of the inputs. This observation is crucial for cryptographic applications.

10.1.4 The RSA and Rabin functions

We are now ready to describe the RSA and Rabin trapdoor functions:

Definition 10.7 — RSA function. Given a number $m = pq$ and e such that $\gcd((p-1)(q-1), e) = 1$, the *RSA function* w.r.t m and e is the map $f_{m,e} : \mathbb{Z}_m^* \rightarrow \mathbb{Z}_m^*$ such that $RSA_{m,e}(x) = x^e \pmod{m}$.

Definition 10.8 — Rabin function. Given a number $m = pq$, the *Rabin function* w.r.t. m , is the map $Rabin_m : \mathbb{Z}_m^* \rightarrow \mathbb{Z}_m^*$ such that $Rabin_m(x) = x^2 \pmod{m}$.

Note that both maps can be computed in polynomial time. Using the Chinese Remainder Theorem and [Theorem 10.4](#), we know that both functions can be *inverted* efficiently if we know the factorization.¹

However [Theorem 10.4](#) is a much too big of a Hammer to invert the RSA and Rabin functions, and there are direct and simple inversion algorithms (see homework exercises). By [Theorem 10.6](#), inverting the Rabin function amounts to factoring m . No such result is known for the RSA function, but there is no better algorithm known to attack it than proceeding via factorization of m . The RSA function has the advantage that it is a *permutation* over \mathbb{Z}_m^* :

Lemma 10.9 $RSA_{m,e}$ is one to one over \mathbb{Z}_m^* .

Proof. Suppose that $RSA_{m,e}(a) = RSA_{m,e}(a')$. By the CRT, it means that there is $(x, y) \neq (x', y') \in \mathbb{Z}_p^* \times \mathbb{Z}_q^*$ such that $x^e = x'^e \pmod{p}$ and $y^e = y'^e \pmod{q}$. But if that's the case we get that $(xx'^{-1})^e = 1 \pmod{p}$ and $(yy'^{-1})^e = 1 \pmod{q}$. But this means that e has to be a multiple of the *order* of xx'^{-1} and yy'^{-1} (at least one of which is *not* 1 and hence has order > 1). But since the order always divides the group size, this implies that e has to have non-trivial gcd with either $|\mathbb{Z}_p^*|$ or $|\mathbb{Z}_q^*|$ and hence with $(p-1)(q-1)$. ■

¹ Using [Theorem 10.4](#) to invert the function requires e to be not too large. However, as we will see below it turns out that using the factorization we can invert the RSA function for every e . Also, in practice people often use a small value for e (sometimes as small as $e = 3$) for reasons of efficiency.



Plain/Textbook RSA The RSA trapdoor function is known also as “plain” or “textbook” RSA encryption. This is because initially Diffie and Hellman

(and following them, RSA) thought of an encryption scheme as a deterministic procedure and so considered simply encrypting a message x by applying $ESA_{m,e}(x)$. Today however we know that it is insecure to use a trapdoor function directly as an encryption scheme without adding some randomization.

10.1.5 Abstraction: trapdoor permutations

We can abstract away the particular construction of the RSA and Rabin functions to talk about a general *trapdoor permutation family*. We make the following definition

Definition 10.10 — Trapdoor permutation. A *trapdoor permutation family* (TDP) is a family of functions $\{p_k\}$ such that for every $k \in \{0,1\}^n$, the function p_k is a permutation on $\{0,1\}^n$ and:

- * There is a *key generation algorithm* G such that on input 1^n it outputs a pair (k, τ) such that the maps $k, x \mapsto p_k(x)$ and $\tau, y \mapsto p_k^{-1}(y)$ are efficiently computable.
- For every efficient adversary A , $\mathbb{P}_{(k,\tau) \leftarrow_R G(1^n), y \in \{0,1\}^n} [A(k, y) = p_k^{-1}(y)] < negl(n)$.

R

Domain of permutations The RSA function is not a permutation over the set of strings but rather over \mathbb{Z}_m^* for some $m = pq$. However, if we find primes p, q in the interval $[2^{n/2}(1 - negl(n)), 2^{n/2}]$, then m will be in the interval $[2^n(1 - negl(n)), 2^n]$ and hence \mathbb{Z}_m^* (which has size $pq - p - q + 1 = 2^n(1 - negl(n))$) can be thought of as essentially identical to $\{0,1\}^n$, since we will always pick elements from $\{0,1\}^n$ at random and hence they will be in \mathbb{Z}_m^* with probability $1 - negl(n)$. It is widely believed that for every sufficiently large n there is a prime in the interval $[2^n - poly(n), 2^n]$ (this follows from the *Extended Riemann Hypothesis*) and Baker, Harman and Pintz proved that there is a prime in the interval $[2^n - 2^{0.6n}, 2^n]$.²

10.1.6 Public key encryption from trapdoor permutations

Here is how we can get a public key encryption from a trapdoor permutation scheme $\{p_k\}$.

² Another, more minor issue is that the description of the key might not have the same length as $\log m$; I defined them to be the same for simplicity of notation, and this can be ensured via some padding and concatenation tricks.

TDP-based public key encryption (TDPENC):

- *Key generation:* Run the key generation algorithm of the TDP to get (k, τ) . k is the *public encryption key* and τ is the *secret decryption key*.
- *Encryption:* To encrypt a message m with key $k \in \{0,1\}^n$, choose $x \in \{0,1\}^n$ and output $(p_k(x), H(x) \oplus m)$ where $H : \{0,1\}^n \rightarrow \{0,1\}^\ell$ is a hash function we model as a random oracle.
- *Decryption:* To decrypt the ciphertext (y, z) with key τ , output $m = H(p_k^{-1}(y)) \oplus z$.

Please verify that you understand why TDPENC is a *valid* encryption scheme, in the sense that decryption of an encryption of m yields m .

Theorem 10.11 — Public key encryption from trapdoor permutations. If $\{p_k\}$ is a secure TDP and H is a random oracle then TDPENC is a CPA secure public key encryption scheme.

Proof. Suppose, towards the sake of contradiction, that there is a polynomial-size adversary A that succeeds in the CPA game of TDPENC (with access to a random oracle H) with non-negligible advantage ϵ over half. We will use A to design an algorithm I that inverts the trapdoor permutation.

Recall that the CPA game works as follows:

- The adversary A gets as input a key $k \in \{0,1\}^n$.
- The algorithm A makes some polynomial amount of computation and $T_1 = \text{poly}(n)$ queries to the random oracle H and produces a pair of messages $m_0, m_1 \in \{0,1\}^\ell$.
- The “challenger” chooses $b^* \leftarrow_R \{0,1\}$, chooses $x^* \leftarrow_R \{0,1\}^n$ and computes the ciphertext $(y^* = p_k(x^*), z^* = H(x^*) \oplus m_{b^*})$ which is an encryption of m_{b^*} .
- The adversary A gets (y^*, z^*) as input, makes some additional polynomial amount of computation and $T_2 = \text{poly}(n)$ queries to H , and then outputs b .
- The adversary *wins* if $b = b^*$.

We make the following claim:

CLAIM: With probability at least ϵ , the adversary A will make the query x^* to the random oracle.

PROOF: Suppose otherwise. We will prove the claim using the “forgetful gnome” technique as used in the Boneh Shoup book. By the “lazy evaluation” paradigm, we can imagine that queries to H are answered by a “faithful gnome” that whenever presented with a new query x , chooses a uniform and independent value $w \leftarrow_R \{0,1\}^\ell$ as a response, and then records that $H(x) = w$ to use that as answers for future queries.

Now consider the experiment where in the challenge part we use a “forgetful gnome” that answers $H(x^*)$ by a uniform and independent string $w^* \leftarrow_R \{0,1\}^\ell$ and *does not* record the answer for future queries. In the “forgetful experiment”, the second component of the ciphertext $z^* = w^* \oplus m_{b^*}$ is distributed uniformly in $\{0,1\}^\ell$ and independently from all other random choices, regardless of whether $b^* = 0$ or $b^* = 1$. Hence in this “forgetful experiment” the adversary gets no information about b^* and its probability of winning is at most $1/2$. But the forgetful experiment is identical to the actual experiment if the value x^* is only queried to H once. Apart from the query of x^* by the challenger, all other queries to H are made by the adversary. Under our assumption, the adversary makes the query x^* with probability at most ϵ , and conditioned on this not happening the two experiments are identical. Since the probability of winning in the forgetful experiment is at most $1/2$, the probability of winning in the overall experiment is less than $1/2 + \epsilon$, thus yielding a contradiction and establishing the claim. (These kind of analyses on sample spaces can be confusing; See Fig. 10.1 for a graphical illustration of this argument.)

Given the claim, we can now construct our inverter algorithm I as follows:

- The input to I is the key k to the trapdoor permutation and $y^* = p_k(x^*)$. The goal of I is to output x^* .
- The inverter simulates the adversary in a CPA attack, answering all its queries to the oracle H by random values if they are new or the previously supplied answers if they were asked before. Whenever the adversary makes a query x to H , I checks if $p_h(x) = y^*$ and if so halts and outputs x .
- When the time comes to produce the challenge, the inverter I chooses z^* at random and provides the adversary with (y^*, z^*) where $z^* = w^* \oplus m_{b^*}$.³
- The inverter continues the simulation again halting and outputting x if the adversary makes the query x such that $p_k(x) = y^*$ to H .

³ It would have been equivalent to answer the adversary with a uniformly chosen z^* in $\{0,1\}^\ell$, can you see why?

We claim that up to the point we halt, the experiment is identical to the actual attack. Indeed, since p_k is a permutation, we know that if the time came to produce the challenge and we have not halted, then the query x^* has not been made yet to H . Therefore we are free to choose an independent random value w^* as the value $H(x^*)$. (Our inverter does not know what the value x^* is, but this does not matter for this argument: can you see why?) Therefore, since by the claim the adversary will make the query x^* to H with probability at least ϵ , our inverter will succeed with the same probability. ■

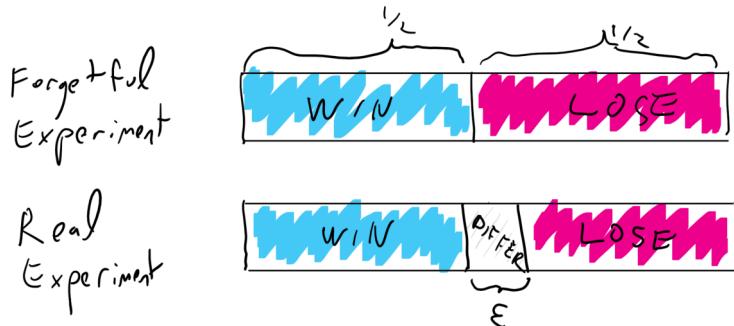


Figure 10.1: In the proof of security of TDPENC, we show that if the assumption of the claim is violated, the “forgetful experiment” is identical to the real experiment with probability larger $1 - \epsilon$. In such a case, even if all that probability mass was on the points in the sample space where the adversary in the forgetful experiment will lose and the adversary of the real experiment will win, the probability of winning in the latter experiment would still be less than $1/2 + \epsilon$.

P This proof of [Theorem 10.11](#) is not very long but it is somewhat subtle. Please re-read it and make sure you understand it. I also recommend you look at the version of the same proof in Boneh Shoup: Theorem 11.2 in Section 11.4 (“Encryption based on a trapdoor function scheme”).

R **Security without random oracles** We do *not* need to use a random oracle to get security in this scheme, especially if ℓ is sufficiently short. We can replace $H()$ with a hash function of specific properties known as a *hard core* construction; this was first shown by Goldreich and Levin.

10.1.7 Digital signatures from trapdoor permutations

Here is how we can get digital signatures from trapdoor permutations $\{p_k\}$. This is known as the “full domain hash” signatures.

Full domain hash signatures (FDHSIG):

- *Key generation:* Run the key generation algorithm of the TDP to get (k, τ) . k is the *public verification key* and τ is the *secret signing key*.
- *Signing:* To sign a message m with key τ , we output $p_k^{-1}(H(m))$ where $H : \{0,1\}^* \rightarrow \{0,1\}^n$ is a hash function modeled as a random oracle.
- *Verification:* To verify a message-signature pair (m, x) we check that $p_k(x) = H(m)$.

We now prove the security of full domain hash:

Theorem 10.12 — Full domain hash security. If $\{p_k\}$ is a secure TDP and H is a random oracle then FDHSIG is chosen message attack secure digital signature scheme.

Proof. Suppose towards the sake of contradiction that there is a polynomial-sized adversary A that succeeds in a chosen message attack with non-negligible probability $\epsilon > 0$. We will construct an inverter I for the trapdoor permutation collection that succeeds with non-negligible probability as well.

Recall that in a chosen message attack the adversary makes T queries m_1, \dots, m_T to its signing box which are interspersed with T' queries $m'_1, \dots, m'_{T'}$ to the random oracle H . We can assume without loss of generality (by modifying the adversary and at most doubling the number of queries) that the adversary always queries the message m_i to the random oracle *before* it queries it to the signing box, though it can also make additional queries to the random oracle (and hence in particular $T' \geq T$). At the end of the attack the adversary outputs with probability ϵ a pair (x^*, m^*) such that m^* was not queried to the signing box and $p_k(x^*) = H(m^*)$.

Our inverter I works as follows:

- **Input:** k and $y^* = p_k(y^*)$. Goal is to output x^* .
- I will guess at random t^* which is the step in which the adversary will query to H the message m^* that it is eventually going to forge in. With probability $1/T'$ the guess will be correct.
- I simulates the execution of A . Except for step t^* , whenever A makes a new query m to the random oracle, I will choose a ran-

$\text{dom } x \leftarrow_R \{0,1\}^n$, compute $y = p_k(x)$ and designate $H(m) = y$. In step t^* , when the adversary makes the query m^* , the inverter I will return $H(m^*) = y^*$. I will record the values (x, y) and so in particular will always know $p_k^{-1}(H(m))$ for every $H(m) \neq y^*$ that it returned as answer from its oracle on query m .

- When A makes the query m to the signature box, then since m was queried before to H , if $m \neq m^*$ then I returns $x = p_k^{-1}(H(m))$ using its records. If $m = m^*$ then I halts and outputs “failure”.
- At the end of the game, the adversary outputs (m^*, x^*) . If $p_k(x^*) = y^*$ then I outputs x^* .

We claim that, conditioned on the probability $\geq \epsilon/T'$ event that the adversary is successful and the final message m^* is the one queried in step t^* , we provide a perfect simulation of the actual game. Indeed, while in an actual game, the value $y = H(m)$ will be chosen independently at random in $\{0,1\}^n$, this is equivalent to choosing $x \leftarrow_R \{0,1\}^n$ and letting $y = p_k(x)$. After all, a permutation applied to the uniform distribution is uniform.

Therefore with probability at least ϵ/T' the inverter I will output x^* such that $p_k(x^*) = y^*$ hence succeeding in the inverter. ■



Once again, this proof is somewhat subtle. I recommend you also read the version of this proof in Section 13.4 of Boneh-Shoup.



Hash and sign There is another reason to use hash functions with signatures. By combining a collision-resistant hash function $h : \{0,1\}^* \rightarrow \{0,1\}^\ell$ with a signature scheme (S, V) for ℓ -length messages, we can obtain a signature for arbitrary length messages by defining $S'_s(m) = S_s(h(m))$ and $V'_v(m, \sigma) = V_v(h(m), \sigma)$.

10.2 Hardcore bits and security without random oracles

To be completed.

11

Lattice based crypto

Lattice based public key encryption (and its cousins known as knapsack and coding based encryption) have almost as long a history as discrete logarithm and factoring based schemes. Already in 1976, right after the Diffie-Hellman key exchange was discovered (and before RSA), Ralph Merkle was working on building public key encryption from the NP hard *knapsack* problem (see [Diffie's recollection](#)). This can be thought of as the task of solving a linear equation of the form $Ax = y$ (where A is a given matrix, y is a given vector, and the unknown are x) over the real numbers but with the additional constraint that x must be either 0 or 1. His proposal evolved into the Merkle-Hellman system proposed in 1978 (which was broken in 1984).

McEliece proposed in 1978 a system based on the difficulty of the decoding problem for general linear codes. This is the task of solving *noisy linear equations* where one is given A and y such that $y = Ax + e$ for a “small” error vector e , and needs to recover x . Crucially, here we work in a finite field, such as working modulo q for some prime q (that can even be 2) rather than over the reals or rationals. There are special matrices A^* for which we know how to solve this problem efficiently: these are known as efficiently decodable [error correcting codes](#). McEliece suggested a scheme where the key generator lets A be a “scrambled” version of a special A^* (based on the [Goppa algebraic geometric code](#)). So, someone that knows the scrambling could solve the problem, but (hopefully) someone that doesn’t know it wouldn’t. McEliece’s system has so far not been broken.

In a 1996 breakthrough, Ajtai showed a *private key* scheme based on integer lattices that had a very curious property- its security could be based on the assumption that certain problems were only hard in the *worst case*, and moreover variants of these problems were known

to be NP hard. This re-ignited the hope that we could perhaps realize the old dream of basing crypto on the mere assumption that $P \neq NP$. Alas, we now understand that there are fundamental barriers to this approach.

Nevertheless, Ajtai's work attracted significant interest, and within a year both Ajtai and Dwork, as well as Goldreich, Goldwasser and Halevi came up with lattice based constructions for *public key* encryption (the former based also on *worst case* assumptions). At about the same time, Hoffstein, Pipher, and Silverman came up with their NTRU public key system which is based on stronger assumptions but offers better performance, and they started a company around it together with Daniel Lieman.

You may note that I haven't yet said what *lattices* are; we will do so later, but for now if you simply think of questions involving linear equations modulo some prime q , you will get enough of the intuition that you need. (The lattice viewpoint is more geometric, and we'll discuss it more below; it was first used to *attack* cryptosystems and in particular break the Merkle-Hellman knapsack scheme and many of its variants.)

Lattice based cryptography has captured a lot of attention recently from both theory and practice. In the theory side, many cool new constructions are now based on lattice based cryptography, and chief among them fully homomorphic encryption, as well as indistinguishability obfuscation (though the latter's security's foundations are still far less solid). On the applied side, the steady advances in the technology of quantum computers have finally gotten practitioners worried about RSA, Diffie Hellman and Elliptic Curves. While current constructions for quantum computers are nowhere near being able to, say, factor larger numbers that can be done classically (or even than can be done by hand), given that it takes many years to develop new standards and get them deployed, many believe the effort to transition away from these factoring/dlog based schemes should start today (or perhaps should have started several years ago). The NSA has [suggested](#) that it plans to initiate the process to "transition to quantum resistant algorithms in the not too distant future"; see also this [very interesting FAQ](#) on this topic.

Cryptography has the peculiar/unfortunate feature that if a machine is built that can factor large integers in 20 years, it can still be used to break the communication we transmit *today*, provided this communication was recorded. So, if you have some data that you expect you'd want still kept secret in 20 years (as many government and commercial entities do), you might have reasons to worry. Cur-

rently lattice based cryptography is the only real “game in town” for potentially quantum-resistant public key encryption schemes.

Lattice based cryptography is a huge area, and in this lecture and this course we only touch on few aspects of it. I highly recommend [Chris Peikert’s Survey](#) for a much more in depth treatment of this area.

11.1 A world without Gaussian elimination

The general approach people used to get a public key encryption is to obtain a hard computational problem with some mathematical *structure*. We’ve seen this in the *discrete logarithm* problem, where the task is to invert the map $a \mapsto g^a \pmod{p}$, and the integer factoring problem, where the task is to invert the map $a, b \mapsto a \cdot b$. Perhaps the simplest structure to consider is the task of solving linear equations.

Pretend that we didn’t know of Gaussian elimination,¹ and that if we picked a “generic” matrix A then the map $x \mapsto Ax$ would be hard to invert. (Here and elsewhere, our default interpretation of a vector x is as a *column* vector, and hence if x is n dimensional and A is $m \times n$ then Ax is m dimensional. We use x^\top to denote the row vector obtained by *transposing* x .) Could we use that to get a public key encryption scheme?

Here is a concrete approach. Let us fix some prime q (think of it as polynomial size, e.g., q is smaller than 1024 or so, though people can and sometimes do consider q of exponential size), and all computation below will be done modulo q . The secret key is a vector $x \in \mathbb{Z}_q^n$, and the public key is (A, y) where A is a random $m \times n$ matrix with entries in \mathbb{Z}_q and $y = Ax$. Under our assumption, it is hard to recover the secret key from the public key, but how do we use the public key to encrypt?

The crucial observation is that even if we don’t know how to solve linear equations, we can still combine several equations to get new ones. To keep things simple, let’s consider the case of encrypting a single bit.



If you have a CPA secure public key encryption scheme for single bit messages then you can extend it to a CPA secure encryption scheme for messages of any length. Can you see why?

¹ Despite the name, *Gaussian elimination* has been known to Chinese mathematicians since 150BC or so, and was popularized in the west through the 1670 notes of Isaac Newton.

We think of the public key as the set of equations $\langle a_1, x \rangle = y_1, \dots, \langle a_m, x \rangle = y_m$ in the unknown variables x . The idea is that to encrypt the value 0 we will generate a new *correct* equation on x , while to encrypt the value 1 we will generate an *incorrect* equation. To decrypt a ciphertext $(a, \sigma) \in \mathbb{Z}_q^{n+1}$, we think of it as an equation of the form $\langle a, x \rangle = \sigma$ and output 1 if and only if the equation is correct.

How does the encrypting algorithm, that does not know x , get a correct or incorrect equation on demand? One way would be to simply take two equations $\langle a_i, x \rangle = y_i$ and $\langle a_j, x \rangle = y_j$ and add them together to get the equation $\langle a_i + a_j, x \rangle = y_i + y_j$. This equation is correct and so one can use it to encrypt 0, while to encrypt 1 we simply add some fixed nonzero number $\alpha \in \mathbb{Z}_q$ to the right hand side to get the incorrect equation $\langle a_i + a_j, x \rangle = y_i + y_j + \alpha$. However, even if it's hard to solve for x given the equations, an attacker (who also knows the public key (A, y)) can try itself all pairs of equations and do the same thing.

Our solution for this is simple- just add more equations! If the encryptor adds a random subset of equations then there are 2^m possibilities for that, and an attacker can't guess them all. Thus, at least intuitively, the following encryption scheme would be "secure" in the Gaussian-elimination free world of attackers that haven't taken freshman linear algebra:

Scheme LwoE-ENC: Public key encryption under the hardness of "learning linear equations without errors".

- *Key generation:* Pick random $m \times n$ matrix A over \mathbb{Z}_q , and $x \leftarrow_R \mathbb{Z}_q^n$, the secret key is x and the public key is (A, y) where $y = Ax$.
- *Encryption:* To encrypt a message $b \in \{0, 1\}$, pick $w \in \{0, 1\}^m$ and output $w^\top A, \langle w, y \rangle + \alpha b$ for some fixed nonzero $\alpha \in \mathbb{Z}_q$.
- *Decryption:* To decrypt a ciphertext (a, σ) , output 0 iff $\langle a, x \rangle = \sigma$.



Please stop here and make sure that you see why this is a valid encryption, and this description corresponds to the previous one; as usual all calculations are done modulo q .

11.2 Security in the real world.

Like it or not (and cryptographers typically don't) Gaussian elimination is possible in the real world and the scheme above is completely insecure. However, the Gaussian elimination algorithm is extremely *brittle*.

Errors tend to be amplified when you combine equations. This is usually thought of as a bad thing, and numerical analysis is much about dealing with issue. However, from the cryptographic point of view, these errors can be our saving grace and enable us to salvage the security of the ridiculous scheme above.

To see why Gaussian elimination is brittle, let us recall how it works. Think of $m = n$ for simplicity. Given equations $Ax = y$ in the unknown variables x , the goal of Gaussian elimination is to transform them into the equations $Ix = y'$ where I is the identity matrix (and hence the solution is simply $x = y'$). Recall how we do it: by rearranging and scaling, we can assume that the top left corner of A is equal to 1, and then we add the first equation to the other equations (scaled appropriately) to zero out the first entry in all the other rows of A (i.e., make the first column of A equal to $(1, 0, \dots, 0)$) and continue onwards to the second column and so on and so forth.

Now, suppose that the equations were *noisy*, in the sense that we added to y a vector $e \in \mathbb{Z}_q^m$ such that $|e_i| < \delta q$ for every i .² Even ignoring the effect of the scaling step, simply adding the first equation to the rest of the equations would typically tend to increase the relative error of equations $2, \dots, m$ from $\approx \delta$ to $\approx 2\delta$. Now, when we repeat the process, we increase the error of equations $3, \dots, m$ from $\approx 2\delta$ to $\approx 4\delta$, and we see that by the time we're done dealing with about $n/2$ variables, the remaining equations have error level roughly $2^{n/2}\delta$. So, unless δ was truly tiny (and q truly big, in which case the difference between working in \mathbb{Z}_q and simply working with integers or rationals disappears), the resulting equations have the form $Ix = y' + e'$ where e' is so big that we get no information on x .

The *Learning With Errors (LWE)* conjecture is that this is *inherent*:

Conjecture (Learning with Errors, Regev 2005):

Let $q = q(n)$ and $\delta = \delta(n)$ be some functions. The *Learning with Error (LWE) conjecture with respect to q, δ* , is that for every polynomial-time adversary E and $m = \text{poly}(n)$, the probability that $E(A, Ax + e) = x$ is negligible, where A is a random $m \times n$ matrix in \mathbb{Z}_q , x is random in \mathbb{Z}_q^n , and $e \in \mathbb{Z}_q^m$

² Over \mathbb{Z}_q , we can think of $q - 1$ also as the number -1 , and so on. Thus if $a \in \mathbb{Z}_q$, we define $|a|$ to be the minimum of a and $q - a$. This ensures the absolute value satisfies the natural property of $|a| = |-a|$.

is a random noise vector with magnitude δq .³
 The LWE conjecture is that for every polynomial $p(n)$ there is some polynomial $q(n)$ such that LWE holds with respect to $q(n)$ and $\delta(n) = 1/p(n)$.⁴

11.3 Search to decision

It turns out that if the LWE is hard, then it is even hard to distinguish between random equations and nearly correct ones:

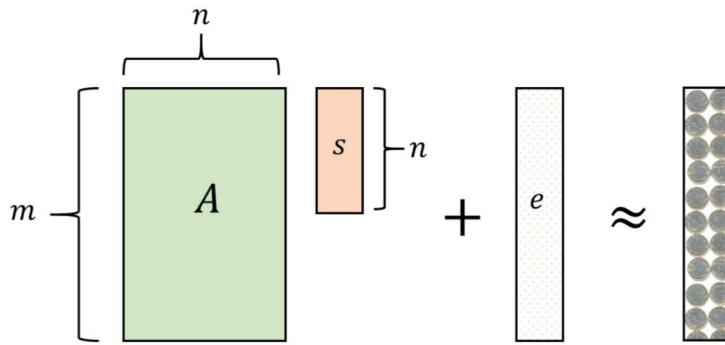


Figure 11.1: The search to decision reduction (Theorem 11.1) implies that under the LWE conjecture, for every $m = \text{poly}(n)$, if we choose and fix a random $m \times n$ matrix A over \mathbb{Z}_q , the distribution $Ax + e$ is indistinguishable from a random vector in \mathbb{Z}_q^m , where x is a random vector in \mathbb{Z}_q^n and e is a random “short” vector in \mathbb{Z}_q^m . The two distributions are indistinguishable even to an adversary that knows A .

Theorem 11.1 — Search to decision reduction for LWE. If the LWE conjecture is true then for every $q = \text{poly}(n)$ and $\delta = 1/\text{poly}(n)$ and $m = \text{poly}(n)$, the following two distributions are computationally indistinguishable:

- $\{(A, Ax + e)\}$ where A is random $m \times n$ matrix in \mathbb{Z}_q , x is random in \mathbb{Z}_q^n and $e \in \mathbb{Z}_q^m$ is random noise vector of magnitude δ .
- $\{(A, y)\}$ where A is random $m \times n$ matrix in \mathbb{Z}_q and y is random in \mathbb{Z}_q^m

Proof. Suppose that we had a decisional adversary D that succeeds in distinguishing the two distributions above with bias ϵ . For example, suppose that D outputs 1 with probability $p + \epsilon$ on inputs from the first distribution, and outputs 1 with probability p on inputs from the second distribution.

⁴ One can think of e as chosen by simply letting every coordinate be chosen at random in $\{-\delta q, -\delta q + 1, \dots, +\delta q\}$. For technical reasons, we sometimes consider other distributions and in particular the *discrete Gaussian* distribution which is obtained by letting every coordinate of e be an independent Gaussian random variable with standard deviation δq , conditioned on it being an integer. (A closely related distribution is obtained by picking such a Gaussian random variable and then rounding it to the nearest integer.)

⁴ People sometimes also consider variants where both $p(n)$ and $q(n)$ can be as large as exponential.

We will show how we can use this to obtain a polynomial-time algorithm S that on input m noisy equations on x and a value $a \in \mathbb{Z}_q$, will learn with high probability whether or not the first coordinate of x equals a . Clearly, we can repeat this for all the possible q values of a to learn the first coordinate exactly, and then continue in this way to learn all coordinates.

Our algorithm S gets as input the pair (A, y) where $y = Ax + e$ and we need to decide whether $x_1 = a$. Now consider the instance $A + (r\|0^m\|\cdots\|0^m), y + ar$, where r is a random vector in \mathbb{Z}_q^m and the matrix $(r\|0^m\|\cdots\|0^m)$ is simply the matrix with first column equal to r and all other columns equal to 0. If A is random then $A + r\|0^m\|\cdots\|0^m$ is random as well. Now note that $Ax + (r\|0^m\|\cdots\|0^m)x = Ax + x_1r$ and hence if $x_1 = a$ then we still have an input of the same form $(A', A'x + e)$.

In contrast, we claim that if if $x_1 \neq a$ then the distribution (A', y') where $A' = A + (r\|0^m\|\cdots\|0^m)$ and $y' = Ax + e + ar$ is identical to the uniform distribution over a random uniformly chosen matrix A' and a random and independent uniformly chosen vector y' . Indeed, we can write this distribution as (A', y') where A' is chosen uniformly at random, and $y' = A'x + e + (a - x_1)r$ where r is a random and independent vector. (Can you see why?) Since $a - x_1 \neq 0$, this amounts to adding a random and independent vector r' to y' , which means that the distribution (A', y') is uniform and independent.

Hence if we send the input (A', y') to our the decision algorithm D , then we would get 1 with probability $p + \epsilon$ if $x_1 = a$ and an output of 1 with probability p otherwise.

Now the crucial observation is that if our decision algorithm D requires m equations to succeed with bias ϵ , we can use $100mn/\epsilon^2$ equations (which is still polynomial) to invoke it $100n/\epsilon^2$ times. This allows us to distinguish with probability $1 - 2^{-n}$ between the case that D outputs 1 with probability $p + \epsilon$ and the case that it outputs 1 with probability p (this follows from the Chernoff bound; can you see why?). Hence by using polynomially more samples than the decision algorithm D , we get a search algorithm S that can actually recover x . ■

11.4 An LWE based encryption scheme

We can now show the secure variant of our original encryption scheme:

LWE-based encryption LWEENC:

- *Parameters:* Let $\delta(n) = 1/n^4$ and let $q = \text{poly}(n)$ be a prime such that LWE holds w.r.t. q, δ . We let $m = n^2 \log q$.
- *Key generation:* Pick $x \in \mathbb{Z}_q^n$. The private key is x and the public key is (A, y) with $y = Ax + e$ with e a δ -noise vector and A a random $m \times n$ matrix.
- *Encrypt:* To encrypt $b \in \{0, 1\}$ given the key (A, y) , pick $w \in \{0, 1\}^m$ and output $w^\top A, \langle w, y \rangle + b\lfloor q/2 \rfloor$ (all computations are done in \mathbb{Z}_q).
- *Decrypt:* To decrypt (a, σ) , output 0 iff $|\langle a, x \rangle - \sigma| < q/10$.

(P)

The scheme LWEENC is also described in Fig. 11.2 with slightly different notation. I highly recommend you stop and verify you understand why the two descriptions are equivalent.

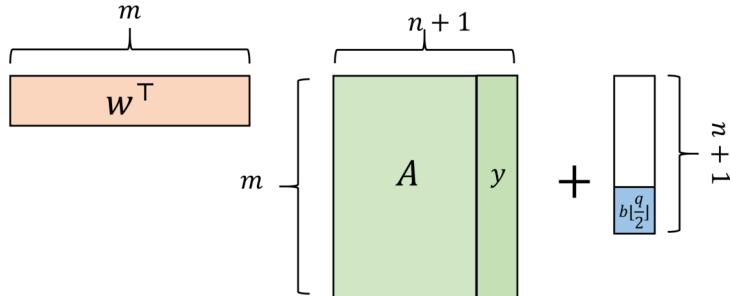


Figure 11.2: In the encryption scheme LWEENC, the public key is a matrix $A' = (A|y)$, where $y = As + e$ and s is the secret key. To encrypt a bit b we choose a random $w \leftarrow_R \{0, 1\}^m$, and output $w^\top A' + (0, \dots, 0, b\lfloor \frac{q}{2} \rfloor)$. We decrypt $c \in \mathbb{Z}_q^{n+1}$ to zero with key s iff $|\langle c, (s, -1) \rangle| \leq q/10$ where the inner product is done modulo q .

Unlike our typical schemes, here it is not immediately clear that this encryption is valid, in the sense that the decrypting an encryption of b returns the value b . But this is the case:

Lemma 11.2 With high probability, the decryption of the encryption of b equals b .

Proof. $\langle w^\top A, x \rangle = \langle w, Ax \rangle$. Hence, if $y = Ax + e$ then $\langle w, y \rangle = \langle w^\top A, x \rangle + \langle w, e \rangle$. But since every coordinate of w is either 0 or 1, $|\langle w, e \rangle| < \delta mq < q/10$ for our choice of parameters.⁵ So, we get that if $a = w^\top A$ and $\sigma = \langle w, y \rangle + b\lfloor q/2 \rfloor$ then $\sigma - \langle a, x \rangle = \langle w, e \rangle + b\lfloor q/2 \rfloor$ which will be smaller than $q/10$ iff $b = 0$. ■

⁵ In fact, due to the fact that the signs of the error vector's entries are different, we expect the errors to have significant cancellations and hence we would expect $|\langle w, e \rangle|$ to only be roughly of magnitude $\sqrt{m}\delta q$, but this is not crucial for our discussions.

We now prove security of the LWE based encryption:

Theorem 11.3 — CPA security of LWEENC. If the LWE conjecture is true then LWEENC is CPA secure.

For a public key encryption scheme with messages that are just bits, CPA security means that an encryption of 0 is indistinguishable from an encryption of 1, even given the public key. Thus [Theorem 11.3](#) will follow from the following lemma:

Lemma 11.4 Let q, m, δ be set as in LWEENC. Then assuming the LWE conjecture, the following distributions are computationally indistinguishable:

- D : The distribution over four-tuples of the form $(A, y, w^\top A, \langle w, y \rangle)$ where A is uniform in $\mathbb{Z}_q^{m \times n}$, x is uniform in \mathbb{Z}_q^n , $e \in \mathbb{Z}_q$ is chosen with $e_i \in \{-\delta q, \dots, +\delta q\}$, $y = Ax + e$, and w is uniform in $\{0, 1\}^m$.
- \bar{D} : The distribution over four-tuples (A, y', a, σ) where all entries are uniform: A is uniform in $\mathbb{Z}_q^{m \times n}$, y' is uniform in \mathbb{Z}_q^m , a is uniform in \mathbb{Z}_q^n and σ is uniform in \mathbb{Z}_q .



You should stop here and verify that (i) You understand the statement of [Lemma 11.4](#) and (ii) you understand why this lemma implies [Theorem 11.3](#). The idea is that [Lemma 11.4](#) shows that the concatenation of the public key and encryption of 0 is indistinguishable from something that is completely random. You can then use it to show that the concatenation of the public key and encryption of 1 is indistinguishable from the same thing, and then finish using the hybrid argument.

We now prove [Lemma 11.4](#), which will complete the proof of [Theorem 11.3](#).

Proof of Lemma 11.4. Define D to be the distribution $(A, y, w^\top A, \langle w, y \rangle)$ as in the lemma's statement (i.e., $y = Ax + e$ for some x, e chosen as above). Define D' to be the distribution $(A, y', w^\top A, \langle w, y' \rangle)$ where y' is chosen uniformly in \mathbb{Z}_q^m . We claim that D' is computationally indistinguishable from D under the LWE conjecture. Indeed by [Theorem 11.1](#) (search to decision reduction) this conjecture implies that the distribution X over pairs (A, y) with $y = Ax + e$ is indistinguishable from the distribution X' over pairs (A, y') where y' is uniform. But if there was some polynomial-time algorithm T distinguishing D from D' then we can design a randomized

polynomial-time algorithm T' distinguishing X from X' with the same advantage by setting $T'(A, y) = T(A, y, w^\top A, \langle w, y \rangle)$ for random $w \leftarrow_R \{0, 1\}^m$.

We will finish the proof by showing that the distribution D' is *statistically indistinguishable* (i.e., has negligible total variation distance) from \bar{D} . This follows from the following claim:

CLAIM: Suppose that $m > 100n \log q$. If A' is a random $m \times n + 1$ matrix in \mathbb{Z}_q^m , then with probability at least $1 - 2^{-n}$ over the choice of A' , the distribution $Z_{A'}$ over \mathbb{Z}_q^n which is obtained by choosing w at random in $\{0, 1\}^m$ and outputting $w^\top A'$ has at most 2^{-n} statistical distance from the uniform distribution over \mathbb{Z}_q^{n+1} .

Note that the randomness used for the distribution $Z_{A'}$ is only obtained by the choice of w , and *not* by the choice of A' that is fixed. (This passes a basic “sanity check” since w has m random bits, while the uniform distribution over \mathbb{Z}_q^n requires $n \log q \ll m$ random bits, and hence $Z_{A'}$ at least has a “fighting chance” in being statistically close to it.) Another way to state the same claim is that the pair $(A', w^\top A')$ is statistically indistinguishable from the uniform distribution (A', z) where z is a vector chosen independently at random from \mathbb{Z}_q^n .

The claim completes the proof of the theorem, since letting A' be the matrix $(A|y)$ and $z = (a, \sigma)$, we see that the distribution D' , as the form (A', z) where A' is a uniformly random $m \times (n + 1)$ matrix and z is sampled from $Z_{A'}$ (i.e., $z = w^\top A'$ where w is uniformly chosen in $\{0, 1\}^m$). Hence this means that the statistical distance of D' from \bar{D} (where all elements are uniform) is $O(2^{-n})$. (Please make sure you understand this reasoning!)

We will not do the whole proof of the claim (which uses the mod q version of the **leftover hash lemma** which we mentioned before and is also “Wikipedia-able”) but the idea is simple. For every $m \times (n + 1)$ matrix A' over \mathbb{Z}_q , define $h_{A'} : \mathbb{Z}_q^m \rightarrow \mathbb{Z}_q^n$ to be the map $h_{A'}(w) = w^\top A'$. This collection can be shown to be a “good” hash function collection in some specific technical sense, which in particular implies that for every distribution D with much more than $n \log q$ bits of min-entropy, with all but negligible probability over the choice of A' , $h_{A'}(D)$ is statistically indistinguishable from the uniform distribution. Now when we choose w at random in $\{0, 1\}^m$, it is coming from a distribution with m bits of entropy. If $m \gg (n + 1) \log q$, then because the output of this function is so much smaller than m , we expect it to be completely uniform, and this is what’s shown by the leftover hash lemma. ■

P The proof of [Theorem 11.3](#) is quite subtle and requires some re-reading and thought. To read more about this, you can look at the survey of Oded Regev, “[On the Learning with Error Problem](#)” Sections 3 and 4.

11.5 But what are lattices?

You can think of a lattice as a discrete version of a subspace. A lattice L is simply a discrete subset of \mathbb{R}^n such that if $u, v \in L$ and a, b are integers then $au + bv \in L$.⁶ A lattice is given by a basis which simply a matrix B such that every vector $u \in L$ is obtained as $u = Bx$ for some vector of integers x . It can be shown that we can assume without loss of generality that B is full dimensional and hence it's an n by n invertible matrix. Note that given a basis B we can generate vectors in L , as well as test whether a vector v is in L by testing if $B^{-1}v$ is an integer vector. There can be many different bases for the same lattice, and some of them are easier to work with than others (see [Fig. 11.3](#)).

⁶ By discrete we mean that points in L are isolated. One formal way to define it is that there is some $\epsilon > 0$ such that every distinct $u, v \in L$ are of distance at least ϵ from one another.

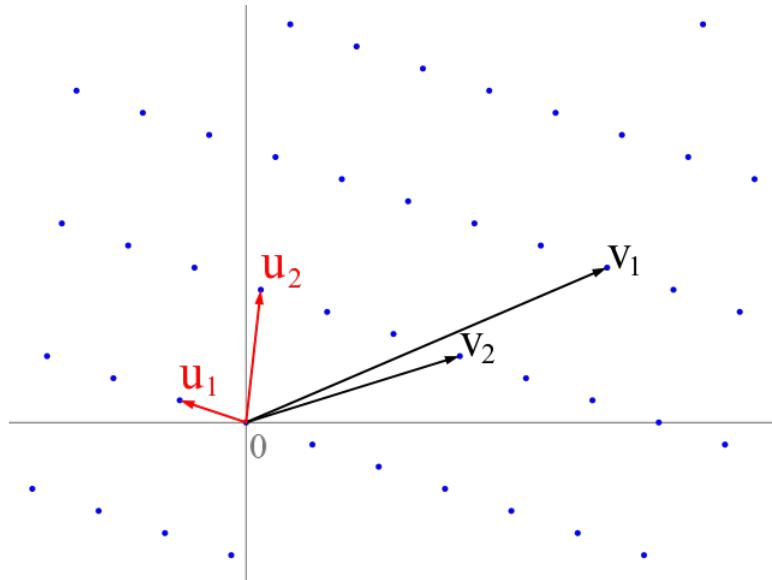


Figure 11.3: A lattice is a discrete subspace $L \subseteq \mathbb{R}^n$ that is closed under integer combinations. A basis for the lattice is a minimal set b_1, \dots, b_m (typically $m = n$) such that every $u \in L$ is an integer combination of b_1, \dots, b_m . The same lattice can have different bases. In this figure the lattice is a set of points in \mathbb{R}^2 , and the black vectors v_1, v_2 and the ref vectors u_1, u_2 are two alternative bases for it. Generally we consider the basis u_1, u_2 “better” since the vectors are shorter and it is less “skewed”.

Some classical computational questions on lattices are:

- *Shortest vector problem:* Given a basis B for L , find the nonzero

vector v with smallest norm in L .

- *Closest vector problem:* Given a basis B for L and a vector u that is *not* in L , find the closest vector to u in L .
- *Bounded distance decoding:* Given a basis B for L and a vector u of the form $u = v + e$ where v is in L , and e is a particularly short “error” vector (so in particular no other vector in the lattice is within distance $\|e\|$ to u), recover v . Note that this is a special case of the closest vector problem.

In particular, if V is a linear subspace of \mathbb{Z}_q^n , we can think of it also as a lattice \hat{V} of \mathbb{R}^n where we simply say that that a vector \hat{u} is in \hat{V} if all of \hat{u} ’s coordinates are integers and if we let $u_i = \hat{u}_i \pmod{q}$ then $u \in V$. The learning with error task of recovering x from $Ax + e$ can then be thought of as an instance of the bounded distance decoding problem for \hat{V} .

A natural algorithm to try to solve the *closest vector* and *bounded distance decoding* problems is that to take the vector u , express it in the basis B by computing $w = B^{-1}u$, and then round all the coordinates of w to obtain an integer vector \tilde{w} and let $v = B\tilde{w}$ be a vector in the lattice. If we have an extremely good basis L for the lattice then v may indeed be the closest vector in the lattice, but in other more “skewed” bases it can be extremely far from it.

11.6 Ring based lattices

One of the biggest issues with lattice based cryptosystem is the key size. In particular, the scheme above uses an $m \times n$ matrix where each entry takes $\log q$ bits to describe. (It also encrypts a single bit using a whole vector, but more efficient “multi-bit” variants are known.) Schemes using *ideal lattices* are an attempt to get more practical variants. These have very similar structure except that the matrix A chosen is not completely random but rather can be described by a single vector. One common variant is the following: we fix some polynomial p over \mathbb{Z}_q with degree n and then treat vectors in \mathbb{Z}_q^n as the coefficients of $n - 1$ degree polynomials and always work modulo this polynomial $p()$. (By this I mean that for every polynomial t of degree at least n we write t as $ps + r$ where p is the polynomial above, s is some polynomial and r is the “remainder” polynomial of degree $< n$; then $t \pmod{p} = r$.) Now for every fixed polynomial t , the operation A_t which is defined as $s \mapsto ts \pmod{p}$ is a linear operation mapping polynomials of degree at most $n - 1$ to polynomials of degree at most $n - 1$, or put another way, it is a

linear map over \mathbb{Z}_q^n . However, the map A_d can be described using the n coefficients of t as opposed to the n^2 description of a matrix. It also turns out that by using the Fast Fourier Transform we can evaluate this operation in roughly n steps as opposed to n^2 . The ideal lattice based cryptosystem use matrices of this form to save on key size and computation time. It is still unclear if this structure can be used for attacks; recent papers attacking principal ideal lattices have shown that one needs to be careful about this.

One ideal-lattice based system is the “[New Hope](#)” cryptosystem (see also [paper](#)) that has been experimented with by Google. People have also made highly optimized general (non ideal) lattice based constructions, see in particular the “[Frodo](#)” system (paper [here](#), can you guess what’s behind the name?). Both New Hope and Frodo have been submitted to the [NIST competition](#) to select a “post quantum” public key encryption standard.

12

Chosen ciphertext security for public key encryption

To be completed

13

Establishing secure connections over insecure channels

We've now compiled all the tools that are needed for the basic goal of cryptography (which is still being subverted quite often) allowing Alice and Bob to exchange messages assuring their integrity and confidentiality over a channel that is observed or controlled by an adversary. Our tools for achieving this goal are:

- Public key (aka asymmetric) encryption schemes.
- Public key (aka asymmetric) digital signatures schemes.
- Private key (aka symmetric) encryption schemes - block ciphers and stream ciphers.
- Private key (aka symmetric) message authentication codes and pseudorandom functions.
- Hash functions that are used both as ways to compress messages for authentication as well as key derivation and other tasks.

The notions of security we require from these building blocks can vary as well. For encryption schemes we talk about CPA (chosen plaintext attack) and CCA (chosen ciphertext attacks), for hash functions we talk about collision-resistance, being used (combined with keys) as pseudorandom functions, and then sometimes we simply model those as random oracles. Also, all of those tools require access to a source of randomness, and here we use hash functions as well for entropy extraction.

13.1 Cryptography's obsession with adjectives.

As we learn more and more cryptography we see more and more adjectives, every notion seems to have modifiers such as "non mal-

leable”, “leakage-resilient”, “identity based”, “concurrently secure”, “adaptive”, “non-interactive”, etc.. etc.... . Indeed, this motivated a parody web page of an [automatic crypto paper title generator](#). Unlike algorithms, where typically there are straightforward *quantitative* tradeoffs (e.g., faster is better), in cryptography there are many *qualitative* ways protocols can vary based on the assumptions they operate under and the notions of security they provide.

In particular, the following issues arise when considering the task of securely transmitting information between two parties Alice and Bob:

- **Infrastructure/setup assumptions:** What kind of setup can Alice and Bob rely upon? For example in the TLS protocol, typically Alice is a website and Bob is user; Using the infrastructure of certificate authorities, Bob has a trusted way to obtain Alice’s *public signature key*, while Alice doesn’t know anything about Bob. But there are many other variants as well. Alice and Bob could share a (low entropy) *password*. One of them might have some hardware token, or they might have a secure out of band channel (e.g., text messages) to transmit a short amount of information. There are even variants where the parties authenticate by something they *know*, with one recent example being the notion of *witness encryption* (Garg, Gentry, Sahai, and Waters) where one can encrypt information in a “digital time capsule” to be opened by anyone who, for example, finds a proof of the Riemann hypothesis.
- **Adversary access:** What kind of attacks do we need to protect against. The simplest setting is a *passive* eavesdropping adversary (often called “Eve”) but we sometimes consider an *active person-in-the-middle* attacks (sometimes called “Mallory”). We sometimes consider notions of *graceful recovery*. For example, if the adversary manages to hack into one of the parties then it can clearly read their communications from that time onwards, but we would want their past communication to be protected (a notion known as *forward secrecy*). If we rely on trusted infrastructure such as certificate authorities, we could ask what happens if the adversary breaks into those. Sometimes we rely on the security of several entities or secrets, and we want to consider adversaries that control *some* but not *all* of them, a notion known as *threshold cryptography*.
- **Interaction:** Do Alice and Bob get to interact and relay several messages back and forth or is it a “one shot” protocol? You may think that this is merely a question about efficiency but it turns out to be crucial for some applications. Sometimes Alice and Bob might not be two parties separated in space but the same party

separated in time. That is, Alice wishes to send a message to her future self by storing an encrypted and authenticated version of it on some media. In this case, absent a time machine, back and forth interaction between the two parties is obviously impossible.

- **Security goal:** The security goals of a protocol are usually stated in the negative- what does it mean for an adversary to *win* the security game. We typically want the adversary to learn absolutely no information about the secret beyond what she obviously can. For example, if we use a shared password chosen out of t possibilities, then we might need to allow the adversary $1/t$ success probability, but we wouldn't want her to get anything beyond $1/t + negl(n)$. In some settings, the adversary can obviously completely disconnect the communication channel between Alice and Bob, but we want her to be essentially limited to either dropping communication completely or letting it go by unmolested, and not have the ability to modify communication without detection. Then in some settings, such as in the case of steganography and anonymous routing, we would want the adversary not to find out even the fact that a conversation had taken place.

13.2 Basic Key Exchange protocol

The basic primitive for secure communication is a *key exchange* protocol, whose goal is to have Alice and Bob share a common random secret key $k \in \{0,1\}^n$. Once this is done, they can use a CCA secure / authenticated private-key encryption to communicate with confidentiality and integrity.

The canonical example of a basic key exchange protocol is the *Diffie Hellman* protocol. It uses as public parameters a group \mathbb{G} with generator g , and then follows the following steps:

1. Alice picks random $a \leftarrow_R \{0, \dots, |\mathbb{G}| - 1\}$ and sends $A = g^a$.
2. Bob picks random $b \leftarrow_R \{0, \dots, |\mathbb{G}| - 1\}$ and sends $B = g^b$.
3. They both set their key as $k = H(g^{ab})$ (which Alice computes as B^a and Bob computes as A^b), where H is some hash function.

Another variant is using an arbitrary public key encryption scheme such as RSA:

1. Alice generates keys (d, e) and sends e to Bob.
2. Bob picks random $k \leftarrow_R \{0, 1\}^m$ and sends $E_e(k)$ to Alice.

3. They both set their key to k (which Alice computes by decrypting Bob's ciphertext)

Under plausible assumptions, it can be shown that these protocols secure against a *passive* eavesdropping adversary Eve. The notion of security here means that, similar to encryption, if after observing the transcript Eve receives with probability $1/2$ the value of k and with probability $1/2$ a random string $k' \leftarrow \{0,1\}^n$, then her probability of guessing which is the case would be at most $1/2 + negl(n)$ (where n can be thought of as $\log |G|$ or some other parameter related to the length of bit representation of members in the group).

13.3 Authenticated key exchange

The main issue with this key exchange protocol is of course that adversaries often are *not* passive. In particular, an active Eve could agree on her own key with Alice and Bob separately and then be able to see and modify all future communication. She might also be able to create weird (with some potential security implications) correlations by, say, modifying the message A to be A^2 etc..

For this reason, in actual applications we typically use *authenticated* key exchange. The notion of authentication used depends on what we can assume on the setup assumptions. A standard assumption is that Alice has some public keys but Bob doesn't. (This is the case when Alice is a website and Bob is a user.) However, one needs to take care in how to use this assumption. Indeed, the standard protocol for securing the web: the **transport Layer Security (TLS)** protocol (and its predecessor SSL) has gone through six revisions (including a name change from SSL to TLS) largely because of security concerns. We now illustrate one of those attacks.

13.3.1 Bleichenbacher's attack on RSA PKCS #1 V1.5 and SSL V3.0

If you have a public key, a natural approach is to take the encryption-based protocol and simply skip the first step since Bob already knows the public key e of Alice. This is basically what happened in the SSL V3.0 protocol. However, as was shown by Bleichenbacher in 1998, it turns out this is susceptible to the following attack:

- The adversary listens in on a conversation, and in particular observes $c = E_e(k)$ where k is the private key.

- The adversary then starts many connections with the server with ciphertexts related to c , and observes whether they succeed or fail (and in what way they fail, if they do). It turns out that based on this information, the adversary would be able to recover the key k .

Specifically, the version of RSA (known as PKCS # V1.5) used in the SSL V3.0 protocol requires the value x to have a particular format, with the top two bytes having a certain form. If in the course of the protocol, a server decrypts y and gets a value x not of this form then it would send an error message and halt the connection. While the designers of SSL V3.0 might not have thought of it that way, this amounts to saying that an SSL V3.0 server supplies to any party an oracle that on input y outputs 1 iff $y^d \pmod{m}$ has this form, where $d = e^{-1} \pmod{|Z_m^*|}$ is the secret decryption key. It turned out that one can use such an oracle to invert the RSA function. For a result of a similar flavor, see the (1/2 page) proof of Theorem 11.31 (page 418) in KL, where they show that an oracle that given y outputs the least significant bit of $y^d \pmod{m}$ allows to invert the RSA function.¹

For this reason, new versions of the SSL used a different variant of RSA known as PKCS #1 V2.0 which satisfies (under assumptions) *chosen ciphertext security* (CCA) and in particular such oracles cannot be used to break the encryption. (Nonetheless, there are still some implementation issues that allowed to perform some attacks, see the note in KL page 425 on Manfer's attack.)

13.4 Chosen ciphertext attack security for public key cryptography

The concept of chosen ciphertext attack security makes perfect sense for *public key* encryption as well. It is defined in the same way as it was in the private key setting:

Definition 13.1 — CCA secure public key encryption. A public key encryption scheme (G, E, D) is *chosen ciphertext attack (CCA) secure* if every efficient Mallory wins in the following game with probability at most $1/2 + negl(n)$:

- The keys (e, d) are generated via $G(1^n)$, and Mallory gets the public encryption key e and 1^n .
- For $poly(n)$ rounds, Mallory gets access to the function $c \mapsto D_d(c)$. (She doesn't need access to $m \mapsto E_e(m)$ since she already knows e .)

¹ The first attack of this flavor was given in the 1982 paper of Goldwasser, Micali, and Tong. Interestingly, this notion of "hardcore bits" has been used for both practical *attacks* against cryptosystems as well as theoretical (and sometimes practical) *constructions* of other cryptosystems.

- Mallory chooses a pair of messages $\{m_0, m_1\}$, a secret b is chosen at random in $\{0, 1\}$, and Mallory gets $c^* = E_e(m_b)$. (Note that she of course does *not* get the randomness used to generate this challenge encryption.)
- Mallory now gets another $\text{poly}(n)$ rounds of access to the function $c \mapsto D_d(c)$ except that she is not allowed to query c^* .
- Mallory outputs b' and *wins* if $b' = b$.

In the private key setting, we achieved CCA security by combining a CPA-secure private key encryption scheme with a message authenticating code (MAC), where to CCA-encrypt a message m , we first used the CPA-secure scheme on m to obtain a ciphertext c , and then added an authentication tag τ by signing c with the MAC. The decryption algorithm first verified the MAC before decrypting the ciphertext. In the public key setting, one might hope that we could repeat the same construction using a CPA-secure *public key* encryption and replacing the MAC with *digital signatures*.



Try to think what would be such a construction, and whether there is a fundamental obstacle to combining digital signatures and public key encryption in the same way we combined MACs and private key encryption.

Alas, as you may have realized, there is a fly in this ointment. In a signature scheme (necessarily) it is the *signing key* that is *secret*, and the *verification key* that is *public*. But in a public key encryption, the *encryption key* is *public*, and hence it makes no sense for it to use a secret signing key. (It's not hard to see that if you reveal the secret signing key then there is no point in using a signature scheme in the first place.)

Why CCA security matters. For the reasons above, constructing CCA secure public key encryption is very challenging. But is it worth the trouble? Do we really need this “ultra conservative” notion of security? The answer is *yes*. Just as we argued for *private key* encryption, chosen ciphertext security is the notion that gets us as close as possible to designing encryptions that fit the metaphor of *secure sealed envelopes*. Digital analogies will never be a perfect imitation of physical ones, but such metaphors are what people have in mind when designing cryptographic protocols, which is a hard enough task even when we don't have to worry about the ability of an adversary to reach inside a sealed envelope and XOR the contents

of the note written there with some arbitrary string. Indeed, several practical attacks, including Bleichenbacher's attack above, exploited exactly this gap between the physical metaphor and the digital realization. For more on this, please see [Victor Shoup's survey](#) where he also describes the Cramer-Shoup encryption scheme which was the first practical public key system to be shown CCA secure without resorting to the random oracle heuristic. (The first definition of CCA security, as well as the first polynomial-time construction, was given in a seminal 1991 work of Dolev, Dwork and Naor.)

13.5 CCA secure public key encryption in the Random Oracle Model

We now show how to convert any CPA-secure public key encryption scheme to a CCA-secure scheme in the random oracle model (this construction is taken from Fujisaki and Okamoto, CRYPTO 99). In the homework, you will see a somewhat simpler direct construction of a CCA secure scheme from a *trapdoor permutation*, a variant of which is known as OAEP (which has better ciphertext expansion) has been standardized as PKCS #1 V2.0 and is used in several protocols. The advantage of a generic construction is that it can be instantiated not just with the RSA and Rabin schemes, but also directly with Diffie-Hellman and Lattice based schemes (though there are direct and more efficient variants for these as well).

CCA-ROM-ENC Scheme:

- **Ingredients:** A public key encryption scheme (G', E', D') and a two hash functions $H, H' : \{0,1\}^* \rightarrow \{0,1\}^n$ (which we model as independent random oracles²)
- **Key generation:** We generate keys $(e, d) = G'(1^n)$ for the underlying encryption scheme.
- **Encryption:** To encrypt a message $m \in \{0,1\}^\ell$, we select randomness $r \leftarrow_R \{0,1\}^\ell$ for the underlying encryption algorithm E' and output $E'_e(r; H(m||r))||(r \oplus m)||H'(m||r)$, where by $E'_e(m'; r')$ we denote the result of encrypting m' using the key e and the randomness r' (we assume the scheme takes n bits of randomness as input; otherwise modify the output length of H accordingly).
- **Decryption:** To decrypt a ciphertext $c||y||z$ first let $r = D_d(c)$, $m = r \oplus y$ and then check that $c = E_e(m; H(m||r))$ and $z = H'(m||r)$. If any of the checks fail we output **error**; otherwise we

output m .

The above CCA-ROM-ENC scheme is CCA secure.

Proof of ??. Suppose towards a contradiction that there exists an adversary M that wins the CCA game with probability at least $1/2 + \epsilon$ where ϵ is non-negligible. Our aim is to show that the decryption box would be “useless” to M and hence reduce CCA security to CPA security (which we’ll then derive from the CPA security of the underlying scheme).

Consider the following “box” \hat{D} that will answer decryption queries $c\|y\|z$ of the adversary as follows:

- * If z was returned before to the adversary as an answer to $H'(m\|r)$ for some m, r , and $c = E_e(m\|H(m\|r))$ and $y = m \oplus r$ then return m .

- * Otherwise return error

Claim: The probability that \hat{D} answers a query differently than D is negligible.

Proof of claim: If D gives a non error response to a query $c\|y\|z$ then it must be that $z = H'(m\|r)$ for some m, r such that $y = r \oplus m$ and $c = E_e(r; H(m\|r))$, in which case D will return m . The only way that \hat{D} will answer this question differently is if $z = H'(m\|r)$ but the query $m\|r$ hasn’t been asked before by the adversary. Here there are two options. If this query has never been asked before at all, then by the lazy evaluation principle in this case we can think of $H'(m\|r)$ as being independently chosen at this point, and the probability it happens to equal z will be 2^{-n} . If this query was asked by someone apart from the adversary then it could only have been asked by the encryption oracle while producing the challenge ciphertext $c^*\|y^*\|z^*$, but since the adversary is not allowed to ask this precise ciphertext, then it must be a ciphertext of the form $c\|y\|z^*$ where $(c, y) \neq (c^*, y^*)$ and such a ciphertext would get an error response from both oracles.

QED (claim)

Note that we can assume without loss of generality that if m^* is the challenge message and r^* is the randomness chosen in this challenge, the adversary never asks the query $m^*\|r^*$ to the its H or H' oracles, since we can modify it so that before making a query $m\|r$, it will first check if $E_e(m\|r) = c^*$ where $c^*\|y^*\|z^*$ is the challenge ciphertext, and if so use this to win the game.

² Recall that it’s easy to obtain two independent random oracles H, H' from a single oracle H'' , for example by letting $H(x) = H''(0\|x)$ and $H'(x) = H''(1\|x)$.

In other words, if we modified the experiment so the values $R^* = H(r^* \| m)$ and $z^* = H'(m^* \| r^*)$ chosen while producing the challenge are simply random strings chosen completely independently of everything else. Now note that our oracle \hat{D} did *not* need to use the decryption key d . So, if the adversary wins the CCA game, then it wins the *CPA game* for the encryption scheme $E_e(m) = E'_e(r; R) \| r \oplus m \| R'$ where R and R' are simply independent random strings; we leave proving that this scheme is CPA secure as an exercise to the reader. ■

13.5.1 Defining secure authenticated key exchange

The basic goal of secure communication is to set up a *secure channel* between two parties Alice and Bob. We want to do so over an open network, where messages between Alice and Bob might be read, modified, deleted, or added by the adversary. Moreover, we want Alice and Bob to be sure that they are talking to one another rather than other parties. This raises the question of what is identity and how is it verified. Ultimately, if we want to use identities, then we need to trust some authority that decides which party has which identity. This is typically done via a *certificate authority (CA)*. This is some trusted authority, whose verification key v_{CA} is public and known to all parties. Alice proves in some way to the CA that she is indeed Alice, and then generates a pair (s_{Alice}, v_{Alice}) , and gets from the CA the message $\sigma_{Alice} = \text{"The key } v_{Alice} \text{ belongs to Alice"}$ signed with s_{CA} .³ Now Alice can send $(v_{Alice}, \sigma_{Alice})$ to Bob to certify that the owner of this public key is indeed Alice.

For example, in the web setting, certain *certificate authorities* can certify that a certain public key is associated with a certain website. If you go to a website using the `https` protocol, you should see a “lock” symbol on your browser which will give you details on the certificate. Often the certificate is a chain of certificate. If I click on this lock symbol in my Chrome browser, I see that the certificate that amazon.com’s public key is some particular string (corresponding to a 2048 RSA modulus and exponent) is signed by the Symantec Certificate authority, whose own key is certified by Verisign. My communication with Amazon is an example of a setting of *one sided authentication*. It is important for me to know that I am truly talking to amazon.com, while Amazon is willing to talk to any client. (Though of course once we establish a secure channel, I could use it to login to my Amazon account.) Chapter 21 of Boneh Shoup contains an in depth discussion of authenticated key exchange protocols.

³ The registration process could be more subtle than that, and for example Alice might need to *prove* to the CA that she does indeed know the corresponding secret key.

P You should stop here and read Section 21.9 of Boneh Shoup with the formal definitions of authenticated key exchange, going back as needed to the previous section for the definitions of protocols AEK1 - AEK4.

13.5.2 The compiler approach for authenticated key exchange

There is a generic “compiler” approach to obtaining authenticated key exchange protocols:

- Start with a protocol such as the basic Diffie-Hellman protocol that is only secure with respect to a *passive eavesdropping* adversary.
- Then *compile* it into a protocol that is secure with respect to an active adversary using authentication tools such as digital signatures, message authentication codes, etc., depending on what kind of setup you can assume and what properties you want to achieve.

This approach has the advantage of being modular in both the construction and the analysis. However, direct constructions might be more efficient. There are a great many potentially desirable properties of key exchange protocols, and different protocols achieve different subsets of these properties at different costs. The most common variant of authenticated key exchange protocols is to use some version of the Diffie-Hellman key exchange. If both parties have public signature keys, then they can simply sign their messages and then that effectively rules out an active attack, reducing active security to passive security (though one needs to include identities in the signatures to ensure non repeating of messages, see [here](#)).

The most efficient variants of Diffie Hellman achieve authentication implicitly, where the basic protocol remains the same (sending $X = g^x$ and $Y = g^y$) but the computation of the secret shared key involves some authentication information. Of these protocols a particularly efficient variant is the MQV protocol of Law, Menezes, Qu, Solinas and Vanstone (which is based on similar principles as DSA signatures), and its variant **HMQV** by Krawczyk that has some improved security properties and analysis

13.6 Password authenticated key exchange.

To be completed (the most natural candidate: use MACS with a password-derived key to authenticate communication - completely fails)



Please skim Boneh Shoup Chapter 21.11

13.7 Client to client key exchange for secure text messaging - ZRTP, OTR, TextSecure

To be completed. See [Matthew Green's blog](#), [text secure](#), [OTR](#).

Security requirements: forward secrecy, deniability.

13.8 Heartbleed and logjam attacks

- Vestiges of past crypto policies.
- Importance of “perfect forward secrecy”

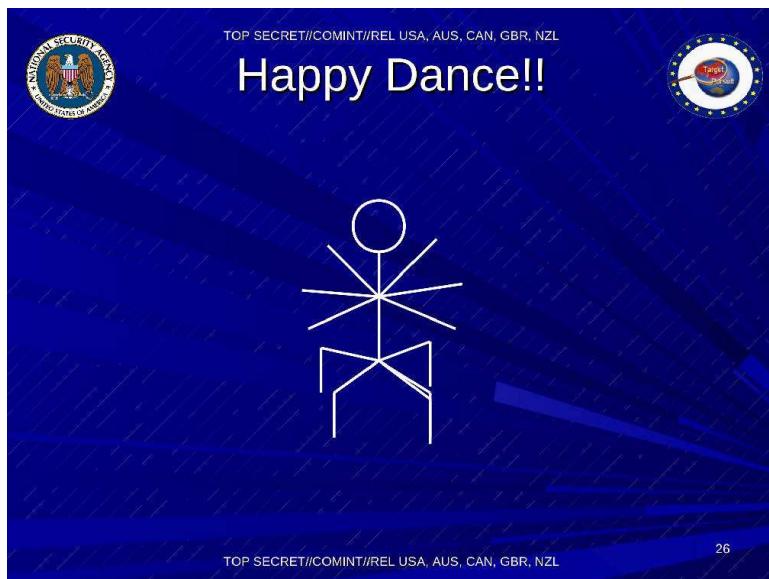


Figure 13.1: How the NSA feels about breaking encrypted communication

14

Zero knowledge proofs

The notion of *proof* is central to so many fields. In mathematics, we want to prove that a certain assertion is correct. In other sciences, we often want to accumulate a preponderance of evidence (or statistical significance) to reject certain hypothesis. In criminal law the prosecution famously needs to prove its case “beyond a reasonable doubt”. Cryptography turns out to give some new twists on this ancient notion.

Typically a proof that some assertion X is true, also reveals some information about *why* X is true. When Hercule Poirot proves that Norman Gale killed Madame Giselle he does so by showing *how* Gale committed the murder by dressing up as a flight attendant and stabbing Madame Gisselle with a poisoned dart. Could Hercule convince us beyond a reasonable doubt that Gale did the crime without giving any information on *how* the crime was committed? Can the Russians prove to the U.S. that a sealed box contains an authentic nuclear warhead without revealing anything about its design? Can I prove to you that the number $m = 385,608,108,395,369,363,400,501,273,594,475,104,405,448,848,047,062,278,473,983$ has a prime factor whose last digit is 7 without giving you any information about m 's prime factors? We won't answer the first question, but will show some insights on the latter two.¹

Zero knowledge proofs are proofs that fully convince that a statement is true without yielding *any additional knowledge*. So, after seeing a zero knowledge proof that m has a factor ending with 7, you'll be no closer to knowing m 's factorization than you were before. Zero knowledge proofs were invented by Goldwasser, Micali and Rackoff in 1982 and have since been used in great many settings. How would you achieve such a thing, or even define it? And why on earth would it be useful? This is the topic of this lecture.

¹ In case you are curious, the factors of m are 1, 172, 192, 558, 529, 627, 184, 841, 954, 822, 099 and 328, 963, 108, 995, 562, 790, 517, 498, 071, 717.

14.1 Applications for zero knowledge proofs.

Before we talk about how to achieve zero knowledge, let us discuss some of its potential applications:

14.1.1 Nuclear disarmament

The United States and Russia have reached a dangerous and expensive equilibrium by which each has about [7000 nuclear warheads](#), much more than is needed to decimate each others' population (and the population of much of the rest of the world).² Having so many weapons increases the chance of "leakage" of weapons, or of an accidental launch (which can result in an all out war) through fault in communications or rogue commanders. This also threatens the delicate balance of the [Non-Proliferation Treaty](#) which at its core is a bargain where non-weapons states agree not to pursue nuclear weapons and the five nuclear weapon states agree to make progress on nuclear disarmament. These huge quantities of nuclear weapons are not only dangerous, as they increase the chance of a leak or of an individual failure or rogue commander causing a world catastrophe, but also extremely expensive to maintain.

For all of these reasons, in 2009, U.S. President Obama called to set as a long term goal a "world without nuclear weapons" and in 2012 talked about concretely talking to Russia about reducing "not only our strategic nuclear warheads, but also tactical weapons and warheads in reserve". On the other side, Russian President Putin has said already in 2000 that he sees "no obstacles that could hamper future deep cuts of strategic offensive armaments". (Though as of 2018, political winds on both sides have shifted away from disarmament and more toward armament.)

There are many reasons why progress on nuclear disarmament has been so slow, and most of them have nothing to do with zero knowledge or any other piece of technology. But there are some technical hurdles as well. One of those hurdles is that for the U.S. and Russia to go beyond restricting the number of *deployed* weapons to significantly reducing the *stockpiles*, they need to find a way for one country to verifiably prove that it has dismantled warheads. As mentioned in my [work with Glaser and Goldston](#) (see also [this page](#)), a key stumbling block is that the design of a nuclear warhead is of course highly classified and about the last thing in the world that the U.S. would like to share with Russia and vice versa. So, how can the

² To be fair, "only" about 170 million Americans live in the [50 largest metropolitan areas](#) and so arguably many people will survive at least the initial impact of a nuclear war, though it had been estimated that even a "small" nuclear war involving detonation of 100 not too large warheads could have [devastating global consequences](#).

U.S. convince the Russian that it has destroyed a warhead, when it cannot let Russian experts anywhere near it?

14.1.2 Voting

Electronic voting has been of great interest for many reasons. One potential advantage is that it could allow completely transparent vote counting, where every citizen could verify that the votes were counted correctly. For example, Chaum suggested an approach to do so by publishing an encryption of every vote and then having the central authority *prove* that the final outcome corresponds to the counts of all the plaintexts. But of course to maintain voter privacy, we need to prove this without actually revealing those plaintexts. Can we do so?

14.1.3 More applications

I chose these two examples above precisely because they are hardly the first that come to mind when thinking about zero knowledge. Zero knowledge has been used for many cryptographic applications. One such application (originating from work of Fiat and Shamir) is the use for *identification protocols*. Here Alice knows a solution x to a puzzle P , and proves her identity to Bob by, for example, providing an encryption c of x and proving in zero knowledge that c is indeed an encryption of a solution for P .³ Bob can verify the proof, but because it is zero knowledge, learns nothing about the solution of the puzzle and will not be able to impersonate Alice. An alternative approach to such identification protocols is through using *digital signatures*; this connection goes both ways and zero knowledge proofs have been used by Schnorr and others as a basis for signature schemes.

Another very generic application is for “compiling protocols”. As we’ve seen time and again, it is often much easier to handle *passive* adversaries than *active* ones. (For example, it’s much easier to get CPA security against the eavesdropping Eve than CCA security against the person-in-the-middle Mallory.) Thus it would be wonderful if we could “compile” a protocol that is secure with respect to passive attacks into one that is secure with respect to active ones. As was first shown by Goldreich, Micali, and Wigderson, zero knowledge proofs yield a very general such compiler. The idea is that all parties prove in zero knowledge that they follow the protocol’s specifications. Normally, such proofs might require the parties to reveal

³ As we’ll see, technically what Alice needs to do in such a scenario is use a *zero knowledge proof of knowledge* of a solution for P .

their secret inputs, hence violating security, but zero knowledge precisely guarantees that we can verify correct behaviour without access to these inputs.

14.2 Defining and constructing zero knowledge proofs

So, zero knowledge proofs are wonderful objects, but how do we get them? In fact, we haven't answered the even more basic question of how do we *define* zero knowledge? We have to start by the most basic task of defining what we mean by a *proof*.

A *proof system* can be thought of as an algorithm V (for "verifier") that takes as input a *statement* which is some string x and another string π known as the *proof* and outputs 1 if and only if π is a valid proof that the statement x is correct. For example:

- In *Euclidean geometry*, *statements* are geometric facts such as "in any triangle the degrees sum to 180 degrees" and the *proofs* are step by step derivations of the statements from the five basic *postulates*.
- In *Zermelo-Fraenkel + Axiom of Choice (ZFC)* a *statement* is some purported fact about sets (e.g., the Riemann Hypothesis⁴), and a *proof* is a step by step derivation of it from the axioms.
- We can many define other "theories". For example, a theory where the statements are pairs (x, m) such that x is a quadratic residue modulo m and a proof for x is the number s such that $x = s^2 \pmod{m}$, or a theory where the theorems are *Hamiltonian graphs* G (graphs on n vertices that contain an n -long cycle) and the proofs are the description of the cycle.

All these proof systems have the property that the verifying algorithm V is *efficient*. Indeed, that's the whole point of a proof π - it's a sequence of symbols that makes it easy to verify that the statement is true.

To achieve the notion of zero knowledge proofs, Goldwasser and Micali had to consider a generalization of proofs from static sequences of symbols to *interactive probabilistic protocols* between a prover and a verifier. Let's start with an informal example. The vast majority of humans have three types of cone cells in their eyes. This is the reason why *we perceive the sky as blue* (see also [this](#)), despite its color being quite a different spectrum than the blue of the rainbow, is that the projection of the sky's color to our cones is closest to the projection of blue. It has been suggested that a tiny fraction of the human population might have four functioning cones (in fact,

⁴ Integers can be coded as sets in various ways. For example, one can encode 0 as \emptyset and if N is the set encoding n , we can encode $n+1$ using the $n+1$ -element set $\{N\} \cup N$.

only women, as it would require two X chromosomes and a certain mutation). How would a person *prove* to another that she is in fact such a **tetrachromat**?

Proof of tetrachromacy:

Suppose that Alice is a tetrachromat and can distinguish between the colors of two pieces of plastic that would be identical to a trichromat. She wants to prove to a trichromat Bob that the two pieces are not identical. She can do this as follows:

Alice and Bob will repeat the following experiment n times: Alice turns her back and Bob tosses a coin and with probability $1/2$ leaves the pieces as they are, and with probability $1/2$ switches the right piece with the left piece. Alice needs to guess whether Bob switched the pieces or not.

If Alice is successful in all of the n repetitions then Bob will have $1 - 2^{-n}$ confidence that the pieces are truly different.

We now consider a more “mathematical” example along similar lines. Recall that if x and m are numbers then we say that x is a *quadratic residue* modulo m if there is some s such that $x = s^2 \pmod{m}$. Let us define the function $NQR(m, x)$ to output 1 if and only if $x \neq s^2 \pmod{m}$ for every $s \in \{0, \dots, m-1\}$. There is a very simple way to prove statements of the form “ $NQR(m, x) = 0$ ”: just give out s . However, here is an interactive proof system to prove statements of the form “ $NQR(m, x) = 1$ ”:

- We have two parties: **Alice** and **Bob**. The **common input** is (m, x) and Alice wants to convince Bob that $NQR(m, x) = 1$. (That is, that x is *not* a quadratic residue modulo m).
- We assume that Alice can compute $NQR(m, w)$ for every $w \in \{0, \dots, m-1\}$ but Bob is polynomial time.
- The protocol will work as follows:
 1. Bob will pick some random $s \in \mathbb{Z}_m^*$ (e.g., by picking a random number in $\{1, \dots, m-1\}$ and discard it if it has nontrivial g.c.d. with m) and toss a coin $b \in \{0, 1\}$. If $b = 0$ then Bob will send $s^2 \pmod{m}$ to Alice and otherwise he will send $xs^2 \pmod{m}$ to Alice.
 2. Alice will use her ability to compute $NQR(m, \cdot)$ to respond with $b' = 0$ if Bob sent a quadratic residue and with $b' = 1$ otherwise.
 3. Bob *accepts* the proof if $b = b'$.

To see that Bob will indeed accept the proof, note that if x is a non-residue then xs^2 will have to be a non-residue as well (since if it had a root s' then $s's^{-1}$ would be a root of x). Hence it will always be the case that $b' = b$.

Moreover, if x was a quadratic residue of the form $x = s'^2 \pmod{m}$ for some s' , then $xs^2 = (s's)^2$ is simply a random quadratic residue, which means that in this case Bob's message is distributed the same regardless of whether $b = 0$ or $b = 1$, and no matter what she does, Alice has probability at most $1/2$ of guessing b . Hence if Alice is always successful than after n repetitions Bob would have $1 - 2^{-n}$ confidence that x is indeed a non-residue modulo m .



Please stop and make sure you see the similarities between this protocol and the one for demonstrating that the two pieces of plastic do not have identical colors.

Let us now make the formal definition:

Definition 14.1 — Proof systems. Let $f : \{0,1\}^* \rightarrow \{0,1\}$ be some function. A *probabilistic proof for f* (i.e., a proof for statements of the form " $f(x) = 1$ ") is a pair of interactive algorithms (P, V) such that V runs in polynomial time and they satisfy:

- **Completeness:** If $f(x) = 1$ then on input x , if P and V are given input x and interact, then at the end of the interaction V will output `Accept` with probability at least 0.9.
- **Soundness:** If $f(x) = 0$ then for any arbitrary (efficient or non efficient) algorithm P^* , if P^* and V are given input x and interact then at the end V will output `Accept` with probability at most 0.1



Functions vs languages In many texts proof systems are defined with respect to *languages* as opposed to *functions*. That is, instead of talking about a function $f : \{0,1\}^* \rightarrow \{0,1\}$ we talk about a language $L \subseteq \{0,1\}^*$. These two viewpoints are completely equivalent via the mapping $f \longleftrightarrow L$ where $L = \{x \mid f(x) = 1\}$.

Note that we don't necessarily require the prover to be efficient (and indeed, in some cases it might not be). On the other hand, our soundness condition holds even if the prover uses a non efficient

strategy.⁵ We say that a proof system has an *efficient prover* if there is an NP-type proof system Π for L (that is some efficient algorithm Π such that there exists π with $\Pi(x, \pi) = 1$ iff $x \in L$ and such that $\Pi(x, \pi) = 1$ implies that $|\pi| \leq \text{poly}(|x|)$), such that the strategy for P can be implemented efficiently given any static proof π for x in this system.

R

Notation for strategies Up until now, we always considered cryptographic protocols where Alice and Bob trusted one another, but were worried about some adversary controlling the channel between them. Now we are in a somewhat more “suspicious” setting where the parties do not fully trust one another. In such protocols there is always a “prescribed” or **honest** strategy that a particular party *should* follow, but we generally don’t want the other parties’ security to rely on someone else’s good intention, and hence analyze also the case where a party uses an arbitrary **malicious** strategy. We sometimes also consider the **honest but curious** case where the adversary is passive and only collects information, but does not deviate from the prescribed strategy.

Protocols typically only guarantee security for party A when it behaves honestly - a party can always choose to violate its own security and there is not much we can (or should?) do about it.

⁵ People have considered the notion of zero knowledge systems where soundness holds only with respect to efficient provers; these are known as *argument systems*.

14.3 Defining zero knowledge

So far we merely defined the notion of an interactive proof system, but we need to define what it means for a proof to be *zero knowledge*. Before we attempt a definition, let us consider an example. Going back to the notion of quadratic residuosity, suppose that x and m are public and Alice knows s such that $x = s^2 \pmod{m}$. She wants to convince Bob that this is the case. However she prefers not to reveal s . Can she convince Bob that such an s exist without revealing any information about it? Here is a way to do so:

Protocol ZK-QR: Public input for Alice and Bob: x, m ; Alice’s private input is s such that $x = s^2 \pmod{m}$.

1. Alice will pick a random s' and send to Bob $x' = xs'^2 \pmod{m}$.
2. Bob will pick a random bit $b \in \{0, 1\}$ and send b to Alice.
3. If $b = 0$ then Alice reveals ss' , hence giving out a root for x' ; if

$b = 1$ then Alice reveals s' , hence showing a root for $x'x^{-1}$.

4. Bob checks that the value s'' revealed by Alice is indeed a root of $x'x^{-b}$, if so then it “accepts” the proof.

If x was *not* a quadratic residue then no matter how x' was chosen, either x' or $x'x^{-1}$ is *not* a residue and hence Bob will reject the proof with probability at least $1/2$. By repeating this n times, we can reduce the probability of Bob accepting the proof of a non residue to 2^{-n} .

On the other hand, we claim that we didn’t really reveal anything about s . Indeed, if Bob chooses $b = 0$, then the two messages (x', ss') he sees can be thought of as a random quadratic residue x' and its root. If Bob chooses $b = 1$ then after dividing by x (which he could have done by himself) he still gets a random residue x'' and its root s' . In both cases, the distribution of these two messages is completely independent of s , and hence intuitively yields no additional information about it beyond whatever Bob knew before.

To define zero knowledge mathematically we follow the following intuition:

A proof system is zero knowledge if the verifier did not learn anything after the interaction that he could not have learned on his own.

Here is how we formally define this:

Definition 14.2 — Zero knowledge proofs. A proof system (P, V) for f is *zero knowledge* if for every efficient verifier strategy V^* there exists an efficient probabilistic algorithm S^* (known as the *simulator*) such that for every x s.t. $f(x) = 1$, the following random variables are computationally indistinguishable:

- The output of V^* after interacting with P on input x .
- The output of S^* on input x .

That is, we can show the verifier does not gain anything from the interaction, because no matter what algorithm V^* he uses, whatever he learned as a result of interacting with the prover, he could have just as equally learned by simply running the standalone algorithm S^* on the same input.

R **The simulation paradigm** The natural way to define security is to say that a system is secure if some “laundry list” of bad outcomes X,Y,Z can’t happen. The definition of zero knowledge is different. Rather than giving a list of the events that are *not allowed* to occur, it gives a maximalist *simulation* condition.

At its heart the definition of zero knowledge says the following: clearly, we cannot prevent the verifier from running an efficient algorithm S^* on the public input, but we want to ensure that this is the most he can learn from the interaction. This *simulation paradigm* has become the standard way to define security of a great many cryptographic applications. That is, we bound what an adversary Eve can learn by postulating some hypothetical adversary Lilith that is under much harsher conditions (e.g., does not get to interact with the prover) and ensuring that Eve cannot learn anything that Lilith couldn’t have learned either. This has an advantage of being the most conservative definition possible, and also phrasing security in *positive* terms- there exists a simulation - as opposed to the typical *negative* terms - events X,Y,Z can’t happen. Since it’s often easier for us to think of positive terms, paradoxically sometimes this stronger security condition is easier to prove. Zero knowledge is in some sense the simplest setting of the simulation paradigm and we’ll see it time and again in dealing with more advanced notions.

The definition of zero knowledge is confusing since intuitively one thing that if the verifier gained confidence that the statement is true than surely he must have learned *something*. This is another one of those cases where cryptography is counterintuitive. To understand it better, it is worthwhile to see the formal proof that the protocol above for quadratic residuosity is zero knowledge:

Theorem 14.3 — Zero knowledge for quadratic residuosity. Protocol ZK-QR above is a zero knowledge protocol.

Proof. Let V^* be an arbitrary efficient strategy for Bob. Since Bob only sends a single bit, we can think of this strategy as composed of two functions:

- $V_1(x, m, x')$ outputs the bit b that Bob chooses on input x, m and after Alice’s first message is x' .
- $V_2(x, m, x', s'')$ is whatever Bob outputs after seeing Alice’s response s'' to the bit b .

Both V_1 and V_2 are efficiently computable. We now need to come up with an efficient simulator S^* that is a standalone algorithm that on input x, m will output a distribution indistinguishable from the output V^* . The simulator S^* will work as follows:

1. Pick $b' \leftarrow_R \{0, 1\}$.
2. Pick s'' at random in \mathbb{Z}_m^* . If $b = 0$ then let $x' = s''^2 \pmod{m}$. Otherwise output $x' = xs''^2 \pmod{m}$.
3. Let $b = V_1(x, m, x')$. If $b \neq b'$ then go back to step 1.
4. Output $V_2(x, m, x', s'')$.

The correctness of the simulator follows from the following claims (all of which assume that x is actually a quadratic residue, since otherwise we don't need to make any guarantees and in any case Alice's behaviour is not well defined):

Claim 1: The distribution of x' computed by S^* is identical to the distribution of x' chosen by Alice.

Claim 2: With probability at least $1/2$, $b' = b$.

Claim 3: Conditioned on $b = b'$ and the value x' computed in step 2, the value s'' computed by S^* is identical to the value that Alice sends when her first message is X' and Bob's response is b .

Together these three claims imply that in expectation S^* only invokes V_1 and V_2 a constant number of times (since every time it goes back to step 1 with probability at most $1/2$). They also imply that the output of S^* is in fact identical to the output of V^* in a true interaction with Alice. Thus, we only need to prove the claims, which is actually quite easy:

Proof of Claim 1: In both cases, x' is a random quadratic residue. QED

Proof of Claim 2: This is a corollary of Claim 1; since the distribution of x' is identical to the distribution chosen by Alice, in particular x' gives out no information about the choice of b' . QED

Proof of Claim 3: This follows from a direct calculation. The value s'' sent by Alice is a square root of x' if $b = 0$ and of $x'x^{-1}$ if $x = 1$. But this is identical to what happens for S^* if $b = b'$. QED

Together these complete the proof of the theorem. ■

Theorem 14.3 is interesting but not yet good enough to guarantee security in practice. After all, the protocol that we really need to

show is zero knowledge is the one where we repeat this procedure n times. This is a general theorem that if a protocol is zero knowledge then repeating it polynomially many times one after the other (so called “sequential repetition”) preserves zero knowledge. You can think of this as cryptography’s version of the equality “ $0 + 0 = 0$ ”, but as usual, intuitive things are not always correct and so this theorem does require (a not super trivial) proof. It is a good exercise to try to prove it on your own. There are known ways to achieve zero knowledge with negligible soundness error and a *constant* number of communication rounds, see Goldreich’s book (Vol 1, Sec 4.9).

14.4 Zero knowledge proof for Hamiltonicity.

We now show a proof for another language. Suppose that Alice and Bob know an n -vertex graph G and Alice knows a *Hamiltonian cycle* C in this graph (i.e.. a length n simple cycle- one that traverses all vertices exactly once). Here is how Alice can prove that such a cycle exists without revealing any information about it:

Protocol ZK-Ham:

1. **Common input:** graph H (in the form of an $n \times n$ adjacency matrix); **Alice’s private input:** a Hamiltonian cycle $C = (C_1, \dots, C_n)$ which are distinct vertices such that $(C_\ell, C_{\ell+1})$ is an edge in H for all $\ell \in \{1, \dots, n-1\}$ and (C_n, C_1) is an edge as well.
2. Bob chooses a random string $z \in \{0, 1\}^{3n}$
3. Alice chooses a random permutation π on $\{1, \dots, n\}$ and let M be the π -permuted adjacency matrix of H (i.e., $M_{\pi(i), \pi(j)} = 1$ iff (i, j) is an edge in H). For every i, j , Alice chooses a random string $x_{i,j} \in \{0, 1\}^n$ and let $y_{i,j} = G(x_{i,j}) \oplus M_{i,j}z$, where $G : \{0, 1\}^n \rightarrow \{0, 1\}^{3n}$ is a pseudorandom generator. She sends $\{y_{i,j}\}_{i,j \in [n]}$ to Bob.
4. Bob chooses a bit $b \in \{0, 1\}$.
5. If $b = 0$ then Alice sends out π and the strings $\{x_{i,j}\}$ for all i, j ; If $b = 1$ then Alice sends out the n strings $x_{\pi(C_1), \pi(C_2)}, \dots, x_{\pi(C_n), \pi(C_1)}$ together with their indices.
6. If $b = 0$ then Bob computes M to be the π -permuted adjacency matrix of H and verifies that all the $y_{i,j}$ ’s were computed from the $x_{i,j}$ ’s appropriately. If $b = 1$ then verify that the indices of the strings $\{x_{i,j}\}$ sent by Alice form a cycle and that indeed $y_{i,j} = G(x_{i,j}) \oplus z$ for every string $x_{i,j}$ that was sent by Alice.

Theorem 14.4 — Zero Knowledge proof for Hamiltonian Cycle. Protocol ZK-Ham is a zero knowledge proof system for the language of Hamiltonian graphs.⁶

Proof. We need to prove **completeness**, **soundness**, and **zero knowledge**.

Completeness can be easily verified, and so we leave this to the reader.

For **soundness**, we recall that (as we've seen before) with extremely high probability the sets $S_0 = \{G(x) : x \in \{0,1\}^n\}$ and $S_1 = \{G(x) \oplus z : x \in \{0,1\}^n\}$ will be disjoint (this probability is over the choice of z that is done by the verifier). Now, assuming this is the case, given the messages $\{y_{i,j}\}$ sent by the prover in the first step, define an $n \times n$ matrix M' with entries in $\{0,1,?\}$ as follows: $M'_{i,j} = 0$ if $y_{i,j} \in S_0$, $M'_{i,j} = 1$ if $y_{i,j} \in S_1$ and $M'_{i,j} = ?$ otherwise.

We split into two cases. The first case is that there exists some permutation π such that **(i)** M' is a π -permuted version of the input graph G and **(ii)** M' contains a Hamiltonian cycle. Clearly in this case G contains a Hamiltonian cycle as well, and hence we don't need to consider it when analyzing soundness. In the other case we claim that with probability at least $1/2$ the verifier will reject the proof. Indeed, if **(i)** is violated then the proof will be rejected if Bob chooses $b = 0$ and if **(ii)** is violated then the proof will be rejected if Bob chooses $b = 1$.

We now turn to showing **zero knowledge**. For this we need to build a *simulator* S^* for an arbitrary efficient strategy V^* of Bob. Recall that S^* gets as input the graph H (but not the *Hamiltonian* cycle C) and needs to produce an output that is indistinguishable from the output of V^* . It will do so as follows:

1. Pick $b' \in \{0,1\}$.
2. Let $z \in \{0,1\}^{3n}$ be the first message computed by V^* on input H .
3. If $b' = 0$ then S^* computes the second message as Alice does: chooses a random permutation π on $\{1, \dots, n\}$ and let M be the π -permuted adjacency matrix of H (i.e., $M_{\pi(i),\pi(j)} = 1$ iff (i,j) is an edge in H). In contrast, if $b' = 1$ then S^* lets M be the all 1' matrix. For every i,j , S^* chooses a random string $x_{i,j} \in \{0,1\}^n$ and let $y_{i,j} = G(x_{i,j}) \oplus M_{i,j}z$, where $G : \{0,1\}^n \rightarrow \{0,1\}^{3n}$ is a pseudorandom generator.

⁶ Goldreich, Micali and Wigderson were the first to come up with a zero knowledge proof for an NP complete problem, though the Hamiltonicity protocol here is from a later work by Blum. We use Naor's commitment scheme.

4. Let b be the output of V^* when given the input H and the first message $\{y_{i,j}\}$ computed as above. If $b \neq b'$ then go back to step 0.
5. We compute the fourth message of the protocol similarly to how Alice does it: if $b = 0$ then it consists of π and the strings $\{x_{i,j}\}$ for all i, j ; If $b = 1$ then we pick a random length- n cycle C' and the message consists of the n strings $x_{C'_1,C'_2}, \dots, x_{C'_n,C'_1}$ together with their indices.
6. Output whatever V^* outputs when given the prior message.

We prove the output of the simulator is indistinguishable from the output of V^* in an actual interaction by the following claims:

Claim 1: The message $\{y_{i,j}\}$ computed by S^* is computationally indistinguishable from the first message computed by Alice.

Claim 2: The probability that $b = b'$ is at least $1/3$.

Claim 3: The fourth message computed by S^* is computationally indistinguishable from the fourth message computed by Alice.

We will simply sketch here the proofs (again see Goldreich's book for full proofs):

For Claim 1, note that if $b' = 0$ then the message is *identical* to the way Alice computes it. If $b' = 1$ then the difference is that S^* computes some strings $y_{i,j}$ of the form $G(x_{i,j}) + z$ where Alice would compute the corresponding strings as $G(x_{i,j})$ this is indistinguishable because G is a pseudorandom generator (and the distribution $U_{3n} \oplus z$ is the same as U_{3n}).

Claim 2 is a corollary of Claim 1. If V^* managed to pick a message b such that $\mathbb{P}[b = b'] < 1/2 - negl(n)$ then in particular it could distinguish between the first message of Alice (that is computed independently of b' and hence contains no information about it) from the first message of V^* .

For Claim 3, note that again if $b = 0$ then the message is computed in a way identical to what Alice does. If $b = 1$ then this message is also computed in a way identical to Alice, since it does not matter if instead of picking C' at random, we picked a random permutation π and let C' be the image of the Hamiltonian cycle under this permutation.

This completes the proof of the theorem. ■

14.4.1 Why is this interesting?

The reason that a protocol for Hamiltonicity is more interesting than a protocol for Quadratic residuosity is that Hamiltonicity is an NP-complete question. This means that for every other NP language L , we can use the reduction from L to Hamiltonicity combined with protocol ZK-Ham to give a zero knowledge proof system for L . In particular this means that we can have zero knowledge proofs for the following languages:

- The language of numbers m such that there exists a prime p dividing m whose remainder modulo 10 is 7.
- The language of tuples X, e, c_1, \dots, c_n such that c_i is an encryption of a number x_i with $\sum x_i = X$. (This is essentially what we needed in the voting example above).
- For every efficient function F , the language of pairs x, y such that there exists some input r satisfying $y = F(x||r)$. (This is what we often need in the “protocol compiling” applications to show that a particular output was produced by the correct program F on public input x and private input r .)

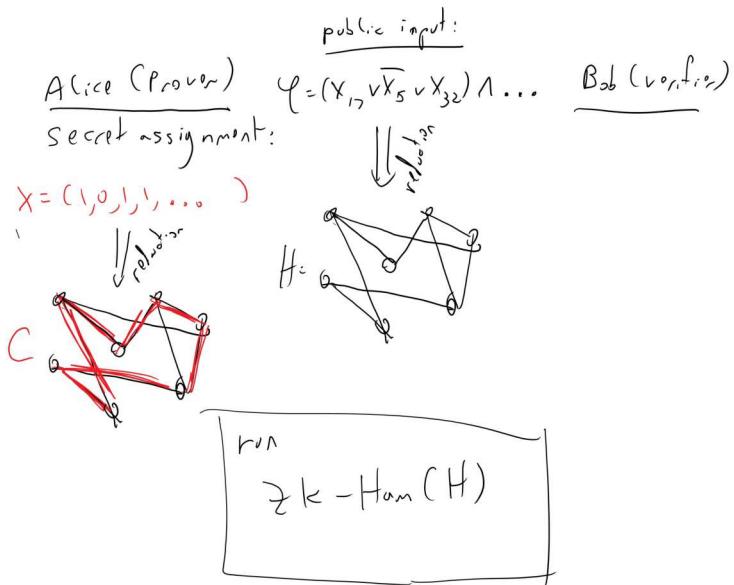


Figure 14.1: Using a zero knowledge protocol for Hamiltonicity we can obtain a zero knowledge protocol for any language L in NP. For example, if the public input is a SAT formula φ and the Prover’s secret input is a satisfying assignment x for φ then the verifier can run the reduction on φ to obtain a graph H and the prover can run the same reduction to obtain from x a Hamiltonian cycle C in H . They can then run the ZK-Ham protocol to prove that indeed H is Hamiltonian (and hence the original formula was satisfiable) without revealing any information the verifier could not have obtained on his own.

14.5 Parallel repetition and turning zero knowledge proofs to signatures.

While we talked about amplifying zero knowledge proofs by running them n times one *after* the other, one could also imagine running the n copies *in parallel*. It is not trivial that we get the same benefit of reducing the error to 2^{-n} but it turns out that we do in the cases we are interested in here. Unfortunately, zero knowledge is not necessarily preserved. It's an important open problem whether zero knowledge is preserved for the ZK-Ham protocol mentioned above.

However, Fiat and Shamir showed that in protocols (such as the ones we showed here) where the verifier only sends random bits, then if we replaced this verifier by a *random function*, then both soundness and zero knowledge are preserved. This suggests a *non-interactive* version of these protocols in the random oracle model, and this is indeed widely used. Schnorr designed signatures based on this non interactive version.

14.5.1 “Bonus features” of zero knowledge

- Proof of knowledge
- Deniability / non-transferability

Fully homomorphic encryption: Introduction and bootstrapping

In today's era of "cloud computing", much of individual's and businesses' data is stored and computed on by third parties such as Google, Microsoft, Apple, Amazon, Facebook, Dropbox and many others. Classically, cryptography provided solutions to protecting **data in motion** from point A to point B. But these are not always sufficient to protect **data at rest** and particularly **data in use**. For example, suppose that *Alice* has some data $x \in \{0,1\}^n$ (in modern applications x would well be terabytes in length or larger) that she wishes to store with the cloud service *Bob*, but is afraid that Bob will be hacked, subpoenaed or simply does not completely trust Bob.

Encryption does not seem to immediately solve the problem. Alice could store at Bob an *encrypted* version of the data and keep the secret key for herself. But then she would be at a loss if she wanted to do with the data anything more than retrieving particular blocks of it. If she wanted to outsource computation to Bob as well, and compute $f(x)$ for some function f , then she would need to share the secret key with Bob, thus defeating the purpose of encrypting the data in the first place.

For example, after the computing systems of Office of Personnel Management (OPM) were **discovered to be hacked** in June of 2015, revealing sensitive information, including fingerprints and all data gathered during security clearance checks of up to 18 million people, DHS assistant secretary for cybersecurity and communications Andy Ozment **said** that encryption wouldn't have helped preventing it since "if an adversary has the credentials of a user on the network, then they can access data even if it's encrypted, just as the users on the network have to access data". So, can we encrypt data in a way

that still allows some access and computing on it?

Already in 1978, Rivest, Adleman and Dertouzos considered this problem of a business that wishes to use a “commercial time-sharing service” to store some sensitive data. They envisioned a potential solution for this task which they called a privacy homomorphism. This notion later became known as *fully homomorphic encryption* (*FHE*) which is an encryption that allows a party (such as the cloud provider) that *does not know the secret key* to modify a ciphertext c encrypting x to a ciphertext c' encrypting $f(x)$ for every efficiently computable $f()$. In particular in our scenario above (see Fig. 15.1), such a scheme will allow Bob, given an encryption of x , to compute the encryption of $f(x)$ and send this ciphertext to Alice without ever getting the secret key and so without ever learning anything about x (or $f(x)$ for that matter).

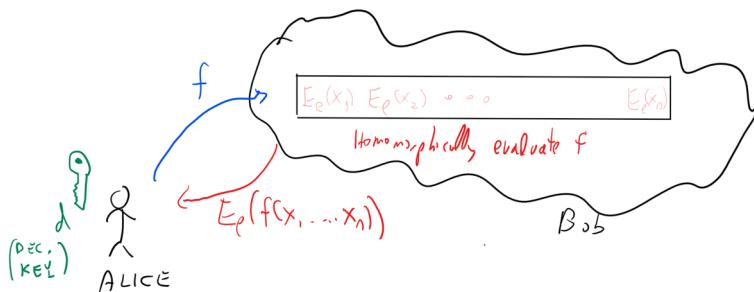


Figure 15.1: A fully homomorphic encryption can be used to store data on the cloud in encrypted form, but still have the cloud provider be able to evaluate functions on the data in encrypted form (without ever learning either the inputs or the outputs of the function they evaluate).

Unlike the case of a trapdoor function, where it only took a year for Diffie and Hellman’s challenge to be answered by RSA, in the case of fully homomorphic encryption for more than 30 years cryptographers had no constructions achieving this goal. In fact, some people suspected that there is something inherently incompatible between the security of an encryption scheme and the ability of a user to perform all these operations on ciphertexts. Stanford cryptographer Dan Boneh used to joke to incoming graduate students that he will immediately sign the thesis of anyone who came up with a fully homomorphic encryption. But he never expected that he will actually encounter such a thesis, until in 2009, Boneh’s student Craig Gentry released a paper doing just that. Gentry’s paper shook the world of cryptography, and instigated a flurry of research results making his scheme more efficient, reducing the assumptions it relied on, extending and applying it, and much more. In particular, Brakerski and Vaikuntanathan managed to obtain a fully homomorphic encryption

scheme based only on the *Learning with Error (LWE)* assumption we have seen before.

Although there is [open source library](#), as well as [other implementations](#), there is still much work to be done in order to turn FHE from theory to practice. For a comparable level of security, the encryption and decryption operations of a fully homomorphic encryption scheme are several orders of magnitude slower than a conventional public key system, and (depending on its complexity) homomorphically evaluating a circuit can be significantly more taxing. However, this is a fast evolving field, and already since 2009 significant optimizations have been discovered that reduced the computational and storage overhead by many orders of magnitudes. As in public key encryption, one would imagine that for larger data one would use a “hybrid” approach of combining FHE with symmetric encryption, though one might need to come up with tailor-made symmetric encryption schemes that can be efficiently evaluated.¹

In this lecture and the next one we will focus on the fully homomorphic encryption schemes that are *easiest to describe*, rather than the ones that are most *efficient* (though the efficient schemes share many similarities with the ones we will talk about). As is generally the case for lattice based encryption, the current most efficient schemes are based on *ideal* lattices and on assumptions such as ring LWE or the security of the NTRU cryptosystem.²

R

Lesson from verifying computation To take the distance between theory and practice in perspective, it might be useful to consider the case of *verifying computation*. In the early 1990’s researchers (motivated initially by zero knowledge proofs) came up with the notion of [probabilistically checkable proofs \(PCP’s\)](#) which could yield in principle extremely succinct ways to check correctness of computation. Probabilistically checkable proofs can be thought of as “souped up” versions of NP completeness reductions and like these reductions, have been mostly used for *negative* results, especially since the initial proofs were extremely complicated and also included enormous hidden constants. However, with time people have slowly understood these better and made them more efficient (e.g., see [this survey](#)) and it has now reached the point where these results, are [nearly practical](#) (see also [this](#)) and in fact these ideas underly at least one [startup](#). Overall, constructions for verifying computation have improved by at least 20 orders of magnitude over the last two decades. (We will talk about some of

¹ In 2012 the state of art on homomorphically evaluating AES was about six orders of magnitude slower than non-homomorphic AES computation. I don’t know what’s the current record.

² As we mentioned before, as a general rule of thumb, the difference between the ideal schemes and the one that we describe is that in the ideal setting one deals with *structured* matrices that have a compact representation as a single vector and also enable fast FFT-like matrix-vector multiplication. This saves a factor of about n in the storage and computation requirements (where n is the dimension of the subspace/lattice). However, there can be some subtle security implications for ideal lattices as well, see e.g., [here](#), [here](#), [here](#), and [here](#).

these constructions later in this course.) If progress on fully homomorphic encryption follows a similar trajectory, then we can expect the road to practical utility to be very long, but there is hope that it's not a "bridge to nowhere".

(R)

Poor man's FHE via hardware Since large scale fully homomorphic encryption is still impractical, people have been trying to achieve at least weaker security goals using certain assumptions. In particular Intel chips have so called "**Secure enclaves**" which one can think of as a somewhat tamper-protected region of the processor that is supposed to be out of reach for the outside world. The idea is that a cloud provider client would treat this enclave as a trusted party that it can communicate with through the cloud provider. The client can store their data on the cloud encrypted with some key k , and then set up a secure channel with the enclave using an authenticated key exchange protocol, and send k over. Then, when the client sends over a function f to the cloud provider, the latter party can simulate FHE by asking the enclave to compute the encryption of $f(x)$ given the encryption of x . In this solution ultimately the private key does reside on the cloud provider's computers, and the client has to trust the security of the enclave. In practice, this could provide reasonable security against remote hackers, but (unlike FHE) probably not against sophisticated attackers (e.g., governments) that have physical access to the server.

15.1 Defining fully homomorphic encryption

We start by defining *partially homomorphic* encryption. We focus on encryption for single bits. This is without loss of generality for CPA security (CCA security is anyway ruled out for homomorphic encryption- can you see why?), though there are more efficient constructions that encrypt several bits at a time.

Definition 15.1 — Partially Homomorphic Encryption. Let $\mathcal{F} = \cup \mathcal{F}_\ell$ be a class of functions where every $f \in \mathcal{F}_\ell$ maps $\{0, 1\}^\ell$ to $\{0, 1\}$.

An \mathcal{F} -homomorphic public key encryption scheme is a CPA secure public key encryption scheme (G, E, D) such that there exists a polynomial-time algorithm $EVAL : \{0, 1\}^* \rightarrow \{0, 1\}^*$ such that for every $(e, d) = G(1^n)$, $\ell = poly(n)$, $x_1, \dots, x_\ell \in \{0, 1\}$, and $f \in \mathcal{F}_\ell$ of

description size $|f|$ at most $\text{poly}(\ell)$ it holds that:

- $c = \text{EVAL}_e(f, E_e(x_1), \dots, E_e(x_\ell))$ has length at most n
- $D_d(c) = f(x_1, \dots, x_\ell)$.



Please stop and verify you understand the definition. In particular you should understand why some bound on the length of the output of *EVAL* is needed to rule out trivial constructions that are the analogous of the cloud provider sending over to Alice the entire encrypted database every time she wants to evaluate a function of it. By artificially increasing the randomness for the key generation algorithm, this is equivalent to requiring that $|c| \leq p(n)$ for some fixed polynomial $p(\cdot)$ that does not grow with ℓ or $|f|$. You should also understand the distinction between ciphertexts that are the output of the encryption algorithm on the plaintext b , and ciphertexts that decrypt to b , see Fig. 15.2.

A *fully homomomorphic encryption* is simply a partially homomorphic encryption scheme for the family \mathcal{F} of *all* functions, where the description of a function is as a circuit (say composed of **NAND** gates, which are known to be a universal basis).

15.1.1 Another application: fully homomorphic encryption for verifying computation

The canonical application of fully homomorphic encryption is for a client to store encrypted data $E(x)$ on a server, send a function f to the server, and get back the encryption $E(f(x))$ of $f(x)$. This ensures that the server does not learn any information about x , but does not ensure that it actually computes the correct function!

Here is a cute protocol to achieve the latter goal (due to [Chung Kalai and Vadhan](#)). Curiously the protocol involves “doubly encrypting” the input, and homomorphically evaluating the *EVAL* function itself.

- **Assumptions:** We assume that all functions f that the client will be interested in can be described by a string of length n .
- **Preprocessing:** The client generates a pair of keys (e, d) . In the initial stage the client computes the encrypted database $\bar{c} = E_e(x)$ and sends \bar{c}, e, e' to the server. It also computes $c^* = E_e(f^*)$ for some function f^* as well as $C^{**} = \text{EVAL}_e(\text{eval}, E_e(f^*) \| \bar{c})$ for some

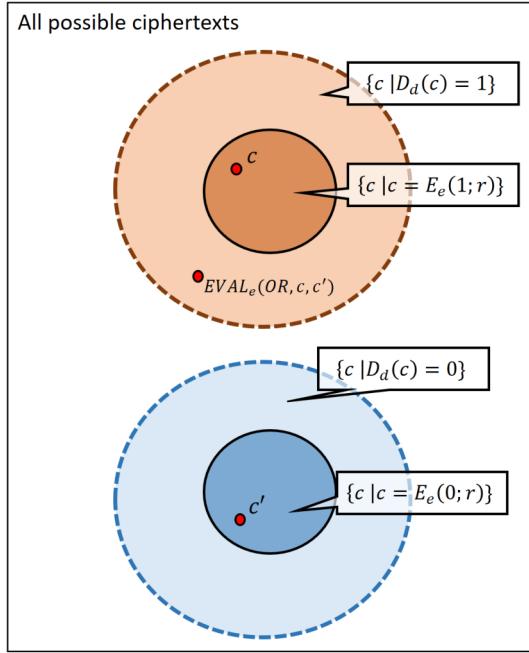


Figure 15.2: In a valid encryption scheme E , the set of ciphertexts c such that $D_d(c) = b$ is a superset of the set of ciphertexts c such that $c = E_e(b; r)$ for some $r \in \{0, 1\}^t$ where t is the number of random bits used by the encryption algorithm. Our definition of partially homomorphic encryption scheme requires that for every $f : \{0, 1\}^\ell \rightarrow \{0, 1\}$ in our family and $x \in \{0, 1\}^\ell$, if $c_i \in E_e(x_i; \{0, 1\}^t)$ for $i = 1..l$ then $EVAL(f, c_1, \dots, c_\ell)$ is in the superset $\{c \mid D_d(c) = f(x)\}$ of $E_e(f(x); \{0, 1\}^t)$. For example if we apply $EVAL$ to the OR function and ciphertexts c, c' that were obtained as encryptions of 1 and 0 respectively, then the output is a ciphertext c'' that would be decrypted to $OR(1, 0) = 1$, even if c'' is not in the smaller set of possible outputs of the encryption algorithm on 1. This distinction between the smaller and larger set is the reason why we cannot automatically apply the $EVAL$ function to ciphertexts that are obtained from the outputs of previous $EVAL$ operations.

function f^* and keeps c^*, c^{**} for herself, where $\text{eval}(f, x) = f(x)$ is the circuit evaluation function.

- **Client query:** To ask for an evaluation of f , the client generates a new random FHE keypair (e', d') , chooses $b \leftarrow_R \{0,1\}$ and lets $c_b = E_{e'}(E_e(f))$ and $c_{1-b} = E_{e'}(c^*)$. It sends the triple e', c_0, c_1 to the server.
- **Server response:** Given the queries c_0, c_1 , the server defines the function $g : \{0,1\}^* \rightarrow \{0,1\}^*$ where $g(c) = \text{EVAL}_e(\text{eval}, c \parallel \bar{c})$ (for the fixed \bar{c} received) and computes c'_0, c'_1 where $c'_b = \text{EVAL}_{e'}(g_b, c_b)$. (Please pause here and make sure you understand what this step is doing! Note that we use here crucially the fact that EVAL itself is a polynomial time computation.)
- **Client check:** Client checks whether $D_{d'}(c'_{1-b}) = c^{**}$ and if so accepts $D_d(D_{d'}(c'_b))$ as the answer.

We claim that if the server cheats then the client will detect this with probability $1/2 - \text{negl}(n)$. Working this out is a great exercise. The probability of detection can be amplified to $1 - \text{negl}(n)$ using appropriate repetition, see the paper for details.

15.2 Example: An XOR homomorphic encryption

It turns out that Regev's LWE-based encryption LWEENC we saw before is homomorphic with respect to the class of linear (mod 2) functions. Let us recall the LWE assumption and the encryption scheme based on it.

Definition 15.2 — LWE (simplified decision variant). Let $q = q(n)$ be some function mapping the natural numbers to primes. The $q(n)$ -*decision learning with error* ($q(n)$ -dLWE) *conjecture* is the following: for every $m = \text{poly}(n)$ there is a distribution LWE_q over pairs (A, s) such that:

- A is an $m \times n$ matrix over \mathbb{Z}_q and $s \in \mathbb{Z}_q^n$ satisfies $s_1 = \lfloor \frac{q}{2} \rfloor$ and $|As|_i \leq \sqrt{q}$ for every $i \in \{1, \dots, m\}$.
- The distribution A where (A, s) is sampled from LWE_q is computationally indistinguishable from the uniform distribution of $m \times n$ matrices over \mathbb{Z}_q .

The LWE conjecture is that $q(n)$ -dLWE holds for every $q(n)$ that is at most $\text{poly}(n)$. This is not exactly the same phrasing we used before, but can be shown to be essentially equivalent to it.

P It is a good idea for you to pause here and try to show the equivalence on your own.

The reason the two conjectures are equivalent are the following. Before we phrased the conjecture as recovering s from a pair (A', y) where $y = A's' + e$ and $|e_i| \leq \delta q$ for every i . We then showed a *search to decision* reduction ([Theorem 11.1](#)) demonstrating that this is equivalent to the task of distinguishing between this case and the case that y is a random vector. If we now let $\alpha = \lfloor \frac{q}{2} \rfloor$ and $\beta = \alpha^{-1} \pmod{q}$, and consider the matrix $A = (-\beta y | A')$ and the column vector $s = \begin{pmatrix} \alpha \\ s' \end{pmatrix}$ we see that $As = e$. Note that if y is a random vector in \mathbb{Z}_q^m then so is $-\beta y$ and so the current form of the conjecture follows from the previous one. (To reduce the number of free parameters, we fixed δ to equal $1/\sqrt{q}$; in this form the conjecture becomes stronger as q grows.)

A linearly-homomorphic encryption scheme: We can describe the encryption scheme LWEENC presented in class as:

- Key generation: Choose (A, s) from LWE_q where m satisfies $q^{1/4} \gg m \log q \gg n$.
- To encrypt $b \in \{0, 1\}$, choose $w \in \{0, 1\}^m$ and output $w^\top A + (b, 0, \dots, 0)$
- To decrypt $c \in \mathbb{Z}_q^n$, output 0 iff $|\langle c, s \rangle| \leq q/10$, where for $x \in \mathbb{Z}_q$ we defined $|x| = \min\{x, q - x\}$.

The decryption algorithm recovers the original plaintext since $\langle c, s \rangle = w^\top As + s_1 b$ and $|w^\top As| \leq m\sqrt{q} \ll q$. It turns out that this scheme is homomorphic with respect to the class of *linear functions* modulo 2. Specifically we make the following claim:

Lemma 15.3 For every $\ell \ll q^{1/4}$, there is an algorithm $EVAL_\ell$ that on input c_1, \dots, c_ℓ encrypting via LWEENC bits $b_1, \dots, b_\ell \in \{0, 1\}$, outputs a ciphertext c encrypting $b_1 \oplus \dots \oplus b_\ell$.

P This claim is not hard to prove, but working it out for yourself can be a good way to get more familiarity with LWEENC and the kind of manipulations we'll be making time and again in the constructions of many lattice based cryptographic primitives. Try to show that $c = c_1 + \dots + c_\ell$ (where addition is done as vectors in \mathbb{Z}_q) will be the encryption $b_1 \oplus \dots \oplus b_\ell$.

Proof of Lemma 15.3. The proof is quite simple. *EVAL* will simply add the ciphertexts as vectors in \mathbb{Z}_q . If $c = \sum c_i$ then

$$\langle c, s \rangle = \sum b_i \lfloor \frac{q}{2} \rfloor + \xi \mod q \quad (15.1)$$

where $\xi \in \mathbb{Z}_q$ is a “noise term” such that $|\xi| \leq \ell m \sqrt{q} \ll q$. Since $|\lfloor \frac{q}{2} \rfloor - \frac{q}{2}| < 1$, adding at most ℓ terms of this difference adds at most ℓ , and so we can also write

$$\langle c, s \rangle = \lfloor \sum b_i \frac{q}{2} \rfloor + \xi' \mod q \quad (15.2)$$

for $|\xi'| \leq \ell m \sqrt{q} + \ell \ll q$. If $\sum b_i$ is even then $\sum b_i \frac{q}{2}$ is an integer multiple of q and hence in this case $|\langle c, s \rangle| \ll q$. If $\sum b_i$ is odd $\lfloor \sum b_i \frac{q}{2} \rfloor = \lfloor q/2 \rfloor \mod q$ and so in this case $|\langle c, s \rangle| = q/2 \pm o(q) > q/10$. ■

Several other encryption schemes are also homomorphic with respect to linear functions, and even before Gentry’s construction people have managed to achieve homomorphism with respect to slightly larger classes (e.g., quadratic functions by Boneh, Goh and Nissim) but not significantly so.

15.2.1 Abstraction: A trapdoor pseudorandom generator.

It is instructive to consider the following abstraction (which we’ll use in the next lecture) of the above encryption scheme as a *trapdoor generator* (see Fig. 15.3). On input 1^n key generation algorithm outputs a vector $s \in \mathbb{Z}_q^m$ with $s_1 = \lfloor \frac{q}{2} \rfloor$ and a probabilistic algorithm G_s such that the following holds:

- Any polynomial number of samples from the distribution $G_s(1^n)$ is computationally indistinguishable from independent samples from the uniform distribution over \mathbb{Z}_q^n
- If c is output by $G_s(1^n)$ then $|\langle c, s \rangle| \leq n\sqrt{q}$.

Thus s can be thought of a “trapdoor” for the generator that allows to distinguish between a random vector $c \in \mathbb{Z}_q^n$ (that with high probability would satisfy $|\langle c, s \rangle| \geq q/10$) and an output of the generator. We use G_s to encrypt a bit b by letting $c \leftarrow_R G_s(1^n)$ and outputting $c + (b, 0, \dots, 0)^\top$. In the particular instantiation above we obtain G_s by sampling the matrix A from the LWE assumption and having G_s output $w^\top A$ for a random $w \in \{0, 1\}^n$, but we can ignore this particular implementation detail in the forgoing.

Note that this trapdoor generator satisfies the following stronger property: we can generate an alternative generator G' such that the description of G' is indistinguishable from the description of G_s

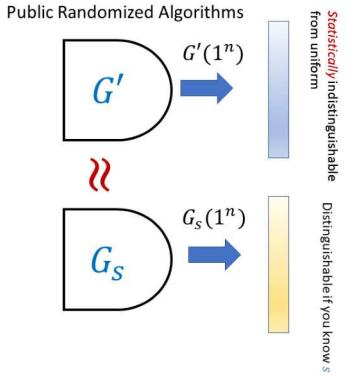


Figure 15.3: In a *trapdoor generator*, we have two ways to generate randomized algorithms. That is, we have some algorithms GEN and GEN' such that GEN outputs a pair (G_s, s) and GEN' outputs G' with G_s, G' being themselves algorithms (e.g., randomized circuits). The conditions we require are that (1) the descriptions of the circuits G_s and G' (considering them as distributions over strings) are computationally indistinguishable and (2) the distribution $G'(1^n)$ is *statistically indistinguishable* from the uniform distribution , (3) there is an efficient algorithm that given the secret “trapdoor” s can distinguish the output of G_s from the uniform distribution. In particular (1),(2), and (3) together imply that it is *not* feasible to extract s from the description of G_s .

but such that G' actually does produce (up to exponentially small statistical error) the uniform distribution over \mathbb{Z}_q^n . We can define trapdoor generators formally as follows

Definition 15.4 — Trapdoor generators. A *trapdoor generator* is a pair of randomized algorithms GEN, GEN' that satisfy the following:

- On input 1^n , GEN outputs a pair (G_s, s) where G_s is a string describing a *randomized circuit* that itself takes 1^n as input and outputs a string of length t where $t = t(n)$ is some polynomial.
- On input 1^n , GEN' outputs G' where G' is a string describing a randomized circuit that itself takes 1^n as input.
- The distributions $GEN(1^n)_1$ (i.e., the first output of $GEN(1^n)$) and $GEN'(1^n)$ are computationally indistinguishable
- With probability $1 - negl(n)$ over the choice of G' output by GEN' , the distribution $G'(1^n)$ is *statistically indistinguishable* (i.e., within $negl(n)$ total variation distance) from U_t .
- There is an efficient algorithm T such that for every pair (G_s, s) output by GEN , $\mathbb{P}[T(s, G_s(1^n)) = 1] \geq 1 - negl(n)$ (where this probability is over the internal randomness used by G_s on the input 1^n) but $\mathbb{P}[T(s, U_t) = 1] \leq 1/3$.³

³ The choice of $1/3$ is arbitrary, and can be amplified as needed.

P This is not an easy definition to parse, but looking at Fig. 15.3 can help. Make sure you understand why LWEENC gives rise to a trapdoor generator satisfying all the conditions of Definition 15.4.

Aside: trapdoor generators in real life: In the above we use the notion of a “trapdoor” in the pseudorandom generator as a mathematical abstraction, but generators with actual trapdoors have arisen in practice. In 2007 the National Institute of Standards (NIST) released standards for pseudorandom generators. Pseudorandom generators are the quintessential private key primitive, typically built out of hash functions, block ciphers, and such and so it was surprising that NIST included in the list a pseudorandom generator based on public key tools - the **Dual EC DRBG** generator based on elliptic curve cryptography. This was already strange but became even more worrying when Microsoft researchers Dan Shumow and Niels Ferguson **showed** that this generator *could* have a trapdoor in the sense that it contained some hard-wired constants that if generated in a particular way, there would be some information that (just like in G_s above) allows to distinguish the generator from random (see here for a [2007 blog post](#) on this issue). We learned more about this when leaks from the Snowden document **showed** that the NSA secretly paid 10 million dollars to RSA to make this generator the default option in their Bsafe software. You’d think that this generator is long dead but it turns out to be the “gift that keeps on giving”. In December of 2015, Juniper systems **announced** that they have discovered a malicious code in their system, dating back to at least 2012 (possibly [2008](#)), that would allow an attacker to surreptitiously decrypt all VPN traffic through their firewalls. The issue is that Juniper has been using the Dual EC DRBG and someone has managed to replace the constant they were using with another one, one that they presumably knew the trapdoor for (see [here](#) and [here](#) for more; of course unless you know to check for this, it’s very hard by looking at the code to see that one arbitrary looking constant has been replaced by another). Apparently, even though this is very surprising to many people in law enforcement and government, inserting back doors into cryptographic primitives might end up making them less secure.

15.3 From linear homomorphism to full homomorphism

Gentry's breakthrough had two components:

- First, he gave a scheme that is homomorphic with respect to arithmetic circuits (involving not just addition but also multiplications) of *logarithmic depth*.
- Second, he showed the amazing “bootstrapping theorem” that if a scheme is homomorphic enough to evaluate its own decryption circuit, then it can be turned into a *fully homomorphic* encryption that can evaluate *any* function.

Combining these two insights led to his fully homomorphic encryption.⁴

In this lecture we will focus on the second component - the bootstrapping theorem. We will show a “partially homomorphic encryption” (based on a later work of Gentry, Sahai and Waters) that can fit that theorem in the next lecture.

15.4 Bootstrapping: Fully Homomorphic “escape velocity”

The bootstrapping theorem is quite surprising. A priori you might expect that given that a homomorphic encryption for linear functions was not trivial to do, a homomorphic encryption for quadratics would be harder, cubics even harder and so on and so forth. But it turns out that there is some special degree t^* such that if we obtain homomorphic encryption for degree t^* polynomials then we can obtain *fully* homomorphic encryption that works for *all* functions. (Specifically, if the decryption algorithm $c \mapsto D_d(c)$ is a degree t polynomial, then homomorphically evaluating polynomials of degree $t^* = 2t$ will be sufficient.) That is, it turns out that once an encryption scheme is strong enough to *homomorphically evaluate its own decryption algorithm* then we can use it to obtain a fully homomorphic encryption by “pulling itself up by its own bootstraps”. One analogy is that at this point the encryption reaches “escape velocity” and we can continue onwards evaluating gates in perpetuity.

We now show the bootstrapping theorem:

⁴ The story is a bit more complex than that. Frustratingly, the decryption circuit of Gentry's basic scheme was just a little bit too deep for the bootstrapping theorem to apply. A lesser man, such as yours truly, would at this point surmise that fully homomprphic encryption was just not meant to be, and perhaps take up knitting or playing bridge as an alternative hobby. However, Craig persevered and managed to come up with a way to “squash” the decryption circuit so it can fit the bootstrapping parameters. Follow up works, and in particular the paper of Brakerski and Vaikuntanathan, managed to get schemes with much better relation between the homomorphism depth and decryption circuit, and hence avoid the need for squashing and also improve the security assumptions.

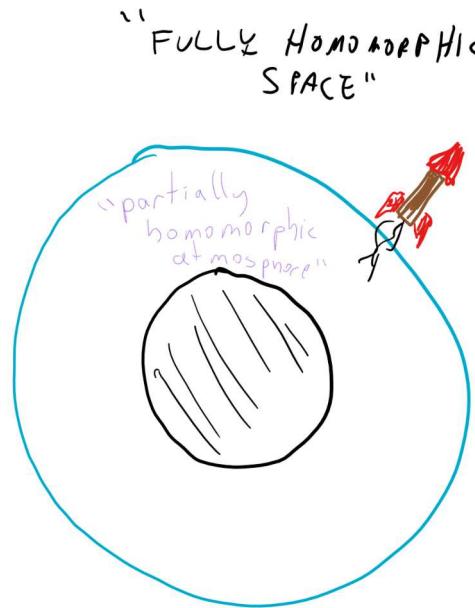


Figure 15.4: The “Bootstrapping Theorem” shows that once a partially homomorphic encryption scheme is homomorphic with respect to a rich enough family of functions, and specifically a family that contains its own decryption algorithm, then it can be converted to a fully homomorphic encryption scheme that can be used to evaluate *any* function.

Theorem 15.5 — Bootstrapping Theorem, Gentry 2009. Suppose that (G, E, D) is a CPA circular⁵ secure partially homomorphic encryption scheme for the family \mathcal{F} and suppose that for every pair of ciphertexts c, c' the map $d \mapsto D_d(c) \text{ NAND } D_d(c')$ is in \mathcal{F} . Then (G, E, D) can be turned a fully homomorphic encryption scheme.

⁵ You can ignore the condition of circular security in a first read - we will discuss it later.

15.4.1 Radioactive legos analogy

Here is one analogy for bootstrapping, inspired by Gentry’s [survey](#). Suppose that you need to construct some complicated object from a highly toxic material (see Fig. 15.5). You are given a supply of sealed bags that are flexible enough so you can manipulate the object from outside the bag. However, each bag can only hold for 10 seconds of such manipulations before it leaks. The idea is that if you can open one bag inside another within 9 seconds then you can perform the manipulations for arbitrary length. That is, if the object is in the i^{th} bag then you put this bag inside the $i + 1^{st}$ bag, spend 9 seconds on opening the i^{th} bag inside the $i + 1^{st}$ bag and then spend another second of whatever manipulations you wanted to perform. We then



Figure 15.5: To build a castle from radioactive Lego bricks, which can be kept safe in a special ziploc bag for 10 seconds, we can: 1) Place the bricks in a bag, and place the bag inside an outer bag. 2) Manipulate the inner bag through the outer bag to remove the bricks from it in 9 seconds, and spend 1 second putting one brick in place 3) Now the outer bag has 9 seconds of life left, and we can put it inside a new bag and repeat the process.

continue this process by putting the $i + 1^{\text{st}}$ bag inside the $i + 2^{\text{nd}}$ bag and so on and so forth.

15.4.2 Proving the bootstrapping theorem

We now turn to the formal proof of [Theorem 15.5](#)

Proof. The idea behind the proof is simple but ingenious. Recall that the NAND gate $b, b' \mapsto \neg(b \wedge b')$ is a universal gate that allows us to compute any function $f : \{0,1\}^n \rightarrow \{0,1\}$ that can be efficiently computed. Thus, to obtain a fully homomorphic encryption it suffices to obtain a function *NANDEVAL* such that $D_d(\text{NANDEVAL}(c, c')) = D_d(c) \text{ NAND } D_d(c')$. (Note that this is stronger than the typical notion of homomorphic evaluation since we require that *NANDEVAL* outputs an encryption of $b \text{ NAND } b'$ when given *any* pair of ciphertexts that decrypt to b and b' respectively, regardless whether these ciphertexts were produced by the encryption algorithm or by some other method, including the *NANDEVAL* procedure itself.)

Thus to prove the theorem, we need to modify (G, E, D) into an encryption scheme supporting the *NANDEVAL* operation. Our new scheme will use the same encryption algorithms E and D but the

following modification G' of the key generation algorithm: after running $(d, e) = G(1^n)$, we will append to the public key an encryption $c^* = E_e(d)$ of the secret key. We have now defined the key generation, encryption and decryption. CPA security follows from the security of the original scheme, where by circular security we refer exactly to the condition that the scheme is secure even if the adversary gets a single encryption of the public key.⁶ This latter condition is not known to be implied by standard CPA security but as far as we know is satisfied by all natural public key encryptions, including the LWE-based ones we will plug into this theorem later on.

So, now all that is left is to define the *NANDEVAL* operation. On input two ciphertexts c and c' , we will construct the function $f_{c,c'} : \{0,1\}^n \rightarrow \{0,1\}$ (where n is the length of the secret key) such that $f_{c,c'}(d) = D_d(c) \text{ NAND } D_d(c')$. It would be useful to pause at this point and make sure you understand what are the inputs to $f_{c,c'}$, what are “hardwired constants” and what is its output. The ciphertexts c and c' are simply treated as fixed strings and are *not* part of the input to $f_{c,c'}$. Rather $f_{c,c'}$ is a function (depending on the strings c, c') that maps the secret key into a bit. When running *NANDEVAL* we of course do not know the secret key d , but we can still design a circuit that computes this function $f_{c,c'}$. Now $\text{NANDEVAL}(c, c')$ will simply be defined as $\text{EVAL}(f_{c,c'}, c^*)$. Since $c^* = E_e(d)$, we get that

$$D_d(\text{NANDEVAL}(c, c')) = D_d(\text{EVAL}(f_{c,c'}, c^*)) = f_{c,c'}(d) = D_d(c) \text{ NAND } D_d(c') . \quad (15.3)$$

Thus indeed we map *any* pair of ciphertexts c, c' that decrypt to b, b' into a ciphertext c'' that decrypts to $b \text{ NAND } b'$. This is all that we needed to prove. ■



Don't let the short proof fool you. This theorem is quite deep and subtle, and requires some reading and re-reading to truly “get” it.

⁶ Without this assumption we can still obtain a form of FHE known as a *leveled* FHE where the size of the public key grows with the *depth* of the circuit to be evaluated. We can do this by having ℓ public keys where ℓ is the depth we want to evaluate, and encrypt the private key of the i^{th} key with the $i + 1^{\text{st}}$ public key. However, since circular security seems quite likely to hold, we ignore this extra complication in the rest of the discussion.

16

Fully homomorphic encryption : Construction

In the last lecture we defined fully homomorphic encryption, and showed the “bootstrapping theorem” that transforms a partially homomorphic encryption scheme into a fully homomorphic encryption, as long as the original scheme can homomorphically evaluate its own decryption circuit. In this lecture we will show an encryption scheme (due to Gentry, Sahai and Waters, henceforth GSW) meeting the latter property. That is, this lecture is devoted to proving¹ the following theorem:

Theorem 16.1 — FHE from LWE. Assuming the LWE conjecture, there exists a partially homomorphic public key encryption $(G, E, D, EVAL)$ that fits the conditions of the bootstrapping theorem (Theorem 15.5). That is, for every two ciphertexts c and c' , the function $d \mapsto D_d(c) \text{ NAND } D_d(c')$ can be homomorphically evaluated by $EVAL$.

16.1 Prelude: from vectors to matrices

In the linear homomorphic scheme we saw in the last lecture, every ciphertext was a vector $c \in \mathbb{Z}_q^n$ such that $\langle c, s \rangle$ equals (up to scaling by $\lfloor \frac{q}{2} \rfloor$) the plaintext bit. We saw that adding two ciphertexts modulo q corresponded to XOR'ing (i.e., adding modulo 2) the corresponding two plaintexts. That is, if we define $c \oplus c'$ as $c + c' \pmod{q}$ then performing the \oplus operation on the ciphertexts corresponds to adding modulo 2 the plaintexts.

However, to get to a fully, or even partially, homomorphic scheme, we need to find a way to perform the NAND operation on the two plaintexts. The challenge is that it seems that to do that we need to find a way to evaluate *multiplications*: find a way to define some

¹ This theorem as stated was proven by Brakerski and Vaikuntanathan (ITCS 2014) building a line of work initiated by Gentry's original STOC 2009 work. We will actually prove a weaker version of this theorem, due to Brakerski and Vaikuntanathan (FOCS 2011), which assumes a quantitative strengthening of LWE. However, we will not follow the proof of Brakerski and Vaikuntanathan but rather a scheme of Gentry, Sahai and Waters (CRYPTO 2013). Also note that, as noted in the previous lecture, all of these results require the extra assumption of *circular security* on top of LWE to achieve a non-leveled fully homomorphic encryption scheme.

operation \otimes on ciphertexts that corresponds to multiplying the plaintexts. Alas, a priori, there doesn't seem to be a natural way to *multiply* two vectors.

The GSW approach to handle this is to move from vectors to *matrices*. As usual, it is instructive to first consider the cryptographer's dream world where Gaussian elimination doesn't exist. In this case, the GSW ciphertext encrypting $b \in \{0,1\}$ would be an $n \times n$ matrix C over \mathbb{Z}_q such that $Cs = bs$ where $s \in \mathbb{Z}_q^n$ is the secret key. That is, the encryption of a bit b is a matrix C such that the secret key is an *eigenvector* (modulo q) of C with corresponding eigenvalue b . (We defer discussion of how the encrypting party generates such a ciphertext, since this is in any case only a "dream" toy example.)

P You should make sure you understand the *types* of all the identifiers we refer to. In particular, above C is an $n \times n$ matrix with entries in \mathbb{Z}_q , s is a vector in \mathbb{Z}_q^n , and b is a scalar (i.e., just a number) in $\{0,1\}$. See Fig. 16.1 for a visual representation of the ciphertexts in this "naive" encryption scheme. Keeping track of the dimensions of all objects will become only more important in the rest of this lecture.

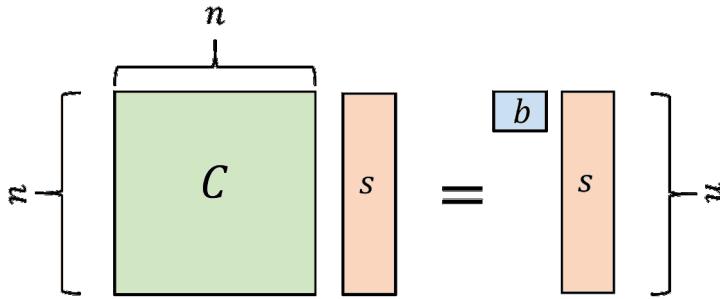


Figure 16.1: In the "naive" version of the GSW encryption, to encrypt a bit b we output an $n \times n$ matrix C such that $Cs = bs$ where $s \in \mathbb{Z}_q^n$ is the secret key. In this scheme we can transform encryptions C, C' of b, b' respectively to an encryption C'' of $NAND(b, b')$ by letting $C'' = I - CC'$.

Given C and s we can recover b by just checking if $Cs = s$ or $Cs = 0^n$. The scheme allows homomorphic evaluation of both addition (modulo q) and multiplication, since if $Cs = bs$ and $C's = b's$ then we can define $C \oplus C' = C + C'$ (where on the righthand side, addition is simply done in \mathbb{Z}_q) and $C \otimes C' = CC'$ (where again this refers to matrix multiplication in \mathbb{Z}_q).

Indeed, one can verify that both addition and multiplication

succeed since

$$(C + C')s = (b + b')s \quad (16.1)$$

and

$$CC's = C(b's) = bb's \quad (16.2)$$

where all these equalities are in \mathbb{Z}_q .

Addition modulo q is not the same as XOR, but given these multiplication and addition operations, we can implement the NAND operation as well. Specifically, for every $b, b' \in \{0, 1\}$, $b \text{ NAND } b' = 1 - bb'$. Hence we can take a ciphertext C encrypting b and a ciphertext C' encrypting b' and transform these two ciphertexts to the ciphertext $C'' = (I - CC')$ that encrypts $b \text{ NAND } b'$ (where I is the identity matrix). Thus in a world without Gaussian elimination it is not hard to get a fully homomorphic encryption.

R **Private key FHE** We have not shown how to generate a ciphertext without knowledge of s , and hence strictly speaking we only showed in this world how to get a *private key* fully homomorphic encryption. Our “real world” scheme will be a full fledged *public key* FHE. However we note that private key homomorphic encryption is already very interesting and in fact sufficient for many of the “cloud computing” applications. Moreover, **Rothblum** gave a generic transformation from a *private key* homomorphic encryption to a *public key* homomorphic encryption.

16.2 Real world partially homomorphic encryption

We now discuss how we can obtain an encryption in the real world where, as much as we’d like to ignore it, there are people who walk among us (not to mention some computer programs) that actually know how to invert matrices. As usual, the idea is to “fool Gaussian elimination with noise” but we will see that we have to be much more careful about “noise management”, otherwise even for the party holding the secret key the noise will overwhelm the signal.²

The main idea is that we can expect the following problem to be hard for a random secret $s \in \mathbb{Z}_q^n$: distinguish between samples of random matrices C and matrices where $Cs = bs + e$ for some

² For this reason, Craig Gentry called his highly recommended survey on fully homomorphic encryption and other advanced constructions [computing on the edge of chaos](#).

$b \in \{0, 1\}$ and “short” e satisfying $|e_i| \leq \sqrt{q}$ for all i . This yields a natural candidate for an encryption scheme where we encrypt b by a matrix C satisfying $Cs = bs + e$ where e is a “short” vector.³

We can now try to check what adding and multiplying two matrices does to the noise. If $Cs = bs + e$ and $C's = b's + e'$ then

$$(C + C')s = (b + b')s + (e + e') \quad (16.3)$$

and

$$CC's = C(b's + e') + e = bb's + (b'e + Ce') . \quad (16.4)$$



I recommend you pause here and check for yourself whether it will be the case that if $C + C'$ encrypts $b + b'$ and CC' encrypts bb' up to small noise or not.

We would have loved to say that we can define as above $C \oplus C' = C + C' \pmod{q}$ and $C \otimes C' = CC' \pmod{q}$. For this we would need that $(C + C')s$ equals $(b + b')s$ plus a “short” vector and $CC's$ equals $bb's$ plus a “short” vector. The former statement indeed holds. Looking at Eq. (16.4) we see that $(C + C')s$ equals $(b + b')s$ up to the “noise” vector $e + e'$, and if e, e' are “short” then $e + e'$ is not too long either. That is, if $|e_i| < \delta q$ and $|e'_i| < \delta q$ for every i then $|e_i + e'_i| < 2\delta q$. So we can at least handle a significant number of additions before the noise gets out of hand.

However, if we consider Eq. (16.4), we see that CC' will be equal to $bb's$ plus the “noise vector” $b'e + Ce'$. The first component $b'e$ of this noise vector is “short” (after all $b \in \{0, 1\}$ and e is “short”). However, the second component Ce' could be a very large vector. Indeed, since C looks like a random matrix in \mathbb{Z}_q , no matter how small the entries of e' , many of the entries of Ce' are quite likely to be of magnitude at least, say, $q/2$ and so multiplying e' by C takes us “beyond the edge of chaos”.

16.3 Noise management via encoding

The problem we had above is that the entries of C are elements in \mathbb{Z}_q that can be very large, while we would have loved them to be small numbers such as 0 or 1. At this point one could say

³ We deliberately leave some flexibility in the definition of “short”. While initially “short” might mean that $|e_i| < \sqrt{q}$ for every i , decryption will succeed as long as long as $|e_i|$ is, say, at most $q/100$.

"If only there was some way to encode numbers between 0 and $q - 1$ using only 0's and 1's"

If you think about it hard enough, it turns out that there is something known as the “binary basis” that allows us to encode a number $x \in \mathbb{Z}_q$ as a vector $\hat{x} \in \{0, 1\}^{\log q}$.⁴ What’s even more surprising is that this seemingly trivial trick turns out to be immensely useful. We will define the *binary encoding* of a vector or matrix x over \mathbb{Z}_q by \hat{x} . That is, \hat{x} is obtained by replacing every coordinate x_i with $\log q$ coordinates $x_{i,0}, \dots, x_{i,\log q-1}$ such that

$$x_i = \sum_{j=0}^{\log q-1} 2^j x_{i,j}. \quad (16.5)$$

⁴ If we were being pedantic the length of the vector (and other constant below) should be the integer $\lceil \log q \rceil$ but I omit the ceiling symbols for simplicity of notation.

Specifically, if $s \in \mathbb{Z}_q^n$, then we denote by \hat{s} the $n \log q$ -dimensional vector with entries in $\{0, 1\}$, such that each $\log q$ -sized block of \hat{s} encodes a coordinate of s . Similarly, if C is an $m \times n$ matrix, then we denote by \hat{C} the $m \times n \log q$ matrix with entries in $\{0, 1\}$ that corresponds to encoding every n -dimensional row of C by an $n \log q$ -dimensional row where each $\log q$ -sized block corresponds to a single entry. (We still think of the entries of these vectors and matrices as elements of \mathbb{Z}_q and so all calculations are still done modulo q .)

While encoding in the binary basis is not a linear operation, the *decoding* operation is linear as one can see in Eq. (16.5). We let Q be the $n \times (n \log q)$ “decoding” matrix that maps an encoding vector \hat{v} back to the original vector v . Specifically, every row of Q is composed of n blocks each of $\log q$ size, where the i -th row has only the i -th block nonzero, and equal to the values $(1, 2, 4, \dots, 2^{\log q-1})$. It’s a good exercise to verify that for every vector v and matrix C , $Q\hat{v} = v$ and $\hat{C}Q^\top = C$. (See Fig. 16.2 and Fig. 16.3.)

In our final scheme the ciphertext encrypting b will be an $(n \log q) \times (n \log q)$ matrix C with small coefficients such that $Cv = bv + e$ for a “short” $e \in \mathbb{Z}_q^{n \log q}$ and $v = Q^\top s$ for $s \in \mathbb{Z}_q^n$. Now given ciphertexts C, C' that encrypt b, b' respectively, we will define $C \oplus C' = C + C' \pmod{q}$ and $C \otimes C' = \widehat{(CQ^\top)}C'$.

Since we have $Cv = bv + e$ and $C'v = b'v + e'$ we get that

$$(C \oplus C')v = (C + C')v = (b + b')v + (e + e') \quad (16.6)$$

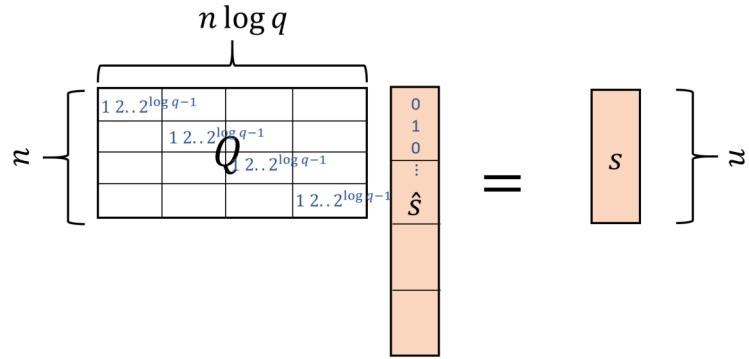


Figure 16.2: We can encode a vector $s \in \mathbb{Z}_q^n$ as a vector $\hat{s} \in \mathbb{Z}_q^{n \log q}$ that has only entries in $\{0, 1\}$ by using the binary encoding, replacing every coordinate of s with a $\log q$ -sized block in \hat{s} . The decoding operation is *linear* and so we can write $s = Q\hat{s}$ for a specific (simple) $n \times (n \log q)$ matrix Q .

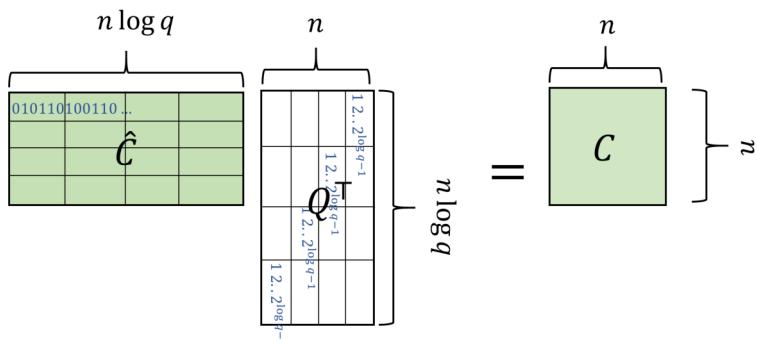


Figure 16.3: We can encode an $n \times n$ matrix C over \mathbb{Z}_q by an $n \times (n \log q)$ matrix \hat{C} using the binary basis. We have the equation $C = \hat{C}Q^\top$ where Q is the same matrix we use to decode a vector.

and

$$(C \otimes C')v = (\widehat{CQ^\top})C'v = (\widehat{CQ^\top})(bv + e') . \quad (16.7)$$

But since $v = Q^\top s$ and $\hat{A}Q^\top = A$ for every matrix A , the right-hand side of Eq. (16.7) equals

$$(\widehat{CQ^\top})(b'Q^\top s + e') = b'CQ^\top s + (\widehat{CQ^\top})e' = b'Cv + (\widehat{CQ^\top})e' \quad (16.8)$$

but since \hat{B} is a matrix with small coefficients for every B and e' is short, the righthand side of Eq. (16.8) equals $b'Cv$ up to a short vector, and since $Cv = bv + e$ and $b'e$ is short, we get that $(C \otimes C')v$ equals $b'bv$ plus a short vector as desired.

If we keep track of the parameters in the above analysis, we can see that

$$C \overline{\wedge} C' = (I - C \otimes C') \quad (16.9)$$

then if C encrypts b and C' encrypts b' with noise vectors e, e' satisfying $\max |e_i| \leq \mu$ and $\max |e'_i| \leq \mu'$ then $C \overline{\wedge} C'$ encrypts b NAND b' up to a vector of maximum magnitude at most $O(\mu + n \log q\mu')$.

16.4 Putting it all together

We now describe the full scheme. We are going to use a quantitatively stronger version of LWE. Namely, the $q(n)$ -dLWE assumption for $q(n) = 2^{\sqrt{n}}$. It is not hard to show that we can relax our assumption to $q(n)$ -LWE $q(n) = 2^{\text{polylog}(n)}$ and Brakerski and Vaikuntanathan showed how to relax the assumption to standard (i.e. $q(n) = \text{poly}(n)$) LWE though we will not present this here.

FHEENC:

- **Key generation:** As in the scheme of last lecture the secret key is $s \in \mathbb{Z}_s^n$ and the public key is a generator G_s such that samples from $G_s(1^n)$ are indistinguishable from independent random samples from \mathbb{Z}_q^n but if c is output by G_s then $|\langle c, s \rangle| < \sqrt{q}$, where the inner product (as all other computations) is done modulo q and for every $x \in \mathbb{Z}_q = \{0, \dots, q-1\}$ we define $|x| = \min\{x, q-x\}$. As before, we can assume that $s_1 = \lfloor q/2 \rfloor$ which implies that $(Q^\top s)_1$ is

also $\lfloor q/2 \rfloor$ since (as can be verified by direct inspection) the first row of Q^\top is $(1, 0, \dots, 0)$.

- **Encryption:** To encrypt $b \in \{0, 1\}$, let $d_1, \dots, d_{n \log q} \xleftarrow{R} G_s(1^n)$ output $C = \widehat{(bQ^\top + D)}$ where D is the matrix whose rows are $d_1, \dots, d_{n \log q}$ generated from G_s . (See Fig. 16.4.)
- **Decryption:** To decrypt the ciphertext C , we output 0 if $|(\widehat{CQ^\top s})_1| < 0.1q$ and output 1 if $0.6q > |(\widehat{CQ^\top s})_1| > 0.4q$, see Fig. 16.5. (It doesn't matter what we output on other cases.)
- **NAND evaluation:** Given ciphertexts C, C' , we define $\widehat{C \overline{\wedge} C'}$ (sometimes also denoted as $\text{NANDEVAL}(C, C')$) to equal $I - \widehat{(CQ^\top)}C'$, where I is the $(n \log q) \times (n \log q)$ identity matrix.



Please take your time to read the definition of the scheme, and go over Fig. 16.4 and Fig. 16.5 to make sure you understand it.

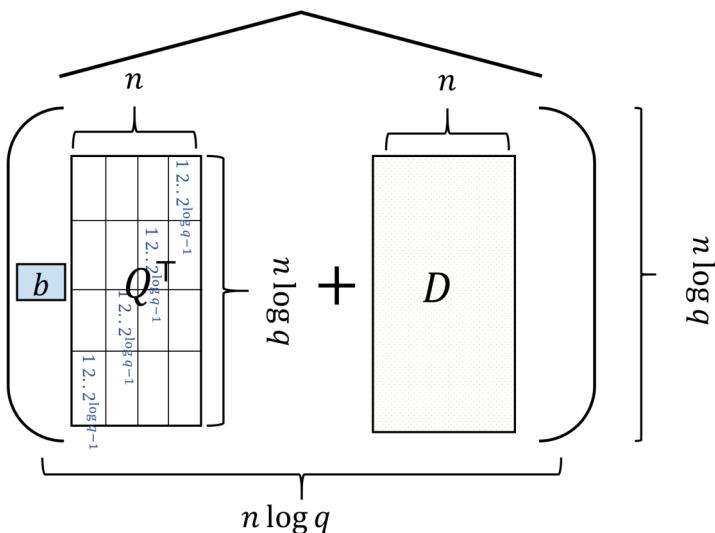


Figure 16.4: In our fully homomorphic encryption, the public key is a trapdoor generator G_s . To encrypt a bit b , we output $C = \widehat{(bQ^\top + D)}$ where D is a $(n \log q) \times n$ matrix whose rows are generated using G_s .

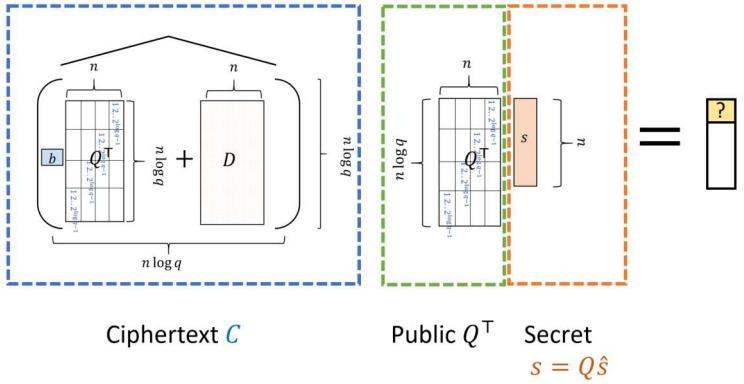


Figure 16.5: We decrypt a ciphertext $C = (\widehat{bQ^T} + D)$ by looking at the first coordinate of $CQ^T s$ (or equivalently, $CQ^T Q\hat{s}$). If $b = 0$ then this equals to the first coordinate of Ds , which is at most \sqrt{q} in magnitude. If $b = 1$ then we get an extra factor of $Q^T s$ which we set to be in the interval $(0.499q, 0.51q)$. We can think of either s or \hat{s} as our secret key.

16.5 Analysis of our scheme

To show that this scheme is a valid partially homomorphic scheme we need to show the following properties:

1. **Correctness:** The decryption of an encryption of $b \in \{0, 1\}$ equals b .
2. **CPA security:** An encryption of 0 is computationally indistinguishable from an encryption of 1 to someone that got the public key.
3. **Homomorphism:** If C encrypts b and C' encrypts b' then $C \overline{\wedge} C'$ encrypts $b \text{ NAND } b'$ (with a higher amount of noise). The growth of the noise will be the reason that we will not get immediately a fully homomorphic encryption.
4. **Shallow decryption circuit:** To plug this scheme into the bootstrapping theorem we will need to show that its decryption algorithm (or more accurately, the function in the statement of the bootstrapping theorem) can be evaluated in depth $\text{polylog}(n)$ (independently of q), and that moreover, the noise grows slowly enough that our scheme is homomorphic with respect to such circuits.

Once we obtain 1-4 above, we can plug FHEENC into the Bootstrapping Theorem (Theorem 15.5) and thus complete the proof of existence of a fully homomorphic encryption scheme. We now address those points one by one.

16.5.1 Correctness

Correctness of the scheme will follow from the following stronger condition:

Lemma 16.2 For every $b \in \{0, 1\}$, if C is the encryption of b then it is an $(n \log q) \times (n \log q)$ matrix satisfying

$$CQ^\top s = bQ^\top s + e \quad (16.10)$$

where $\max |e_i| \ll \sqrt{q}$.

Proof. For starters, let us see that the dimensions make sense: the encryption of b is computed by $C = \widehat{(bQ^\top + D)}$ where D is an $(n \log q) \times n$ matrix satisfying $|Ds|_i \leq \sqrt{q}$ for every i and I is the $(n \log q) \times (n \log q)$.

Since Q^\top is also an $(n \log q) \times n$ matrix, adding bQ^\top (i.e. either Q^\top or the all-zeroes matrix, depending on whether or not $b = 1$) to D makes sense and applying the $\widehat{\cdot}$ operation will transform every row to length $n \log q$ and hence C is indeed a square $(n \log q) \times (n \log q)$ matrix.

Let us now see what this matrix C does to the vector $v = Q^\top s$. Using the fact that $\widehat{M}Q^\top = M$ for every matrix M , we get that

$$Cv = (bQ^\top + D)s = bv + Ds \quad (16.11)$$

but by construction $|(Ds)_i| \leq \sqrt{q}$ for every i . ■

Lemma 16.2 implies correctness of decryption since by construction we ensured that $(Q^\top s)_1 \in (0.499q, 0.5001q)$ and hence we get that if $b = 0$ then $|(Cv)_1| = o(q)$ and if $b = 1$ then $0.499q - o(q) \leq |(Cv)_1| \leq 0.501q + o(q)$.

16.5.2 CPA Security

To show CPA security we need to show that an encryption of 0 is indistinguishable from an encryption of 1. However, by the security of the trapdoor generator, an encryption of b computed according to our algorithm will be indistinguishable from an encryption of b obtained when the matrix D is a random $(q \log n) \times n$ matrix. Now in this case the encryption is obtained by applying the $\widehat{\cdot}$ operation to $bQ^\top + D$ but if D is uniformly random then for every choice of b , $bQ^\top + D$ is uniformly random (since a fixed matrix plus a random

matrix yields a random matrix) and hence the matrix $bQ^\top + D$ (and so also the matrix $\widehat{bQ^\top + D}$) contains no information about b . This completes the proof of CPA security (can you see why?).

If we want to plug in this scheme in the bootstrapping theorem, then we will also assume that it is *circular secure*. It seems a reasonable assumption though unfortunately at the moment we do not know how to derive it from LWE. (If we don't want to make this assumption we can still obtain a *leveled* fully homomorphic encryption as discussed in the previous lecture.)

16.5.3 Homomorphism

Let $v = Qs$, $b \in \{0, 1\}$ and C be a ciphertext such that $Cv = bv + e$. We define the *noise* of C , denoted as $\mu(C)$ to be the maximum of $|e_i|$ over all $i \in [n \log q]$. We make the following lemma, which we'll call the “noisy homomorphism lemma”:

Lemma 16.3 Let C, C' be ciphertexts encrypting b, b' respectively with $\mu(C), \mu(C') \leq q/4$. Then $C'' = C \overline{\wedge} C'$ encrypts $b \text{ NAND } b'$ and satisfies

$$\mu(C'') \leq (2n \log q) \max\{\mu(C), \mu(C')\} \quad (16.12)$$

Proof. This follows from the calculations we have done before. As we've seen,

$$\widehat{CQ^\top} C' v = \widehat{CQ^\top} (b'v + e') = b' \widehat{CQ^\top} Q^\top s + \widehat{CQ^\top} e' = b' (Cv) + \widehat{CQ^\top} e' = bb'v + b'e + \widehat{CQ^\top} e' \quad (16.13)$$

But since $\widehat{CQ^\top}$ is a 0/1 matrix with every row of length $n \log q$, for every i $(\widehat{CQ^\top} e')_i \leq (n \log q) \max_j |e_j|$. We see that the noise vector in the product has magnitude at most $\mu(C) + n \log q \mu(C')$. Adding the identity for the NAND operation adds at most $\mu(C) + \mu(C')$ to the noise, and so the total noise magnitude is bounded by the righthand side of Eq. (16.12). ■

16.5.4 Shallow decryption circuit

Recall that to plug in our homomorphic encryption scheme into the bootstrapping theorem, we needed to show that for every ciphertexts C, C' (generated by the encryption algorithm) the function $f : \{0, 1\}^{n \log q} \rightarrow \{0, 1\}$ defined as

$$f(d) = D_d(C) \text{ NAND } D_d(C') \quad (16.14)$$

can be homomorphically evaluated where d is the secret key and $D_d(C)$ denotes the decryption algorithm applied to C .

In our case we can think of the secret key as the binary string \hat{s} which describes our vector s as a bit string of length $n \log q$. Given a ciphertext C , the decryption algorithm takes the dot product modulo q of s with the first row of CQ^\top (or, equivalently, the dot product of \hat{q} with $CQ^\top Q$) and outputs 0 (respectively 1) if the resulting number is small (respectively large).

By repeatedly applying the noisy homomorphism lemma (Lemma 16.3), we can show that can homomorphically evaluate every circuit of NAND gates whose *depth* ℓ satisfies $(2n \log q)^\ell \ll q$. If $q = 2^{\sqrt{n}}$ then (assuming n is sufficiently large) then as long as $\ell < n^{0.49}$ this will be satisfied.

In particular to show that $f(\cdot)$ can be homomorphically evaluated it will suffice to show that for every fixed vector $c \in \mathbb{Z}_q^{n \log q}$ there is a $\text{polylog}(n) \ll n^{0.49}$ depth circuit F that on input a string $\hat{s} \in \{0,1\}^{n \log q}$ will output 0 if $|\langle c, \hat{s} \rangle| < q/10$ and output 1 if $|\langle c, \hat{s} \rangle| > q/5$. (We don't care what F does otherwise. The above suffices since given a ciphertext C we can use F with the vector c being the top row of $CQ^\top Q$, and hence $\langle c, \hat{s} \rangle$ would correspond to the first entry of $CQ^\top s$. Note that if F has depth ℓ then the function $f()$ above has depth at most $\ell + 1$.)



Please make sure you understand the above argument.

If $c = (c_1, \dots, c_{n \log q})$ is a vector then to compute its inner product with a 0/1 vector \hat{s} we simply need to sum up the numbers c_i where $\hat{s}_i = 1$. Summing up m numbers can be done via the obvious recursion in depth that is $\log m$ times the depth for a single addition of two numbers. However, the naive way to add two numbers in \mathbb{Z}_q (each represented by $\log q$ bits) will have depth $O(\log q)$ which is too much for us.



Please stop here and see if you understand why the natural circuit to compute the addition of two numbers modulo q (represented as $\log q$ -length binary strings) will require depth $O(\log q)$. As a hint, one needs to keep track of the "carry".

Fortunately, because we only care about accuracy up to $q/10$, if we add m numbers, we can drop all but the first $100 \log m$ most

significant digits of our numbers, since including them can change the sum of the n numbers by at most $m(q/m^{100}) \ll q$. Hence we can easily do this work in $\text{poly}(\log m)$ depth, which is $\text{poly}(\log n)$ since $m = \text{poly}(n)$.

Let us now show this more formally:

Lemma 16.4 For every $c \in \mathbb{Z}_q^m$ there exists some function $f : \{0, 1\}^m \rightarrow \{0, 1\}$ such that:

1. For every $\hat{s} \in \{0, 1\}^n$ such that $|\langle \hat{s}, c \rangle| < 0.1q$, $f(\hat{s}) = 0$
2. For every $\hat{s} \in \{0, 1\}^n$ such that $0.4q < |\langle \hat{s}, c \rangle| < 0.6q$, $f(\hat{s}) = 1$
3. There is a circuit computing f of depth at most $100(\log m)^3$.

Proof. For every number $x \in \mathbb{Z}_q$, write \tilde{x} to be the number that is obtained by writing x in the binary basis and setting all digits except the $10 \log m$ most significant ones to zero.

Note that $\tilde{x} \leq x \leq \tilde{x} + q/m^{10}$. We define $f(\hat{s})$ to equal 1 if $|\sum \hat{s}_i \tilde{c}_i \pmod{\tilde{q}}| \geq 0.3\tilde{q}$ and to equal 0 otherwise (where as usual the absolute value of x modulo \tilde{q} is the minimum of x and $\tilde{q} - x$.) Note that all numbers involved have zeroes in all but the $10 \log m$ most significant digits and so these less significant digits can be ignored. Hence we can add any pair of such numbers modulo \tilde{q} in depth $O(\log m)^2$ using the standard elementary school algorithm to add two ℓ -digit numbers in $O(\ell^2)$ steps. Now we can add the m numbers by adding pairs, and then adding up the results, and this way in a binary tree of depth $\log m$ to get a total depth of $O(\log m)^3$. So, all that is left to prove is that this function f satisfies the conditions (1) and (2).

Note that $|\sum \hat{s}_i \tilde{c}_i - \sum \hat{s}_i c_i| < mq/m^{10} = q/m^9$ so now we want to show that the effect of taking modulo \tilde{q} is not much different from taking modulo q . Indeed, note that this sum (before a modular reduction) is an integer between 0 and qm . If x is such an integer and we divide x by q to write $x = kq + r$ for $r < q$, then since $x < qm$, $k < m$, and so we can write $x = k\tilde{q} + k(q - \tilde{q}) + r$ so the difference between $k \pmod{q}$ and $k \pmod{\tilde{q}}$ will be (in our standard modular metric) at most $mq/m^{10} = q/m^9$. Overall we get that if $\sum \hat{s}_i c_i \pmod{q}$ is in the interval $[0.4q, 0.6q]$ then $\sum \hat{s}_i \tilde{c}_i \pmod{\tilde{q}}$ will be in the interval $[0.4q - 100q/m^9, 0.6q + 100q/m^9]$ which is contained in $[0.3\tilde{q}, 0.7\tilde{q}]$. ■

This completes the proof that our scheme can fit into the bootstrapping theorem (i.e., of [Theorem 16.1](#)), hence completing the description of the fully homomorphic encryption scheme.



Now would be a good point to go back and see you understand how all the pieces fit together to obtain the complete construction of the fully homomorphic encryption scheme.

16.6 Example application: Private information retrieval

To be completed

Multiparty secure computation I: Definition and Honest-But-Curious to Malicious complier

Wikipedia [defines](#) cryptography as “the practice and study of techniques for secure communication in the presence of third parties called adversaries”. However, I think a better definition would be:

Cryptography is about replacing trust with mathematics.

After all, the reason we work so hard in cryptography is because a lack of trust. We wouldn't need encryption if Alice and Bob could be guaranteed that their communication, despite going through wireless and wired networks controlled and snooped upon by a plethora of entities, would be as reliable as if it has been hand delivered by a letter-carrier as reliable as [Patti Whitcomb](#), as opposed to the nosy Eve who might look in the messages, or the malicious Mallory, who might tamper with them. We wouldn't need zero knowledge proofs if Vladimir could simply say “trust me Barack, this is an authentic nuke”. We wouldn't need electronic signatures if we could trust that all software updates are designed to make our devices safer and not, to pick a random example, to turn our phones into surveillance devices.

Unfortunately, the world we live in is not as ideal, and we need these cryptographic tools. But what is the limit of what we can achieve? Are these examples of encryption, authentication, zero knowledge etc. isolated cases of good fortune, or are they special cases of a more general theory of what is possible in cryptography? It turns out that the latter is the case and there is in fact an extremely general formulation that (in some sense) captures all of the above and much more. This notion is called *multiparty secure computation*

or sometimes *secure function evaluation* and is the topic of this lecture. We will show (a relaxed version of) what I like to call “the fundamental theorem of cryptography”, namely that under natural computational conjectures (and in particular the LWE conjecture, as well as the RSA or Factoring assumptions) essentially every cryptographic task can be achieved. This theorem emerged from the 1980’s works of Yao, Goldreich-Micali-Wigderson, and many others. As we’ll see, like the “fundamental theorems” of other fields, this is a results that closes off the field but rather opens up many other questions. But before we can even state the result, we need to talk about how can we even define security in a general setting.

17.1 Ideal vs. Real Model Security.

The key notion is that cryptography aims to replace *trust*. Therefore, we imagine an *ideal world* where there is some universally trusted party (cryptographer Silvio Micali likes to denote by Jimmy Carter, but feel free to swap in your own favorite trustworthy personality) that communicates with all participants of the protocol or interaction, including potentially the adversary. We define security by stating that whatever the adversary can achieve in our real world, could have also been achieved in the ideal world.

For example, for obtaining secure communication, Alice will send her message to the trusted party, who will then convey it to Bob. The adversary learns nothing about the message’s contents, nor can she change them. In the zero knowledge application, to prove that there exists some secret x such that $f(x) = 1$ where $f(\cdot)$ is a public function, the prover Alice sends to the trusted party her secret input x , the trusted party then verifies that $f(x) = 1$ and simply sends to Bob the message “the statement is true”. It does not reveal to Bob anything about the secret x beyond that.

But this paradigm goes well beyond this. For example, **second price (or Vickrey) auctions** are known as a way to incentivize bidders to bid their true value. In these auctions, every potential buyer sends a sealed bid, and the item goes to the highest bidder, who only needs to pay the price of the second-highest bid. We could imagine a digital version, where buyers send encrypted versions of their bids. The auctioneer could announce who the winner is and what was the second largest bid, but could we really trust him to do so faithfully? Perhaps we would want an auction where even the auctioneer doesn’t learn anything about the bids beyond the identity of the winner and the value of the second highest bid? Wouldn’t it be great if there was a

trusted party that all bidders could share with their private values, and it would announce the results of the auction but nothing more than that? This could be useful not just in second price auctions but to implement many other mechanisms, especially if you are a [Danish sugar beet farmer](#).

There are other examples as well. Perhaps two hospitals might want to figure out if the same patient visited both, but do not want (or are legally not allowed) to share with one another the list of people that visited each one. A trusted party could get both lists and output only their intersection.

The list goes on and on. Maybe we want to aggregate securely information of the performance of [Estonian IT firms](#) or the financial health of [wall street banks](#). Almost every cryptographic task could become trivial if we just had access to a universally trusted party. But of course in the real world, we don't. This is what makes the notion of *secure multiparty computation* so exciting.

17.2 Formally defining secure multiparty computation

We now turn to formal definitions. As we discuss below, there are many variants of secure multiparty computation, and we pick one simple version below. A *k-party protocol* is a set of efficiently computable k prescribed interactive strategies for all k parties.¹ We assume the existence of an authenticated and private point to point channel between every pair of parties (this can be implemented using signatures and encryptions).² A *k party functionality* is a probabilistic process F mapping k inputs in $\{0,1\}^n$ into k outputs in $\{0,1\}^n$.³

17.2.1 First attempt: a slightly “too ideal” definition

Here is one attempt of a definition that is clean but a bit too strong, which nevertheless captures much of the spirit of secure multiparty computation:

Definition 17.1 — MPC without aborts. Let F be a k -party functionality. A *secure protocol for F* is a protocol for k parties satisfying that for every $T \subseteq [k]$ and every efficient adversary A , there exists an efficient “ideal adversary” (i.e., efficient interactive algorithm) S such that for every set of inputs $\{x_i\}_{i \in [k] \setminus T}$ the following two distributions are computationally indistinguishable:

¹ Note that here k is not a string which the secret key but the number of parties in the protocol.

² Protocols for $k > 2$ parties require also a *broadcast channel* but these can be [implemented](#) using the combination of authenticated channels and digital signatures.

³ Fixing the input and output sizes to n is done for notational simplicity and is without loss of generality. More generally, the inputs and outputs could have sizes up to polynomial in n and some inputs or output can also be empty. Also, note that one can define a more general notion of stateful functionalities, though it is not hard to reduce the task of building a protocol for stateful functionalities to building protocols for stateless ones.

- The tuple (y_1, \dots, y_k) of outputs of all the parties (both controlled and not controlled by the adversary) in an execution of the protocol where A controls the parties in T and the inputs of the parties not in T are given by $\{x_i\}_{i \in [k] \setminus T}$.
- The tuple (y_1, \dots, y_k) that is computed using the following process:
 1. We let $\{x_i\}_{i \in T}$ be chosen by S , and compute $(y'_1, \dots, y'_k) = F(x_1, \dots, x_k)$.
 2. For every $i \in [k]$, if $i \notin T$ (i.e., party i is “honest”) then $y_i = y'_i$ and otherwise, we let S choose y_i .

That is, the protocol is secure if whatever an adversary can gain by taking complete control over the set of parties in T , could have been gain by simply using this control to choose particular inputs $\{x_i\}_{i \in T}$, run the protocol honestly, and observe the outputs of the functionality.

Note that in particular if $T = \emptyset$ (and hence there is no adversary) then if the parties’ inputs are (x_1, \dots, x_k) then their outputs will equal $F(x_1, \dots, x_k)$.

17.2.2 Allowing for aborts

The definition above is a little too strong, in the following sense. Consider the case that $k = 2$ where there are two parties Alice (Party 1) and Bob (Party 2) that wish to compute some output $F(x_1, x_2)$. If Bob is controlled by the adversary then he clearly can simply abort the protocol and prevent Alice from computing y_1 . Thus, in this case in the actual execution of the protocol the output y_1 will be some error message (which we denote by \perp). But we did not allow this possibility for the idealized adversary S : if $1 \notin S$ then it must be the case that the output y_1 is equal to y'_1 for some $(y'_1, y'_2) = F(x_1, x_2)$.

This means that we would be able to distinguish between the output in the real and ideal setting.⁴ This motivates the following, slightly more messy definition, that allows for the ability of the adversary to abort the execution at any point in time:

Definition 17.2 — MPC with aborts. Let F be a k -party functionality. A *secure protocol for F* is a protocol for k parties satisfying that for every $T \subseteq [k]$ and every efficient adversary A , there exists an efficient “ideal adversary” (i.e., efficient interactive algorithm)

⁴ As a side note, we can avoid this issue if we have an honest majority of players - i.e. if $|T| < k/2$, but this of course makes no sense in the two party setting.)

S such that for every set of inputs $\{x_i\}_{i \in [k] \setminus T}$ the following two distributions are computationally indistinguishable:

- The tuple (y_1, \dots, y_k) of outputs of all the parties (both controlled and not controlled by the adversary) in an execution of the protocol where A controls the parties in T and the inputs of the parties not in T are given by $\{x_i\}_{i \in [k] \setminus T}$ we denote by $y_i = \top$ if the i^{th} party aborted the protocol.
- The tuple (y_1, \dots, y_k) that is computed using the following process:
 1. We let $\{x_i\}_{i \in T}$ be chosen by S , and compute $(y'_1, \dots, y'_k) = F(x_1, \dots, x_k)$.
 2. For $i = 1, \dots, k$ do the following: ask S if it wishes to abort at this stage, and if it doesn't then the i^{th} party learns y'_i . If the adversary did abort then we exit the loop at this stage and the parties $i + 1, \dots, k$ (regardless if they are honest or malicious) do not learn the corresponding outputs.
 3. Let k' be the last non-abort stage we reached above. For every $i \notin T$, if $i \leq k'$ then $y_i = y'_i$ and if $i > k'$ then $y_i = \perp$. We let the adversary S choose $\{y_i\}_{i \in T}$.

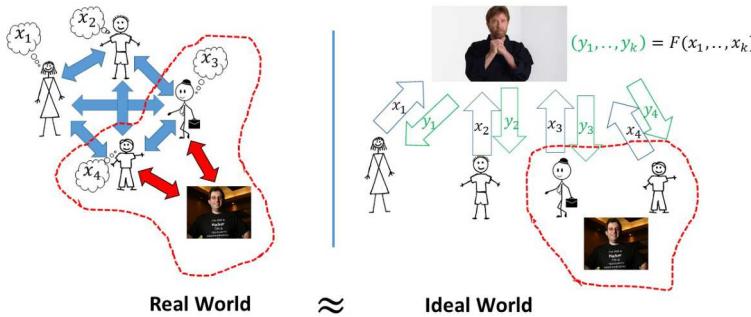


Figure 17.1: We define security of a protocol implementing F by stipulating that for every adversary A that control a subset of the parties, A 's view in an actual execution of the protocol would be indistinguishable from its view in an ideal setting where all the parties send their inputs to an idealized and perfectly trusted party, who then computes the outputs and sends it to each party.

Here are some good exercises to make sure you follow the definition:

- Let F be the two party functionality such that $F(H \parallel C, H')$ outputs $(1, 1)$ if the graph H equals the graph H' and C is a Hamiltonian

cycle and otherwise outputs $(0, 0)$. Prove that a protocol for computing F is a zero knowledge proof⁵ system for the language of Hamiltonicity.⁶

- Let F be the k -party functionality that on inputs $x_1, \dots, x_k \in \{0, 1\}$ outputs to all parties the majority value of the x_i 's. Then, in any protocol that securely computes F , for any adversary that controls less than half of the parties, if at least $n/2 + 1$ of the other parties' inputs equal 0, then the adversary will not be able to cause an honest party to output 1.



It is an excellent idea for you to pause here and try to work out at least informally these exercises.

Amazingly, we can obtain such a protocol for *every* functionality:

Theorem 17.3 — Fundamental theorem of cryptography. Under reasonable assumptions⁷ for every polynomial-time computable k -functionality F there is a polynomial-time protocol that computes it securely.

Theorem 17.3 was originally proven by Yao in 1982 for the special case of two party functionalities, and then proved for the general case by Goldreich, Micali, and Wigderson in 1987. As discussed below, many variants of this theorem have been shown, and this line of research is still ongoing.

17.2.3 Some comments:

There is in fact not a single theorem but rather many variants of this fundamental theorem obtained by great many people, depending on the different security properties desired, as well as the different cryptographic and setup assumptions. Some of the issues studied in the literature include the following:

- Fairness, guaranteed output delivery:** The definition above does not attempt to protect against “denial of service” attacks, in the sense that the adversary is allowed, even in the ideal case, to prevent the honest parties from receiving their outputs.

As mentioned above, without honest majority this is essential for similar reasons to the issue we discussed in [our lecture on bitcoin](#) why achieving consensus is hard if there isn't a honest majority. When there is an honest majority, we can achieve the property of

⁵ Actually, if we want to be pedantic, this is what's known as a zero knowledge *argument* system since soundness is only guaranteed against efficient provers. However, this distinction is not important in almost all applications.

⁶ Our treatment of the input graph H is an instance of a general case. While the definition of a functionality only talks about private inputs, it's very easy to include public inputs as well. If we want to include some public input Z we can simply have Z concatenated to all the private inputs (and the functionality check that they are all the same, otherwise outputting `error` or some similar result).

⁷ Originally this was shown under the assumption of trapdoor permutations (which can be derived from the Factoring or RSA conjectures) but it is known today under a variety of other assumptions, including in particular the LWE conjecture.

guaranteed output delivery, which offers protection against such “denial of service” attacks. Even when there is no guaranteed output delivery, we might want the property of *fairness*, whereas we guarantee that if the honest parties don’t get the output then neither does the adversary. There has been extensive study of fairness and there are protocols achieving variants on it under various computational and setup assumptions.

- **Network models:** The current definition assumes we have a set of k parties with known identities with pairwise secure (confidential and authenticated) channels between them. Other network models studies include broadcast channel, non-private networks, and even [no authentication](#)).
- **Setup assumptions:** The definition does not assume a trusted third party, but people have studied different setup assumptions including a public key infrastructure, common reference string, and more.
- **Adversarial power:** It turns out that under certain condition, it can be possible to obtain secure multiparty computation with respect to adversaries that have unbounded computational power (so called “information theoretic security”). People have also studies different variants of adversaries including “honst but curious” or “passive adversaries”, as well as “covert” adversaries that only deviate from the protocol if they won’t be caught. Other settings studied limit the adversary’s ability to control parties (e.g., honest majority, smaller fraction of parties or particular patterns of control, adaptive vs static corruption).
- **Concurrent compositions:** The definition displayed above are for *standalone execution* which is known not to automatically imply security with respect to *concurrent composition*, where many copies of the same protocol (or different protocols) could be executed simultaneously. This opens up all sorts of new attacks.⁸ See [Yehuda Lindell’s thesis](#) (or [this updated version](#)) for more. A very general notion known as “UC security” (which stands for “Universally Composable” or maybe “Ultimate Chuck”) has been proposed to achieve security in these settings, though at a price of additional setup assumptions, see [here](#) and [here](#).
- **Communication:** The communication cost for [Theorem 17.3](#) can be proportional to the size of the circuit that computes F . This can be a very steep cost, especially when computing over large amounts of data. It turns out that we can sometimes avoid this cost using fully homomorphic encryption or other techniques.

⁸ One example of the kind of issues that can arise is the “grandmasters attack” whereby someone with no knowledge of chess could play two grandmasters simultaneously, relaying their moves to one another and thereby guaranteeing a win in at least one of the games (or a draw in both).

- **Efficiency vs. generality:** While [Theorem 17.3](#) tells us that essentially every protocol problem can be solved *in principle*, its proof will almost never yield a protocol you actually want to run since it has enormous efficiency overhead. The issue of efficiency is the biggest reason why secure multiparty computation has so far not had a great many practical applications. However, researchers have been showing more efficient tailor-made protocols for particular problems of interest, and there has been steady progress in making those results more practical. See [the slides and videos from this workshop](#) for more.

Is multiparty secure computation the end of crypto? The notion of secure multiparty computation seems so strong that you might think that once it is achieved, aside from efficiency issues, there is nothing else to be done in cryptography. This is very far from the truth. Multiparty secure computation do give a way to solve a great many problems in the setting where we have arbitrary rounds of interactions and unbounded communication, but this is far from being always the case. As we mentioned before, interaction can sometimes make a *qualitative* difference (when Alice and Bob are separated by time rather than space). As we've seen in the discussion of fully homomorphic encryption, there are also other properties, such as compact communication, which are not implied by multiparty secure computation but can make all the difference in contexts such as cloud computing. That said, multiparty secure computation is an extremely general paradigm that does apply to many cryptographic problems.

Further reading: The [survey of Lindell and Pinkas](#) gives a good overview of the different variants and security properties considered in the literature, see also Section 7 in [this survey of Goldreich](#). Chapter 6 in [Pass and Shelat's notes](#) is also a good source.

17.3 Example: Second price auction using bitcoin

Suppose we have the following setup: an auctioneer wants to sell some item and run a second-price auction, where each party submits a sealed bid, and the highest bidder gets the item for the price of the second highest bid. However, as mentioned above, the bidders do not want the auctioneer to learn what their bid was, and in general nothing else other than the identity of the highest bidder and the value of the second highest bid. Moreover, we might want the payment is via an electronic currency such as bitcoin, so that the auctioneer not only gets the information about the winning bid but an actual self-certifying transaction they can use to get the payment.

Here is how we could obtain such a protocol using secure multiparty computation:

- We have k parties where the first party is the auctioneer and parties $2, \dots, k$ are bidders. Let's assume for simplicity that each party i has a public key v_i that is associated with some bitcoin account.⁹ We treat all these keys as the public input.
- The private input of bidder i is the value x_i that it wants to bid as well as the secret key s_i that corresponds to their public key.
- The functionality only provides an output to the auctioneer, which would be the identity i of the winning bidder as well as a valid signature on this bidder transferring x bitcoins to the key v_1 of the auctioneer, where x is the value of the second largest valid bid (i.e., x equals to the second largest x_j such that s_j is indeed the private key corresponding to v_j .)

It's worthwhile to think about what a secure protocol for this functionality accomplishes. For example:

- The fact that in the ideal model the adversary needs to choose its queries independently means that the adversary cannot get any information about the honest parties' bids before deciding on its bid.
- Despite all parties using their signing keys as inputs to the protocol, we are guaranteed that no one will learn anything about another party's signing key except the single signature that will be produced.
- Note that if i is the highest bidder and j is the second highest, then at the end of the protocol we get a valid signature using s_i on a transaction transferring x_j bitcoins to v_1 , despite i not knowing the value x_j (and in fact never learning the identity of j .) Nonetheless, i is guaranteed that the signature produced will be on an amount not larger than its own bid and an amount that one of the other bidders actually bid for.

I find the ability to obtain such strong notions of security pretty remarkable. This demonstrates the tremendous power of obtaining protocols for general functionalities.

17.3.1 Another example: distributed and threshold cryptography

It sometimes makes sense to use multiparty secure computation for *cryptographic computations* as well. For example, there might be

⁹ As we discussed before, bitcoin doesn't have the notion of accounts but rather what we mean by that for each one of the public keys, the public ledger contains a sufficiently large amount of bitcoins that have been transferred to these keys (in the sense that whomever can sign w.r.t. these keys can transfer the corresponding coins).

several reasons why we would want to “split” a secret key between several parties, so no party knows it completely.

- Some proposals for *key escrow* (giving government or other entity an option for decrypting communication) suggested to split a cryptographic key between several agencies or institutions (say the FBI, the courts, etc..) so that they must collaborate in order to decrypt communication, thus hopefully preventing unlawful access.
- On the other side, a company might wish to split its own key between several servers residing in different countries, to ensure not one of them is completely under one jurisdiction. Or it might do such splitting for technical reasons, so that if there is a break in into a single site, the key is not compromised.

There are several other such examples. One problem with this approach is that splitting a cryptographic key is not the same as cutting a 100 dollar bill in half. If you simply give half of the bits to each party, you could significantly harm security. (For example, it is possible to recover the full RSA key **from only 27% of its bits**).

Here is a better approach, known as **secret sharing**: To securely share a string $s \in \{0,1\}^n$ among k parties so that any $k - 1$ of them have no information about it, we choose s_1, \dots, s_{k-1} at random in $\{0,1\}^n$ and let $s_k = s \oplus s_1 \oplus \dots \oplus s_{k-1}$ (\oplus as usual denotes the XOR operation), and give party i the string s_i , which is known as the i^{th} *share* of s . Note that $s = s_1 \oplus \dots \oplus s_t$ and so given all k pieces we can reconstruct the key. Clearly the first $k - 1$ parties did not receive any information about s (since their shares were generated independent of s), but the following not-too-hard claim shows that this holds for *every* set of $k - 1$ parties:

Lemma 17.4 For every $s \in \{0,1\}^n$, and set $T \subseteq [k]$ of size $k - 1$, we get exactly the same distribution over (s_1, \dots, s_k) as above if we choose s_i for $i \in T$ at random and set $s_t = s \oplus_{i \in T} s_i$ where $t = [k] \setminus T$.

We leave the proof of [Lemma 17.4](#) as an exercise.

Secret sharing solves the problem of protecting the key “at rest” but if we actually want to *use* the secret key in order to sign or decrypt some message, then it seems we need to collect all the pieces together into one place, which is exactly what we wanted to avoid doing. This is where multiparty secure computation comes into play, we can define a functionality F taking public input m and secret inputs s_1, \dots, s_k and producing a signature or decryption of m . In fact, we can go beyond that and even have the parties sign or decrypt a

message without them knowing what this message is, except that it satisfies some conditions.

Moreover, secret sharing can be generalized so that a threshold other than k is necessary and sufficient to reconstruct the secret (and people have also studied more complicated access patterns). Similarly multiparty secure computation can be used to achieve distributed cryptography with finer access control mechanisms.

17.4 Proving the fundamental theorem:

We will complete the proof of (a relaxed version of) the fundamental theorem over this lecture and the next one. The proof consists of two phases:

1. A protocol for the “honest but curious” case using fully homomorphic encryption.
2. A reduction of the general case into the “honest but curious” case where the adversary follows the protocol precisely but merely attempts to learn some information on top of the output that it is “entitled to” learn. (This reduction is based on zero knowledge proofs and is due to Goldreich, Micali and Wigderson)

We note that while fully homomorphic encryption yields a conceptually simple approach for the second step, it is not currently the most efficient approach, and rather most practical implementations are based on the technique known as “Yao’s Garbled Circuits” (see [this book](#) or [this paper](#) or [this survey](#)) which in turn is based a notion known as **oblivious transfer** which can be thought of as a “baby private information retrieval” (though it preceded the latter notion).

We will focus on the case of *two parties*. The same ideas extend to $k > 2$ parties but with some additional complications.

17.5 Malicious to honest but curious reduction

We start from the second stage. Giving a reduction transforming a protocol in the “honest but curious” setting into a protocol secure in the malicious setting. Note that a priori, it is not obvious at all that such a reduction should exist. In the “honest but curious” setting we assume the adversary follows the protocol to the letter. Thus a protocol where Alice gives away all her secrets to Bob if he merely **asks her to do so politely** can be secure in the “honest but curious”

setting if Bob's instructions are not to ask. More seriously, it could very well be that Bob has an ability to deviate from the protocol in subtle ways that would be completely undetectable but allow him to learn Alice's secrets. Any transformation of the protocol to obtain security in the malicious setting will need to rule out such deviations.

The main idea is the following: we do the compilation one party at a time - we first transform the protocol so that it will remain secure even if Alice tries to cheat, and then transform it so it will remain secure even if Bob tries to cheat. Let's focus on Alice. Let's imagine (without loss of generality) that Alice and Bob alternate sending messages in the protocol with Alice going first, and so Alice sends the odd messages and Bob sends the even ones. Lets denote by m_i the message sent in the i^{th} round of the protocol. Alice's instructions can be thought of as a sequence of functions f_1, f_3, \dots, f_t (where t is the last round in which Alice speaks) where each f_i is an efficiently computable function mapping Alice's secret input x_1 , (possibly) her random coins r_1 , and the transcript of the previous messages m_1, \dots, m_{i-1} to the next message m_i . The functions $\{f_i\}$ are publicly known and part of the protocol's instructions. The only thing that Bob doesn't know is x_1 and r_1 . So, our idea would be to change the protocol so that after Alice sends the message i , she *proves* to Bob that it was indeed computed correctly using f_i . If x_1 and r_1 weren't secret, Alice could simply send those to Bob so he can verify the computation on his own. But because they are (and the security of the protocol could depend on that) we instead use a *zero knowledge proof*.

Let's assume for starters that Alice's strategy is *deterministic* (and so there is no random tape r_1). A first attempt to ensure she can't use a malicious strategy would be for Alice to follow the message m_i with a zero knowledge proof that there exists some x_1 such that $m_i = f(x_1, m_1, \dots, m_{i-1})$. However, this will actually not be secure - it is worth while at this point for you to pause and think if you can understand the problem with this solution.



Really, please stop and think why this will not be secure.

P

Did you stop and think?

The problem is that at every step Alice proves that there exists *some* input x_1 that can explain her message but she doesn't prove that it's *the same input for all messages*. If Alice was being truly honest, she should have picked her input once and use it throughout the protocol, and she could not compute the first message according to the input x_1 and then the third message according to some input $x'_1 \neq x_1$. Of course we can't have Alice reveal the input, as this would violate security. The solution is for Alice to *commit* in advance to the input. We have seen commitments before, but let us now formally define the notion:

Definition 17.5 — Commitment scheme. A *commitment scheme* for strings of length ℓ is a two party protocol between the *sender* and *receiver* satisfying the following:

- **Hiding (sender's security):** For every two sender inputs $x, x' \in \{0,1\}^\ell$, and no matter what efficient strategy the receiver uses, it cannot distinguish between the interaction with the sender when the latter uses x as opposed to when it uses x' .
- **Binding (reciever's security):** No matter what (efficient or non efficient) strategy the sender uses, if the receiver follows the protocol then with probability $1 - negl(n)$, there will exist at most a single string $x \in \{0,1\}^\ell$ such that the transcript is consistent with the input x and some sender randomness r .

That is, a commitment is the digital analog to placing a message in a sealed envelope to be opened at a later time. To commit to a message x the sender and receiver interact according to the protocol, and to *open* the commitment the sender simply sends x as well as the random coins it used during the commitment phase. The variant we defined above is known as *computationally hiding and statistically binding*, since the sender's security is only guaranteed against efficient receivers while the binding property is guaranteed against all senders. There are also statistically hiding and computationally binding commitments, though it can be shown that we need to restrict to efficient strategies for at least one of the parties.

We have already seen a commitment scheme before (due to Naor): the receiver sends a random $z \leftarrow_R \{0,1\}^{3n}$ and the sender commits to a bit b by choosing a random $s \in \{0,1\}^n$ and sending $y = PRG(s) + bz(\bmod 2)$ where $PRG : \{0,1\}^n \rightarrow \{0,1\}^{3n}$ is a

pseudorandom generator. It's a good exercise to verify that it satisfies the above definitions. By running this protocol ℓ times in parallel we can commit to a string of any polynomial length.

We can now describe the transformation ensuring the protocol is secure against a malicious Alice in full, for the case that the original strategy of Alice is *deterministic* (and hence uses no random coins)

- Initially Alice and Bob engage in a commitment scheme where Alice commits to her input x_1 . Let τ be the transcript of this commitment phase and r_{com} be the randomness Alice used during it.¹⁰
- For $i = 1, 2, \dots$:
 - If i is even then Bob sends m_i to Alice
 - If i is odd then Alice sends m_i to Bob and then they engage in a zero knowledge proof that there exists x_1, r_{com} such that (1) x_1, r_{com} is consistent with τ , and (2) $m_i = f(x_1, m_1, \dots, m_{i-1})$. The proof is repeated a sufficient number of times to ensure that if the statement is false then Bob rejects with $1 - negl(n)$ probability.
 - If the proof is rejected then Bob aborts the protocol.

We will not prove security but will only sketch it here, see [Section 7.3.2 in Goldreich's survey](#) for a more detailed proof:

- To argue that we maintain security for *Alice* we use the zero knowledge property: we claim that Bob could not learn anything from the zero knowledge proofs precisely because he could have simulated them by himself. We also use the hiding property of the commitment scheme. To prove security formally we need to show that whatever Bob learns in the modified protocol, he could have learned in the original protocol as well. We do this by *simulating* Bob by replacing the commitment scheme with commitment to some random junk instead of x_1 and the zero knowledge proofs with their simulated version. The proof of security requires a hybrid argument, and is again a good exercise to try to do it on your own.
- To argue that we maintain security for *Bob* we use the binding property of the commitment scheme as well as the soundness property of the zero knowledge system. Once again for the formal proof we need to show that we could transform any potentially malicious strategy for Alice in the modified protocol into an “hon-

¹⁰ Note that even though we assumed that in the original honest-but-curious protocol Alice used a deterministic strategy, we will transform the protocol into one in which Alice uses a randomized strategy in both the commitment and zero knowledge phases.

est but curious” strategy in the original protocol (also allowing Alice the ability to abort the protocol). It turns out that to do so, it is not enough that the zero knowledge system is sound but we need a stronger property known as a *proof of knowledge*. We will not define it formally, but roughly speaking it means we can transform any prover strategy that convinces the verifier that a statement is true with non-negligible probability into an algorithm that outputs the underlying secret (i.e., x_1 and r_{com} in our case). This is crucial in order to transform Alice’s potentially malicious strategy into an honest but curious strategy.

We can repeat this transformation for Bob (or Charlie, David, etc.. in the $k > 2$ party case) to transform a protocol secure in the honest but curious setting into a protocol secure (allowing for aborts) in the malicious setting.

17.5.1 Handling probabilistic strategies:

So far we assumed that the original strategy of Alice in the honest but curious is deterministic but of course we need to consider probabilistic strategies as well. One approach could be to simply think of Alice’s random tape r as part of her secret input x_1 . However, while in the honest but curious setting Alice is still entitled to freely choose her own input x_1 , she is not entitled to choose the random tape as she wishes but is supposed to follow the instructions of the protocol and choose it uniformly at random. Hence we need to use a *coin tossing protocol* to choose the randomness, or more accurately what’s known as a “coin tossing in the well” protocol where Alice and Bob engage in a coin tossing protocol at the end of which they generate some random coins r that only Alice knows but Bob is still guaranteed that they are random. Such a protocol can actually be achieved very simply. Suppose we want to generate m coins:

- Alice selects $r' \leftarrow_R \{0,1\}^m$ at random and engages in a *commitment protocol* to commit to r' .
- Bob selects $r'' \leftarrow_R \{0,1\}^m$ and sends it to Alice in the clear.
- The result of the coin tossing protocol will be the string $r = r' \oplus r''$.

Note that Alice knows r . Bob doesn’t know r but because he chose r'' after Alice committed to r' he knows that it must be fully random regardless of Alice’s choice of r' . It can be shown that if we use this coin tossing protocol at the beginning and then modify the zero knowledge proofs to show that $m_i = f(x_1, r_1, m_1, \dots, m_{i-1})$ where r is the string that is consistent with the transcript of the coin tossing

protocol, then we get a general transformation of an honest but curious adversary into the malicious setting.

The notion of multiparty secure computation - defining it and achieving it - is quite subtle and I do urge you to read some of the other references listed above as well. In particular, the slides and videos from the [Bar Ilan winter school on secure computation and efficiency](#), as well as the ones from the [winter school on advances in practical multiparty computation](#) are great sources for this and related materials.

Multiparty secure computation: Construction using Fully Homomorphic Encryption

In the last lecture we saw the definition of secure multiparty computation, as well as the compiler reducing the task of achieving security in the general (malicious) setting to the passive (honest-but-curious) setting. In this lecture we will see how using fully homomorphic encryption we can achieve security in the honest-but-curious setting.¹ We focus on the two party case, and so prove the following theorem:

Theorem 18.1 — Two party honest-but-curious MPC. Assuming the LWE conjecture, for every two party functionality F there is a protocol computing F in the honest but curious model.

Before proving the theorem it might be worthwhile to recall what is actually the definition of secure multiparty computation, when specialized for the $k = 2$ and honest but curious case. The definition significantly simplifies here since we don't have to deal with the possibility of aborts.

Definition 18.2 — Two party honest-but-curious secure computation.

Let F be (possibly probabilistic) map of $\{0,1\}^n \times \{0,1\}^n$ to $\{0,1\}^n \times \{0,1\}^n$. A *secure protocol* for F is a two party protocol such for every party $t \in \{1,2\}$, there exists an efficient “ideal adversary” (i.e., efficient interactive algorithm) S such that for every pair of inputs (x_1, x_2) the following two distributions are computationally indistinguishable:

- The tuple (y_1, y_2, v) obtained by running the protocol on inputs x_1, x_2 , and letting y_1, y_2 be the outputs of the two parties and

¹ This is by no means the only way to get multiparty secure computation. In fact, multiparty secure computation was known well before FHE was discovered. One common construction for achieving this uses a technique known as *Yao's Garbled Circuit*.

v be the *view* (all internal randomness, inputs, and messages received) of party t .

- The tuple (y_1, y_2, v) that is computed by letting $(y_1, y_2) = F(x_1, x_2)$ and $v = S(x_t, y_t)$.

That is, S , which only gets the input x_t and output y_t , can simulate all the information that an honest-but-curious adversary controlling party t will view.

18.1 Constructing 2 party honest but curious computation from fully homomorphic encryption

Let F be a two party functionality. Lets start with the case that F is *deterministic* and that only Alice receives an output. We'll later show an easy reduction from the general case to this one. Here is a suggested protocol for Alice and Bob to run on inputs x, y respectively so that Alice will learn $F(x, y)$ but nothing more about y , and Bob will learn nothing about x that he didn't know before.

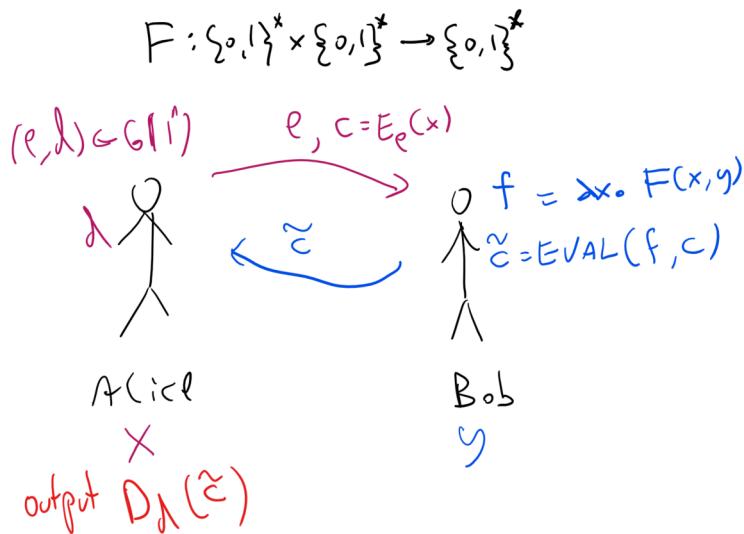


Figure 18.1: An honest but curious protocol for two party computation using a fully homomorphic encryption scheme with circuit privacy.

Protocol 2PC: (See Fig. 18.1)

- Assumptions:** (G, E, D, EVAL) is a fully homomorphic encryption scheme.
- Inputs:** Alice's input is $x \in \{0,1\}^n$ and Bob's in-

put is $y \in \{0,1\}^n$. The goal is for Alice to learn only $F(x,y)$ and Bob to learn nothing.

- **Alice->Bob:** Alice generates $(e,d) \leftarrow_R G(1^n)$ and sends e and $c = E_e(x)$.
- **Bob->Alice:** Bob computes define f to be the function $f(x) = F(x,y)$ and sends $c' = EVAL(f,c)$ to Alice.
- **Alice's output:** Alice computes $z = D_d(c')$.

First, note that if Alice and Bob both follow the protocol, then indeed at the end of the protocol Alice will compute $F(x,y)$. We now claim that Bob does not learn anything about Alice's input:

Claim B: For every x,y , there exists a standalone algorithm S such that $S(y)$ is indistinguishable from Bob's view when interacting with Alice and their corresponding inputs are (x,y) .

Proof: Bob only receives a single message in this protocol of the form (e,c) where e is a public key and $c = E_e(x)$. The simulator S will generate $(e,d) \leftarrow_R G(1^n)$ and compute (e,c) where $c = E_e(0^n)$. (As usual 0^n denotes the length n string consisting of all zeroes.) No matter what x is, the output of S is indistinguishable from the message Bob receives by the security of the encryption scheme. QED

(In fact, Claim B holds even against a *malicious* strategy of Bob- can you see why?)

We would now hope that we can prove the same regarding Alice's security. That is prove the following:

Claim A: For every x,y , there exists a standalone algorithm S such that $S(y)$ is indistinguishable from Alice's view when interacting with Bob and their corresponding inputs are (x,y) .



At this point, you might want to try to see if you can prove Claim A on your own. If you're having difficulties proving it, try to think whether it's even true.

So, it turns out that Claim A is *not* generically true. The reason is the following: the definition of fully homomorphic encryption only requires that $EVAL(f, E(x))$ decrypts to $f(x)$ but it does *not* require that it hides the contents of f . For example, for every FHE, if we modify $EVAL(f, c)$ to append to the ciphertext the first 100 bits of the description of f (and have the decryption algorithm ignore this extra information) then this would still be a secure FHE.² Now we didn't exactly specify how we describe the function $f(x)$ defined as $x \mapsto F(x, y)$ but there are clearly representations in which the first 100 bits of the description would reveal the first few bits of the hardwired constant y , hence meaning that Alice will learn those bits from Bob's message.

Thus we need to get a stronger property, known as *circuit privacy*. This is a property that's useful in other contexts where we use FHE. Let us now define it:

Definition 18.3 — Perfect circuit privacy. Let $\mathcal{E} = (G, E, D, EVAL)$ be an FHE. We say that \mathcal{E} satisfies *perfect circuit privacy* if for every (e, d) output by $G(1^n)$ and every function $f : \{0, 1\}^\ell \rightarrow \{0, 1\}$ of $\text{poly}(n)$ description size, and every ciphertexts c_1, \dots, c_ℓ and $x_1, \dots, x_\ell \in \{0, 1\}$ such that c_i is output by $E_e(x_i)$, the distribution of $EVAL_e(f, c_1, \dots, c_\ell)$ is identical to the distribution of $E_e(f(x))$. That is, for every $z \in \{0, 1\}^*$, the probability that $EVAL_e(f, c_1, \dots, c_\ell) = z$ is the same as the probability that $E_e(f(x)) = z$. We stress that these probabilities are taken only over the coins of the algorithms $EVAL$ and E .

Perfect circuit privacy is a strong property, that also automatically implies that $D_d(EVAL(f, E_e(x_1), \dots, E_e(x_\ell))) = f(x)$ (can you see why?). In particular, once you understand the definition, the following lemma is a fairly straightforward exercise.

Lemma 18.4 If $(G, E, D, EVAL)$ satisfies perfect circuit privacy then if $(e, d) = G(1^n)$ then for every two functions $f, f' : \{0, 1\}^\ell \rightarrow \{0, 1\}$ of $\text{poly}(n)$ description size and every $x \in \{0, 1\}^\ell$ such that $f(x) = f'(x)$, and every algorithm A ,

$$|\mathbb{P}[A(d, EVAL(f, E_e(x_1), \dots, E_e(x_\ell))) = 1] - \mathbb{P}[A(d, EVAL(f', E_e(x_1), \dots, E_e(x_\ell))) = 1]| < \text{negl}(n). \quad (18.1)$$



Please stop here and try to prove Lemma 18.4

The algorithm A above gets the *secret key* as input, but still cannot distinguish whether the $EVAL$ algorithm used f or f' . In fact, the

² It's true that strictly speaking, we allowed $EVAL$'s output to have length at most n , while this would make the output be $n + 100$, but this is just a technicality that can be easily bypassed, for example by having a new scheme that on security parameter n runs the original scheme with parameter $n/2$ (and hence will have a lot of "room" to pad the output of $EVAL$ with extra bits).

expression on the lefthand side of [Eq. \(18.1\)](#) is equal to zero when the scheme satisfies perfect circuit privacy.

However, for our applications bounding it by a negligible function is enough. Hence, we can use the relaxed notion of “imperfect” circuit privacy, defined as follows:

Definition 18.5 — Statistical circuit privacy. Let $\mathcal{E} = (G, E, D, EVAL)$ be an FHE. We say that \mathcal{E} satisfies *statistical circuit privacy* if for every (e, d) output by $G(1^n)$ and every function $f : \{0,1\}^\ell \rightarrow \{0,1\}$ of $poly(n)$ description size, and every ciphertexts c_1, \dots, c_ℓ and $x_1, \dots, x_\ell \in \{0,1\}$ such that c_i is output by $E_e(x_i)$, the distribution of $EVAL_e(f, c_1, \dots, c_\ell)$ is equal up to $negl(n)$ total variation distance to the distribution of $E_e(f(x))$.

That is,

$$\sum_{z \in \{0,1\}^*} |\mathbb{P}[EVAL_e(f, c_1, \dots, c_\ell) = z] - \mathbb{P}[E_e(f(x)) = z]| < negl(n) \quad (18.2)$$

where once again, these probabilities are taken only over the coins of the algorithms *EVAL* and *E*.

If you find [Definition 18.5](#) hard to parse, the most important points you need to remember about it are the following:

- Statistical circuit privacy is as good as perfect circuit privacy for all applications, and so you can imagine the latter notion when using it.
- Statistical circuit privacy can easier to achieve in constructions.

(The third point, which goes without saying, is that you can always ask clarifying questions in class, Piazza, sections, or office hours...)

Intuitively, circuit privacy corresponds to what we need in the above protocol to protect Bob’s security and ensure that Alice doesn’t get any information about his input that she shouldn’t have from the output of *EVAL*, but before working this out, let us see how we can construct fully homomorphic encryption schemes satisfying this property.

18.2 Achieving circuit privacy in a fully homomorphic encryption

We now discuss how we can modify our fully homomorphic encryption schemes to achieve the notion of circuit privacy. In the scheme we saw, the encryption of a bit b , whether obtained through the encryption algorithm or $EVAL$, always had the form of a matrix C over \mathbb{Z}_q (for $q = 2^{\sqrt{n}}$) where $Cv = bv + e$ for some vector e that is “small” (e.g., for every i , $|e_i| < n^{polylog(n)} \ll q = 2^{\sqrt{n}}$). However, the $EVAL$ algorithm was *deterministic* and hence this vector e is a function of whatever function f we are evaluating and someone that knows the secret key v could recover e and then obtain from it some information about f . We want to make $EVAL$ probabilistic and lose that information, and we use the following approach

To kill a signal, drown it in lots of noise

That is, if we manage to add some additional random noise e' that has magnitude much larger than e , then it would essentially “erase” any structure e had. More formally, we will use the following lemma:

Lemma 18.6 Let $a \in \mathbb{Z}_q$ and $T \in \mathbb{N}$ be such that $aT < q/2$. If we let X be the distribution obtained by taking $x \pmod{q}$ for an integer x chosen at random in $[-T, +T]$ and let X' be the distribution obtained by taking $a + x \pmod{q}$ for x chosen in the same way, then

$$\sum_{y \in \mathbb{Z}_q} |\mathbb{P}[X = y] - \mathbb{P}[X' = y]| < |a|/T \quad (18.3)$$

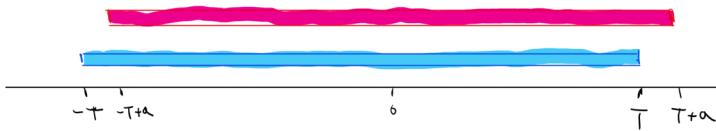


Figure 18.2: If $a \ll T$ then the uniform distribution over the interval $[-T, +T]$ is statistically close to the uniform distribution over the interval $[-T+a, +T+a]$, since the statistical distance is proportional to the event (which happens with probability a/T) that a random sample from one distribution falls inside the symmetric difference of the two intervals.

Proof. This has a simple “proof by picture”: consider the intervals $[-T, +T]$ and $[-T+a, +T+a]$ on the number line (see Fig. 18.2). Note that the symmetric difference of these two intervals is only about a a/T fraction of their union. More formally, X is the uniform distribution over the $2T+1$ numbers in the interval $[-T, +T]$ while X' is the uniform distribution over the shifted version of this interval

$[-T + a, +T + a]$. There are exactly $2|a|$ numbers which get probability zero under one of those distributions and probability $(2T + 1)^{-1} < (2T)^{-1}$ under the other. ■

We will also use the following lemma:

Lemma 18.7 If two distributions over numbers X and X' satisfy $\Delta(X, X') = \sum_{y \in \mathbb{Z}} |\mathbb{P}[X = y] - \mathbb{P}[Y = y]| < \delta$ then the distributions X^m and X'^m over m dimensional vectors where every entry is sampled independently from X or X' respectively satisfy $\Delta(X^m, X'^m) \leq m\delta$.



We omit the proof of Lemma 18.7 and leave it as an exercise to prove it using the hybrid argument. We will actually only use Lemma 18.7 for distributions above; you can obtain intuition for it by considering the $m = 2$ case where we compare the rectangles of the forms $[-T, +T] \times [-T, +T]$ and $[-T + a, +T + a] \times [-T + b, +T + b]$. You can see that their union has size roughly $4T^2$ while their symmetric difference has size roughly $2T \cdot 2a + 2T \cdot 2b$, and so if $|a|, |b| \leq \delta T$ then the symmetric difference is roughly a 2δ fraction of the union.

We will not provide the full details, but together these lemmas show that *EVAL* can use bootstrapping to reduce the magnitude of the noise to roughly $2^{n^{0.1}}$ and then add an additional random noise of roughly, say, $2^{n^{0.2}}$ which would make it statistically indistinguishable from the actual encryption. Here are some hints on how to make this work: the idea is that in order to “re-randomize” a ciphertext C we need a very noisy encryption of zero and add it to C . The normal encryption will use noise of magnitude $2^{n^{0.2}}$ but we will provide an encryption of the secret key with smaller magnitude $2^{n^{0.1}/\text{polylog}(n)}$ so we can use bootstrapping to reduce the noise. The main idea that allows to add noise is that at the end of the day, our scheme boils down to LWE instances that have the form (c, σ) where c is a random vector in \mathbb{Z}_q^{n-1} and $\sigma = \langle c, s \rangle + a$ where $a \in [-\eta, +\eta]$ is a small noise addition. If we take any such input and add to σ some $a' \in [-\eta', +\eta']$ then we create the effect of completely re-randomizing the noise. However, completely analyzing this requires non-trivial amount of care and work.

18.3 Bottom line: A two party honest but curious two party secure computation protocol

We can now prove the following theorem:

Theorem 18.8 — Two party. If $(G, E, D, EVAL)$ is a statistically circuit private fully homomorphic encryption then Protocol 2PC is a secure two party computation protocol with respect to honest but curious adversaries.

Quantum computing and cryptography I

"I think I can safely say that nobody understands quantum mechanics." , Richard Feynman, 1965

"The only difference between a probabilistic classical world and the equations of the quantum world is that somehow or other it appears as if the probabilities would have to go negative", Richard Feynman, 1982

For much of the history of mankind, people believed that the ultimate “theory of everything” would be of the “billiard ball” type. That is, at the end of the day, everything is composed of some elementary particles and adjacent particles interact with one another according to some well specified laws. The types of particles and laws might differ, but not the general shape of the theory. Note that this in particular means that a system of N particles can be simulated by a computer with $\text{poly}(N)$ memory and time.

Alas, in the beginning of the 20th century, several experimental results were calling into question the “billiard ball” theory of the world. One such experiment is the famous “double slit” experiment. Suppose we shoot an electron at a wall that has a single slit at position i and put somewhere behind this slit a detector. If we let p_i be the probability that the electron goes through the slit and let q_i be the probability that conditioned on this event, the electron hits this detector, then the fraction of times the electron hits our detector should be (and indeed is) $\alpha = p_i q_i$. Similarly, if we close this slit and open a second slit at position j then the new fraction of times the electron hits our detector will be $\beta = p_j q_j$. Now if we open both slits then it seems that the fraction should be $\alpha + \beta$ and in particular, “obviously” the probability that the electron hits our detector should only

increase if we open a second slit. However, this is not what actually happens when we run this experiment. It can be that the detector is hit a *smaller* number of times when two slits are open than when only a single one hits. It's almost as if the electron checks whether two slits are open, and if they are, it changes the path it takes. If we try to "catch the electron in the act" and place a detector right next to each slit so we can count which electron went through which slit then something even more bizarre happened. The mere fact that we *measured* the electron path changes the actual path it takes, and now this "destructive interference" pattern is gone and the detector will be hit $\alpha + \beta$ fraction of the time.

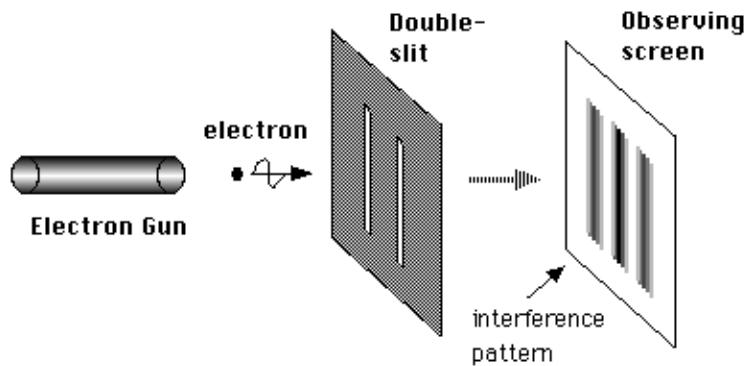


Figure 19.1: The setup of the double slit experiment

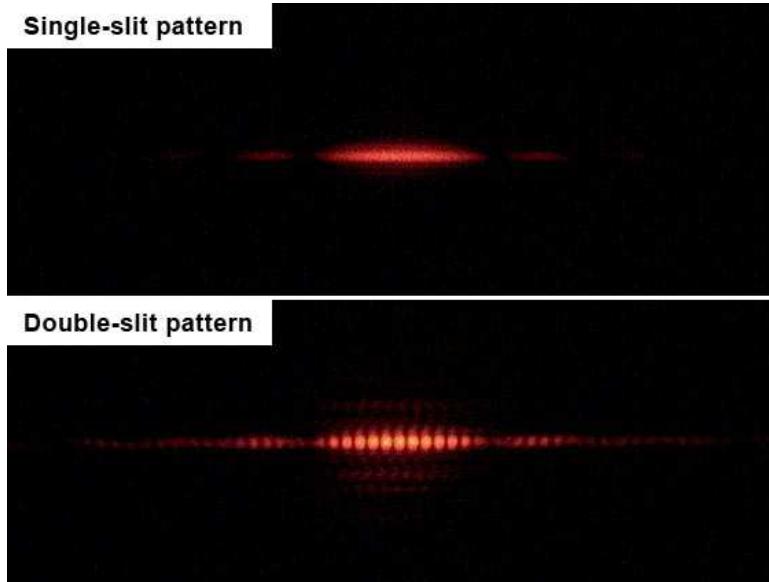


Figure 19.2: In the double slit experiment, opening two slits can actually cause some positions to receive *fewer* electrons than before.

Quantum mechanics is a mathematical theory that allows us to calculate and predict the results of this and many other examples. If you think of quantum as an explanation as to what "really" goes

on in the world, it can be rather confusing. However, if you simply “shut up and calculate” then it works amazingly well at predicting the results of a great many experiments.

In the double slit experiment, quantum mechanics still allows to compute numbers α and β that denote “probabilities” that the first and second electrons hit the detector. The only difference that in quantum mechanics these probabilities might be *negative* numbers. However, probabilities can only be negative when no one is looking at them. When we actually measure what happened to the detector, we make the probabilities positive by *squaring* them. So, if only the first slit is open, the detector will be hit α^2 fraction of the time. If only the second slit is open, the detector will be hit β^2 fraction of the time. And if both slits are open, the detector will be hit $(\alpha + \beta)^2$ fraction of the time. Note that it can well be that $(\alpha + \beta)^2 < \alpha^2 + \beta^2$ and so this calculation explains why the number of times a detector is hit when two slits are open might be *smaller* than the number of times it is hit when either slit is open. If you haven’t seen it before, it may seem like complete nonsense and at this point I’ll have to politely point you back to the part where I said we should not question quantum mechanics but simply “shut up and calculate”.¹

Some of the counterintuitive properties that arise from these negative probabilities include:

- **Interference** - As we see here, probabilities can “cancel each other out”.
- **Measurement** - The idea that probabilities are negative as long as “no one is looking” and “collapse” to positive probabilities when they are *measured* is deeply disturbing. Indeed, people have shown that it can yield to various strange outcomes such as “spooky actions at a distance”, where we can measure an object at one place and instantaneously (faster than the speed of light) cause a difference in the results of a measurements in a place far removed. Unfortunately (or fortunately?) these strange outcomes have been confirmed experimentally.
- **Entanglement** - The notion that two parts of the system could be connected in this weird way where measuring one will affect the other is known as *quantum entanglement*.

Again, as counter-intuitive as these concepts are, they have been experimentally confirmed, so we just have to live with them.

¹ If you *have* seen quantum mechanics before, I should warn that I am making here many simplifications. In particular in quantum mechanics the “probabilities” can actually be *complex* numbers, though one gets most of the qualitative understanding by considering them as potentially negative real numbers. I will also be focusing throughout this presentation on so called “pure” quantum states, and ignore the fact that generally the states of a quantum subsystem are *mixed* states that are a convex combination of pure states and can be described by a so called *density matrix*. This issue does not arise as much in quantum algorithms precisely because the goal is for a quantum computer to be an isolated system that can evolve to continue to be in a pure state; in real world quantum computers however there will be interference from the outside world that causes the state to become mixed and increase its so called “von Neumann entropy” - fighting this interference and the second law of thermodynamics is much of what the challenge of building quantum computers is all about. More generally, this lecture is not meant to be a complete or accurate description of quantum mechanics, quantum information theory, or quantum computing, but rather just give a sense of the main points that are different about it from classical computing and how they relate to cryptography.

19.0.1 Quantum computing and computation - an executive summary.

One of the strange aspects of the quantum-mechanical picture of the world is that unlike in the billiard ball example, there is no obvious algorithm to simulate the evolution of n particles over t time periods in $\text{poly}(n, t)$ steps. In fact, the natural way to simulate n quantum particles will require a number of steps that is *exponential* in n . This is a huge headache for scientists that actually need to do these calculations in practice.

In the 1981, physicist Richard Feynman proposed to “turn this lemon to lemonade” by making the following almost tautological observation:

If a physical system cannot be simulated by a computer in T steps, the system can be considered as performing a computation that would take more than T steps

So, he asked whether one could design a quantum system such that its outcome y based on the initial condition x would be some function $y = f(x)$ such that (a) we don’t know how to efficiently compute in any other way, and (b) is actually useful for something.² In 1985, David Deutsch formally suggested the notion of a quantum Turing machine, and the model has been since refined in works of Detusch and Josza and Bernstein and Vazirani. Such a system is now known as a *quantum computer*.

For a while these hypothetical quantum computers seemed useful for one of two things. First, to provide a general-purpose mechanism to simulate a variety of the real quantum systems that people care about. Second, as a challenge to the theory of computation’s approach to model efficient computation by Turing machines, though a challenge that has little bearing to practice, given that this theoretical “extra power” of quantum computer seemed to offer little advantage in the problems people actually want to solve such as combinatorial optimization, machine learning, data structures, etc..

To a significant extent, this is still true today. We have no real evidence that quantum computers, if built, will offer truly significant³ advantage in 99 percent of the applications of computing.⁴ However, there is one cryptography-sized exception: In 1994 Peter Shor showed that quantum computers can solve the integer factoring and discrete logarithm in polynomial time. This result has captured the imagination of a great many people, and completely energized research into

² As its title suggests, Feynman’s [lecture](#) was actually focused on the other side of simulating physics with a computer, but he mentioned that as a “side remark” one could wonder if it’s possible to simulate physics with a new kind of computer - a “quantum computer” which would “not [be] a Turing machine, but a machine of a different kind”. As far as I know, Feynman did not suggest that such a computer could be useful for computations completely outside the domain of quantum simulation, and in fact he found the question of whether quantum mechanics could be simulated by a classical computer to be more interesting.

³ I am using the theorist’ definition of conflating “significant” with “super-polynomial”. As we’ll see, Grover’s algorithm does offer a very generic *quadratic* advantage in computation. Whether that quadratic advantage will ever be good enough to offset in practice the significant overhead in building a quantum computer remains an open question. We also don’t have evidence that super-polynomial speedups *can’t* be achieved for some problems outside the Factoring/Dlog or quantum simulation domains, and there is at least [one company](#) banking on such speedups actually being feasible.

⁴ This “99 percent” is a figure of speech, but not completely so. It seems that for many web servers, the TLS protocol (which based on the current non-lattice based systems would be completely broken by quantum computing) is responsible [for about 1 percent of the CPU usage](#).

quantum computing.

This is both because the hardness of these particular problems provides the foundations for securing such a huge part of our communications (and these days, our economy), as well as it was a powerful demonstration that quantum computers could turn out to be useful for problems that a-priori seemed to have nothing to do with quantum physics. As we'll discuss later, at the moment there are several intensive efforts to construct large scale quantum computers. It seems safe to say that, as far as we know, in the next five years or so there will not be a quantum computer large enough to factor, say, a 1024 bit number, but there it is quite possible that some quantum computer will be built that is strong enough to achieve some task that is too inefficient to achieve with a non-quantum or "classical" computer (or at least requires more resources classically than it would for this computer). When and if such a computer is built that can break reasonable parameters of Diffie Hellman, RSA and elliptic curve cryptography is anybody's guess. It could also be a "self destroying prophecy" whereby the existence of a small-scale quantum computer would cause everyone to shift away to lattice-based crypto which in turn will diminish the motivation to invest the huge resources needed to build a large scale quantum computer.⁵

The above summary might be all that you need to know as a cryptographer, and enough motivation to study lattice-based cryptography as we do in this course. However, because quantum computing is such a beautiful and (like cryptography) counter-intuitive concept, we will try to give at least a hint of what is it about and how does Shor's algorithm work.

⁵ Of course, given that we're still hearing of attacks exploiting "export grade" cryptography that was supposed to disappear with 1990's, I imagine that we'll still have products running 1024 bit RSA when everyone has a quantum laptop.

19.1 Quantum 101

We now present some of the basic notions in quantum information. It is very useful to contrast these notions to the setting of *probabilistic* systems and see how "negative probabilities" make a difference. This discussion is somewhat brief. The chapter on quantum computation in my [book with Arora](#) (see [draft here](#)) is one relatively short resource that contains essentially everything we discuss here. See also this [blog post of Aaronson](#) for a high level explanation of Shor's algorithm which ends with links to several more detailed expositions. See also [this lecture](#) of Aaronson for a great discussion of the feasibility of quantum computing (Aaronson's [course lecture notes](#) and the [book](#) that they spawned are fantastic reads as well).

States: We will consider a simple quantum system that includes n objects (e.g., electrons/photon/transistors/etc.) each of which can be in either an “on” or “off” state - i.e., each of them can encode a single *bit* of information, but to emphasize the “quantumness” we will call it a *qubit*. A *probability distribution* over such a system can be described as a 2^n dimensional vector v with non-negative entries summing up to 1, where for every $x \in \{0,1\}^n$, v_x denotes the probability that the system is in state x . As we mentioned, quantum mechanics allows negative (in fact even complex) probabilities and so a *quantum state* of the system can be described as a 2^n dimensional vector v such that $\|v\|^2 = \sum_x |v_x|^2 = 1$.

Measurement: Suppose that we were in the classical probabilistic setting, and that the n bits are simply random coins. Thus we can describe the *state* of the system by the 2^n -dimensional vector v such that $v_x = 2^{-n}$ for all x . If we *measure* the system and see what the coins came out, we will get the value x with probability v_x . Naturally, if we measure the system twice we will get the same result. Thus, after we see that the coin is x , the new state of the system *collapses* to a vector v such that $v_y = 1$ if $y = x$ and $v_y = 0$ if $y \neq x$. In a quantum state, we do the same thing: if we *measure* a vector v corresponds to turning it with probability $|v_x|^2$ into a vector that has 1 on coordinate x and zero on all the other coordinates.

Operations: In the classical probabilistic setting, if we have a system in state v and we apply some function $f : \{0,1\}^n \rightarrow \{0,1\}^n$ then this transforms v to the state w such that $w_y = \sum_{x:f(x)=y} v_x$.

Another way to state this, is that $w = M_f$ where M_f is the matrix such that $M_{f(x),x} = 1$ for all x and all other entries are 0. If we toss a coin and decide with probability $1/2$ to apply f and with probability $1/2$ to apply g , this corresponds to the matrix $(1/2)M_f + (1/2)M_g$. More generally, the set of operations that we can apply can be captured as the set of convex combinations of all such matrices- this is simply the set of non-negative matrices whose columns all sum up to 1- the *stochastic* matrices. In the quantum case, the operations we can apply to a quantum state are encoded as a *unitary* matrix, which is a matrix M such that $\|Mv\| = \|v\|$ for all vectors v .

Elementary operations: Of course, even in the probabilistic setting, not every function $f : \{0,1\}^n \rightarrow \{0,1\}^n$ is efficiently computable. We think of a function as efficiently computable if it is composed of polynomially many elementary operations, that involve at most 2 or 3 bits or so (i.e., Boolean *gates*). That is, we say that a matrix M is *elementary* if it only modifies three bits. That is, M is obtained by “lifting” some 8×8 matrix M' that operates on three bits i, j, k , leaving

all the rest of the bits intact. Formally, given an 8×8 matrix M' (indexed by strings in $\{0,1\}^3$) and three distinct indices $i < j < k \in \{1, \dots, n\}$ we define the n -lift of M' with indices i, j, k to be the $2^n \times 2^n$ matrix M such that for every strings x and y that agree with each other on all coordinates except possibly i, j, k , $M_{x,y} = M'_{x_i x_j x_k, y_i y_j y_k}$ and otherwise $M_{x,y} = 0$. Note that if M' is of the form M'_f for some function $f : \{0,1\}^3 \rightarrow \{0,1\}^3$ then $M = M_g$ where $g : \{0,1\}^n \rightarrow \{0,1\}^n$ is defined as $g(x) = f(x_i x_j x_k)$. We define M as an *elementary stochastic matrix* or a *probabilistic gate* if M is equal to an n lift of some stochastic 8×8 matrix M' . The quantum case is similar: a *quantum gate* is a $2^n \times 2^n$ matrix that is an N lift of some unitary 8×8 matrix M' . It is an exercise to prove that lifting preserves stochasticity and unitarity. That is, every probabilistic gate is a stochastic matrix and every quantum gate is a unitary matrix.

Complexity: For every stochastic matrix M we can define its *randomized complexity*, denoted as $R(M)$ to be the minimum number T such that M is can be (approximately) obtained by combining T elementary probabilistic gates. To be concrete, we can define $R(M)$ to be the minimum T such that there exists T elementary matrices M_1, \dots, M_T such that for every x , $\sum_y |M_{y,x} - (M_T \cdots M_1)_{y,x}| < 0.1$. (It can be shown that $R(M)$ is finite and in fact at most 10^n for every M ; we can do so by writing M as a convex combination of function and writing every function as a composition of functions that map a single string x to y , keeping all other inputs intact.) We will say that a probabilistic process M mapping distributions on $\{0,1\}^n$ to distributions on $\{0,1\}^n$ is *efficiently classically computable* if $R(M) \leq \text{poly}(n)$. If M is a unitary matrix, then we define the *quantum complexity* of M , denoted as $Q(M)$, to be the minimum number T such that there are quantum gates M_1, \dots, M_T satisfying that for every x , $\sum_y |M_{y,x} - (M_T \cdots M_1)_{y,x}|^2 < 0.1$.

We say that M is *efficiently quantumly computable* if $Q(M) \leq \text{poly}(n)$.

Computing functions: We have defined what it means for an operator to be probabilistically or quantumly efficiently computable, but we typically are interested in computing some function $f : \{0,1\}^m \rightarrow \{0,1\}^\ell$. The idea is that we say that f is efficiently computable if the corresponding operator is efficiently computable, except that we also allow to use extra memory and so to embed f in some $n = \text{poly}(m)$. We define f to be *efficiently classically computable* if there is some $n = \text{poly}(m)$ such that the operator M_g is efficiently classically computable where $g : \{0,1\}^n \rightarrow \{0,1\}^\ell$ is defined such that $g(x_1, \dots, x_n) = f(x_1, \dots, x_m)$. In the quantum

case we have a slight twist since the operator M_g is not necessarily a unitary matrix.⁶ Therefore we say that f is *efficiently quantumly computable* if there is $n = \text{poly}(m)$ such that the operator M_g is efficiently quantumly computable where $g : \{0,1\}^n \rightarrow \{0,1\}^n$ is defined as $g(x_1, \dots, x_n) = x_1 \cdots x_m \| (f(x_1 \cdots x_m)0^{n-m-\ell} \oplus x_{m+1} \cdots x_n)$.

Quantum and classical computation: The way we defined what it means for a function to be efficiently quantumly computable, it might not be clear that if $f : \{0,1\}^m \rightarrow \{0,1\}^\ell$ is a function that we can compute by a polynomial size Boolean circuit (e.g., combining polynomially many AND, OR and NOT gates) then it is also quantumly efficiently computable. The idea is that for every gate $g : \{0,1\}^2 \rightarrow \{0,1\}$ we can define an 8×8 unitary matrix M_h where $h : \{0,1\}^3 \rightarrow \{0,1\}^3$ have the form $h(a, b, c) = a, b, c \oplus g(a, b)$. So, if f has a circuit of s gates, then we can dedicate an extra bit for every one of these gates and then run the corresponding s unitary operations one by one, at the end of which we will get an operator that computes the mapping $x_1, \dots, x_{m+\ell+s} = x_1 \cdots x_m \| x_{m+1} \cdots x_{m+s} \oplus f(x_1, \dots, x_m) \| g(x_1 \dots x_m)$ where the the $\ell + i^{\text{th}}$ bit of $g(x_1, \dots, x_n)$ is the result of applying the i^{th} gate in the calculation of $f(x_1, \dots, x_m)$. So this is “almost” what we wanted except that we have this “extra junk” that we need to get rid of. The idea is that we now simply run the same computation again which will basically we mean we XOR another copy of $g(x_1, \dots, x_m)$ to the last s bits, but since $g(x) \oplus g(x) = 0^s$ we get that we compute the map $x \mapsto x_1 \cdots x_m \| (f(x_1, \dots, x_m)0^s \oplus x_{m+1} \cdots x_{m+\ell+s})$ as desired.

The “obviously exponential” fallacy: A priori it might seem “obvious” that quantum computing is exponentially powerful, since to compute a quantum computation on n bits we need to maintain the 2^n dimensional state vector and apply $2^n \times 2^n$ matrices to it. Indeed popular descriptions of quantum computing (too) often say something along the lines that the difference between quantum and classical computer is that a classic bit can either be zero or one while a qubit can be in both states at once, and so in many qubits a quantum computer can perform exponentially many computations at once. Depending on how you interpret this, this description is either false or would apply equally well to *probabilistic computation*. However, for probabilistic computation it is a not too hard exercise to show that if $f : \{0,1\}^m \rightarrow \{0,1\}^n$ is an efficiently computable function then it has a polynomial size circuit of AND, OR and NOT gates.⁷ Moreover, this “obvious” approach for simulating a quantum computation will take not just exponential time but *exponential space* as well, while it is not hard to show that using a simple recursive formula one can calculate the final quantum state using *polynomial*

⁶ It is a good exercise to verify that for every $g : \{0,1\}^n \rightarrow \{0,1\}^n$, M_g is unitary if and only if g is a permutation.

⁷ It is a good exercise to show that if M is a probabilistic process with $R(M) \leq T$ then there exists a probabilistic circuit of size, say, $100Tn^2$ that approximately computes M in the sense that for every input x , $\sum_{y \in \{0,1\}^n} |\mathbb{P}[C(x) = y] - M_{x,y}| < 1/3$.

space (in physics parlance this is known as “Feynman path integrals”). So, the exponentially long vector description by itself does not imply that quantum computers are exponentially powerful. Indeed, we cannot *prove* that they are (since in particular we can’t prove that *every* polynomial space calculation can be done in polynomial time, in complexity parlance we don’t know how to rule out that $P = PSPACE$), but we do have some problems (integer factoring most prominently) for which they do provide exponential speedup over the currently best *known* classical (deterministic or probabilistic) algorithms.

19.1.1 Physically realizing quantum computation

To realize quantum computation one needs to create a system with n independent binary states (i.e., “qubits”), and be able to manipulate small subsets of two or three of these qubits to change their state. While by the way we defined operations above it might seem that one needs to be able to perform arbitrary unitary operations on these two or three qubits, it turns out that there several choices for *universal sets* - a small constant number of gates that generate all others. The biggest challenge is how to keep the system from being measured and *collapsing* to a single classical combination of states. This is sometimes known as the *coherence time* of the system. The **threshold theorem** says that there is some absolute constant level of errors τ so that if errors are created at every gate at rate smaller than τ then we can recover from those and perform arbitrary long computations. (Of course there are different ways to model the errors and so there are actually several threshold *theorems* corresponding to various noise models).

There have been several proposals to build quantum computers:

- **Superconducting quantum computers** use super-conducting electric circuits to do quantum computation. **Recent works** have shown one can keep these superconducting qubits fairly robust to the point one can do some error correction on them (see also [here](#)).
- **Trapped ion quantum computers** Use the states of an ion to simulate a qubit. People have made some **recent advances** on these computers too. While it’s not clear that’s the right measuring yard, the **current best implementation** of Shor’s algorithm (for factoring 15) is done using an ion-trap computer.
- **Topological quantum computers** use a different technology, which is more stable by design but arguably harder to manipulate to

create quantum computers.

These approaches are not mutually exclusive and it could be that ultimately quantum computers are built by combining all of them together. I am not at all an expert on this matter, but it seems that progress has been slow but steady and it is quite possible that we'll see a 20-50 qubit computer sometime in the next 5-10 years.

19.1.2 Bra-ket notation

Quantum computing is very confusing and counterintuitive for many reasons. But there is also a “cultural” reason why people sometimes find quantum arguments hard to follow. Quantum folks follow their own special **notation** for vectors. Many non quantum people find it ugly and confusing, while quantum folks secretly wish they people used it all the time, not just for non-quantum linear algebra, but also for restaurant bills and elementary school math classes.

The notation is actually not so confusing. If $x \in \{0,1\}^n$ then $|x\rangle$ denotes the x^{th} standard basis vector in 2^n dimension. That is $|x\rangle$ 2^n -dimensional column vector that has 1 in the x^{th} coordinate and zero everywhere else. So, we can describe the column vector that has α_x in the x^{th} entry as $\sum_{x \in \{0,1\}^n} \alpha_x |x\rangle$. One more piece of notation that is useful is that if $x \in \{0,1\}^n$ and $y \in \{0,1\}^m$ then we identify $|x\rangle|y\rangle$ with $|xy\rangle$ (that is, the 2^{n+m} dimensional vector that has 1 in the coordinate corresponding to the concatenation of x and y , and zero everywhere else). This is more or less all you need to know about this notation to follow this lecture.⁸

A quantum gate is an operation on at most three bits, and so it can be completely specified by what it does to the 8 vectors $|000\rangle, \dots, |111\rangle$. Quantum states are always unit vectors and so we sometimes omit the normalization for convenience; for example we will identify the state $|0\rangle + |1\rangle$ with its normalized version $\frac{1}{\sqrt{2}}|0\rangle + \frac{1}{\sqrt{2}}|1\rangle$.

19.1.3 Bell's Inequality

There is something weird about quantum mechanics. In 1935 **Einsteinst, Podolsky and Rosen (EPR)** tried to pinpoint this issue by highlighting a previously unrealized corollary of this theory. It was already realized that the fact that quantum measurement collapses the state to a definite aspect yields the *uncertainty principle*, whereby if you measure a quantum system in one orthogonal basis, you will

⁸ If you are curious, there is an analog notation for *row* vectors as $\langle x|$. Generally if u is a vector then $|u\rangle$ would be its form as a column vector and $\langle u|$ would be its form as a row product. Hence since $u^\top v = \langle u, v \rangle$ the inner product of u and v can be thought of as $\langle u||v\rangle$. The *outer product* (the matrix whose i,j entry is $u_i v_j$) is denoted as $|u\rangle\langle v|$.

not know how it would have measured in an incoherent basis to it (such as position vs. momentum). What EPR showed was that quantum mechanics results in so called “spooky action at a distance” where if you have a system of two particles then measuring one of them would instantenously effect the state of the other. Since this “state” is just a mathematical description, as far as I know the EPR paper was considered to be a thought experiment showing troubling aspects of quantum mechanics, without bearing on experiment. This changed when in 1965 John Bell showed an actual experiment to test the predictions of EPR and hence pit intuitive common sense against the predictions of quantum mechanics. Quantum mechanics won. Nonetheless, since the results of these experiments are so obviously wrong to anyone that has ever sat in an armchair, that there are still a number of **Bell denialists** arguing that quantum mechanics is wrong in some way.

So, what is this Bell’s Inequality? Suppose that Alice and Bob try to convince you they have telepathic ability, and they aim to prove it via the following experiment. Alice and Bob will be in separate closed rooms.⁹ You will interrogate Alice and your associate will interrogate Bob. You choose a random bit $x \in \{0,1\}$ and your associate chooses a random $y \in \{0,1\}$. We let a be Alice’s response and b be Bob’s response. We say that Alice and Bob win this experiment if $a \oplus b = x \wedge y$.

Now if Alice and Bob are not telepathic, then they need to agree in advance on some strategy. The most general form of such a strategy is that Alice and Bob agree on some distribution over a pair of functions $d, g : \{0,1\} \rightarrow \{0,1\}$, such that Alice will set $a = f(x)$ and Bob will set $b = g(y)$. Therefore, the following claim, which is basically Bell’s Inequality,¹⁰ implies that Alice and Bob cannot succeed in this game with probability higher than 3/4:

Claim: For every two functions $f, g : \{0,1\} \rightarrow \{0,1\}$ there exist some $x, y \in \{0,1\}$ such that $f(x) \oplus g(y) \neq x \wedge y$.

Proof: Suppose toward a contradiction that f, g satisfy $f(x) \oplus g(y) = x \wedge y$ (*) or $f(x) = (x \wedge y) \oplus g(y)$ (*); So if $y = 0$ it must be that $f(x) = 0$ for all x , but on the other hand, if $y = 1$ then for (*) to hold then it must be that $f(x) = x \oplus g(1)$ but that means that f cannot be constant. QED

An amazing **experimentally verified** fact is that quantum mechanics allows for telepathy:¹¹

Claim: There is a strategy for Alice and Bob to succeed in this game with probability at least 0.8.

⁹ If you are extremely paranoid about Alice and Bob communicating with one another, you can coordinate with your assistant to perform the experiment exactly at the same time, and make sure that the rooms are so that Alice and Bob couldn’t communicate to each other in time the results of the coin even if they do so at the speed of light.

¹⁰ This form of Bell’s game was shown by CHSH

¹¹ More accurately, one either has to give up on a “billiard ball type” theory of the universe or believe in telepathy (believe it or not, some scientists went for the latter option).

Proof: The main idea is for Alice and Bob to first prepare a 2-qubit quantum system in the state (up to normalization) $|00\rangle + |11\rangle$ (this is known as an *EPR pair*). Alice takes the first qubit in this system to her room, and Bob takes the qubit to his room. Now, when Alice receives x if $x = 0$ she does nothing and if $x = 1$ she applies the unitary map $R_{\pi/8}$ to her qubit where $R_\theta = \begin{pmatrix} \cos\theta & \sin -\theta \\ \sin \theta & \cos \theta \end{pmatrix}$. When Bob receives y , if $y = 0$ he does nothing and if $y = 1$ he applies the unitary map $R_{-\pi/8}$ to his qubit. Then each one of them measures their qubit and sends this as their response. Recall that to win the game Bob and Alice want their outputs to be more likely to differ if $x = y = 1$ and to be more likely to agree otherwise.

If $x = y = 0$ then the state does not change and Alice and Bob always output either both 0 or both 1, and hence in both case $a \oplus b = x \wedge y$. If $x = 0$ and $y = 1$ then after Alice measures her bit, if she gets 0 then Bob's state is equal to $-\cos(\pi/8)|0\rangle - \sin(\pi/8)|1\rangle$ which will equal 0 with probability $\cos^2(\pi/8)$. The case that Alice gets 1, or that $x = 1$ and $y = 0$, is symmetric, and so in all the cases where $x \neq y$ (and hence $x \wedge y = 0$) the probability that $a = b$ will be $\cos^2(\pi/8) \geq 0.85$. For the case that $x = 1$ and $y = 1$, direct calculation via trigonometric identities yields that all four options for (a, b) are equally likely and hence in this case $a = b$ with probability 0.5. The overall probability of winning the game is at least $\frac{1}{4} \cdot 1 + \frac{1}{2} \cdot 0.85 + \frac{1}{4} \cdot 0.5 = 0.8$. QED

Quantum vs probabilistic strategies: It is instructive to understand what is it about quantum mechanics that enabled this gain in Bell's Inequality. For this, consider the following analogous probabilistic strategy for Alice and Bob. They agree that each one of them output 0 if he or she get 0 as input and outputs 1 with probability p if they get 1 as input. In this case one can see that their success probability would be $\frac{1}{4} \cdot 1 + \frac{1}{2}(1-p) + \frac{1}{4}[2p(1-p)] = 0.75 - 0.5p^2 \leq 0.75$. The quantum strategy we described above can be thought of as a variant of the probabilistic strategy for p is $\sin^2(\pi/8) = 0.15$. But in the case $x = y = 1$, instead of disagreeing only with probability $2p(1-p) = 1/4$, because we can use these negative probabilities in the quantum world and rotate the state in opposite directions, the probability of disagreement ends up being $\sin^2(\pi/4) = 0.5$.

19.2 Grover's Algorithm

Shor's Algorithm, which we'll see in the next lecture, is an amazing achievement, but it only applies to very particular problems. It does not seem to be relevant to breaking AES, lattice based cryptography, or problems not related to quantum computing at all such as scheduling, constraint satisfaction, traveling salesperson etc.. etc.. Indeed, for the most general form of these search problems, classically we don't know how to do anything much better than brute force search, which takes 2^n time over an n -bit domain. Lev Grover showed that quantum computers can obtain a quadratic improvement over this brute force search, solving SAT in $2^{n/2}$ time. The effect of Grover's algorithm on cryptography is fairly mild: one essentially needs to double the key lengths of symmetric primitives. But beyond cryptography, if large scale quantum computers end up being built, Grover search and its variants might end up being some of the most useful computational problems they will tackle. Grover's theorem is the following:

Theorem (Grover search , 1996): There is a quantum $O(2^{n/2} \text{poly}(n))$ -time algorithm that given a $\text{poly}(n)$ -sized circuit computing a function $f : \{0,1\}^n \rightarrow \{0,1\}$ outputs a string $x^* \in \{0,1\}^n$ such that $f(x^*) = 1$.

Proof sketch: The proof is not hard but we only sketch it here. The general idea can be illustrated in the case that there exists a single x^* satisfying $f(x^*) = 1$. (There is a classical reduction from the general case to this problem.) As in Simon's algorithm, we can efficiently initialize an n -qubit system to the uniform state $u = 2^{-n/2} \sum_{x \in \{0,1\}^n} |x\rangle$ which has $2^{-n/2}$ dot product with $|x^*\rangle$. Of course if we measure u , we only have probability $(2^{-n/2})^2 = 2^{-n}$ of obtaining the value x^* . Our goal would be to use $O(2^{n/2})$ calls to the oracle to transform the system to a state v with dot product at least some constant $\epsilon > 0$ with the state $|x^*\rangle$.

It is an exercise to show that using *Had* gets we can efficiently compute the unitary operator U such that $Uu = u$ and $Uv = -v$ for every v orthogonal to u . Also, using the circuit for f , we can efficiently compute the unitary operator U^* such that $U^*|x\rangle = |x\rangle$ for all $x \neq x^*$ and $U^*|x^*\rangle = -|x^*\rangle$. It turns out that $O(2^{n/2})$ applications of UU^* to u yield a vector v with $\Omega(1)$ inner product with $|x^*\rangle$. To see why, consider what these operators do in the two dimensional linear subspace spanned by u and $|x^*\rangle$. (Note that the initial state u is in this subspace and all our operators preserve this property.) Let u_\perp be the unit vector orthogonal to u in this subspace and let x_\perp^* be the unit vector orthogonal to $|x^*\rangle$ in this subspace. Restricted to this

subspace, U^* is a reflection along the axis x_{\perp}^* and U is a reflection along the axis u .

Now, let θ be the angle between u and x_{\perp}^* . These vectors are very close to each other and so θ is very small but not zero - it is equal to $\sin^{-1}(2^{-n/2})$ which is roughly $2^{-n/2}$. Now if our state v has angle $\alpha \geq 0$ with u , then as long as α is not too large (say $\alpha < \pi/8$) then this means that v has angle $u + \theta$ with x_{\perp}^* . That means that U^*v will have angle $-\alpha - \theta$ with x_{\perp}^* or $-\alpha - 2\theta$ with u , and hence UU^*v will have angle $\alpha + 2\theta$ with u . Hence in one application from UU^* we move 2θ radians away from u , and in $O(2^{-n/2})$ steps the angle between u and our state will be at least some constant $\epsilon > 0$. Since we live in the two dimensional space spanned by u and $|x\rangle$, it would mean that the dot product of our state and $|x\rangle$ will be at least some constant as well. QED

20

Quantum computing and cryptography II

Bell's Inequality is powerful demonstration that there is something very strange going on with quantum mechanics. But could this “strangeness” be of any use to solve computational problems not directly related to quantum systems? *A priori*, one could guess the answer is *no*. In 1994 Peter Shor showed that one would be wrong:

Theorem 20.1 — Shor’s Theorem. The map that takes an integer m into its prime factorization is efficiently quantumly computable. Specifically, it can be computed using $O(\log^3 m)$ quantum gates.

This is an exponential improvement over the best known classical algorithms, which as we mentioned before, take roughly $2^{O(\log^{1/3} m)}$ time.

We will now sketch the ideas behind Shor’s algorithm. In fact, Shor proved the following more general theorem:

Theorem 20.2 — Order Finding Algorithm. There is a quantum polynomial time algorithm that given a multiplicative Abelian group \mathbb{G} and element $g \in \mathbb{G}$ computes the *order* of g in the group.

Recall that the order of g in \mathbb{G} is the smallest positive integer a such that $g^a = 1$. By “given a group” we mean that we can represent the elements of the group as strings of length $O(\log |\mathbb{G}|)$ and there is a $\text{poly}(\log |\mathbb{G}|)$ algorithm to perform multiplication in the group.

20.1 From order finding to factoring and discrete log

The order finding problem allows not just to factor integers in polynomial time, but also solve the discrete logarithm over arbitrary

Abelian groups, hereby showing that quantum computers will break not just RSA but also Diffie Hellman and Elliptic Curve Cryptography. We merely sketch how one reduces the factoring and discrete logarithm problems to order finding: (see some of the sources above for the full details)

- For **factoring**, let us restrict to the case $m = pq$ for distinct p, q . Recall that we showed that finding the size $(p - 1)(q - 1) = m - p - q + 1$ of the group \mathbb{Z}_m^* is sufficient to recover p and q . One can show that if we pick a few random x 's in \mathbb{Z}_m^* and compute their order, the least common multiplier of these orders is likely to be the group size.
- For **discrete log** in a group G , if we get $X = g^x$ and need to recover x , we can compute the order of various elements of the form $X^a g^b$. The order of such an element is a number c satisfying $c(xa + b) = 0 \pmod{|G|}$. Again, with a few random examples we will get a non trivial example (where $c \neq 0 \pmod{|G|}$) and be able to recover the unknown x .

20.2 Finding periods of a function: Simon's Algorithm

Let \mathbb{H} be some Abelian group with a group operation that we'll denote by \oplus , and f be some function mapping \mathbb{H} to an arbitrary set (which we can encode as $\{0, 1\}^*$). We say that f has *period h^** for some $h^* \in \mathbb{H}$ if for every $x, y \in \mathbb{H}$, $f(x) = f(y)$ if and only if $y = x \oplus kh^*$ for some integer k . Note that if G is some Abelian group, then if we define $\mathbb{H} = \mathbb{Z}_{|G|}$, for every element $g \in G$, the map $f(a) = g^a$ is a periodic map over \mathbb{H} with period the order of g . So, finding the order of an item reduces to the question of finding the period of a function.

How do we generally find the period of a function? Let us consider the simplest case, where f is a function from \mathbb{R} to \mathbb{R} that is h^* periodic for some number h^* , in the sense that f repeats itself on the intervals $[0, h^*]$, $[h^*, 2h^*]$, $[2h^*, 3h^*]$, etc.. How do we find this number h^* ? The key idea would be to transform f from the *time* to the *frequency* domain. That is, we use the *Fourier transform* to represent f as a sum of wave functions. In this representation wavelengths that divide the period h^* would get significant mass, while wavelengths that don't would likely "cancel out".

Similarly, the main idea behind Shor's algorithm is to use a tool known as the *quantum fourier transform* that given a circuit computing the function $f : \mathbb{H} \rightarrow \mathbb{R}$, creates a quantum state over roughly

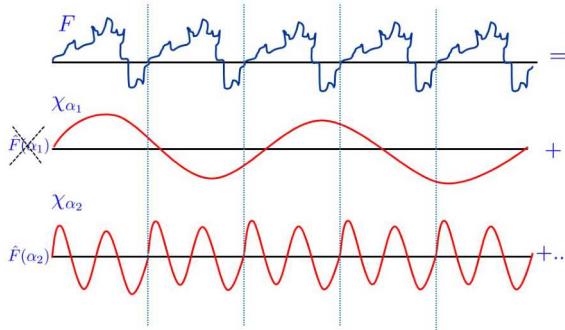


Figure 20.1: If f is a periodic function then when we represent it in the Fourier transform, we expect the coefficients corresponding to wavelengths that do not evenly divide the period to be very small, as they would tend to “cancel out”.

$\log |\mathbb{H}|$ qubits (and hence dimension $|\mathbb{H}|$) that corresponds to the Fourier transform of f . Hence when we measure this state, we get a group element h with probability proportional to the square of the corresponding Fourier coefficient. One can show that if f is h^* -periodic then we can recover h^* from this distribution.

Shor carried out this approach for the group $\mathbb{H} = \mathbb{Z}_q^*$ for some q , but we will start be seeing this for the group $\mathbb{H} = \{0,1\}^n$ with the XOR operation. This case is known as *Simon's algorithm* (given by Dan Simon in 1994) and actually preceded (and inspired) Shor's algorithm:

Theorem 20.3 — Simon's Algorithm. If $f : \{0,1\}^n \rightarrow \{0,1\}^*$ is polynomial time computable and satisfies the property that $f(x) = f(y)$ iff $x \oplus y = h^*$ then there exists a quantum polynomial-time algorithm that outputs a random $h \in \{0,1\}^n$ such that $\langle h, h^* \rangle \equiv 0 \pmod{2}$.

Note that given $O(n)$ such samples, we can recover h^* with high probability by solving the corresponding linear equations.

Proof. Let HAD be the 2×2 unitary matrix corresponding to the one qubit operation $|0\rangle \mapsto \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle)$ and $|1\rangle \mapsto \frac{1}{\sqrt{2}}(|0\rangle - |1\rangle)$ or $|a\rangle \mapsto \frac{1}{\sqrt{2}}(|0\rangle + (-1)^a|1\rangle)$. Given the state $|0^{n+m}\rangle$ we can apply this map to each one of the first n qubits to get the state $2^{-n/2} \sum_{x \in \{0,1\}^n} |x\rangle |0^m\rangle$ and then we can apply the gates of f to map this to the state $2^{-n/2} \sum_{x \in \{0,1\}^n} |x\rangle |f(x)\rangle$ now suppose that we apply this operation again to the first n qubits then we get the state $2^{-n} \sum_{x \in \{0,1\}^n} \prod_{i=1}^n (|0\rangle + (-1)^{x_i} |1\rangle) |f(x)\rangle$ which if we open up each

one of these product and look at all 2^n choices $y \in \{0,1\}^n$ (with $y_i = 0$ corresponding to picking $|0\rangle$ and $y_i = 1$ corresponding to picking $|1\rangle$ in the i^{th} product) we get $2^{-n} \sum_{x \in \{0,1\}^n} \sum_{y \in \{0,1\}^n} (-1)^{\langle x,y \rangle} |y\rangle |f(x)\rangle$. Now under our assumptions for every particular z in the image of f , there exist exactly two preimages x and $x \oplus h^*$ such that $f(x) = f(x + h^*) = z$. So, if $\langle y, h^* \rangle = 0 \pmod{2}$, we get that $(-1)^{\langle x,y \rangle} + (-1)^{\langle x,y+h^* \rangle} = 2$ and otherwise we get $(-1)^{\langle x,y \rangle} + (-1)^{\langle x,y+h^* \rangle} = 0$. Therefore, if measure the state we will get a pair (y, z) such that $\langle y, h^* \rangle = 0 \pmod{2}$. QED ■

Simon's algorithm seems to really use the special bit-wise structure of the group $\{0,1\}^n$, so one could wonder if it has any relevance for the group \mathbb{Z}_m^* for some exponentially large m . It turns out that the same insights that underlie the well known Fast Fourier Transform (FFT) algorithm can be used to essentially follow the same strategy for this group as well.

20.3 From Simon to Shor

(Note: The presentation here is adapted from the quantum computing chapter in my textbook with Arora.)

We now describe how to achieve Shor's algorithm for order finding. We will not do this for a general group but rather focus our attention on the group \mathbb{Z}_ℓ^* for some number ℓ which is the case of interest for integer factoring and the discrete logarithm modulo primes problems.

That is, we prove the following theorem:

Theorem 20.4 — Shor's Algorithm, restated. For every ℓ and $a \in \mathbb{Z}_\ell^*$, there is a quantum $\text{poly}(\log \ell)$ algorithm to find the order of a in \mathbb{Z}_ℓ^* .

The idea is similar to Simon's algorithm. We consider the map $x \mapsto a^x \pmod{\ell}$ which is a periodic map over \mathbb{Z}_m where $m = |\mathbb{Z}_\ell^*|$ with period being the order of a .

To find the period of this map we will now need to perform a *Quantum Fourier Transform (QFT)* over the group \mathbb{Z}_m instead of $\{0,1\}^n$. This is a quantum algorithm that takes a register from some arbitrary state $f \in \mathbb{C}^m$ into a state whose vector is the Fourier transform \hat{f} of f . The QFT takes only $O(\log^2 m)$ elementary steps and is thus very efficient. Note that we cannot say that this algorithm

"computes" the Fourier transform, since the transform is stored in the amplitudes of the state, and as mentioned earlier, quantum mechanics give no way to "read out" the amplitudes per se. The only way to get information from a quantum state is by *measuring* it, which yields a single basis state with probability that is related to its amplitude. This is hardly representative of the entire Fourier transform vector, but sometimes (as is the case in Shor's algorithm) this is enough to get highly non-trivial information, which we do not know how to obtain using classical (non-quantum) computers.

20.3.1 The Fourier transform over \mathbb{Z}_m

We now define the Fourier transform over \mathbb{Z}_m (the group of integers in $\{0, \dots, m-1\}$ with addition modulo m). We give a definition that is specialized to the current context. For every vector $f \in \mathbb{C}^m$, the *Fourier transform* of f is the vector \hat{f} where the x^{th} coordinate of \hat{f} is defined as¹

$$\hat{f}(x) = \frac{1}{\sqrt{m}} \sum_{y \in \mathbb{Z}_m} f(y) \omega^{xy}$$

where $\omega = e^{2\pi i/m}$.

The Fourier transform is simply a representation of f in the *Fourier basis* $\{\chi_x\}_{x \in \mathbb{Z}_m}$, where χ_x is the vector/function whose y^{th} coordinate is $\frac{1}{\sqrt{m}} \omega^{xy}$. Now the inner product of any two vectors χ_x, χ_z in this basis is equal to

$$\langle \chi_x, \chi_z \rangle = \frac{1}{m} \sum_{y \in \mathbb{Z}_m} \omega^{xy} \overline{\omega^{zy}} = \frac{1}{m} \sum_{y \in \mathbb{Z}_m} \omega^{(x-z)y}. \quad (20.1)$$

But if $x = z$ then $\omega^{(x-z)} = 1$ and hence this sum is equal to 1. On the other hand, if $x \neq z$, then this sum is equal to $\frac{1}{m} \frac{1-\omega^{(x-y)m}}{1-\omega^{x-y}} = \frac{1}{m} \frac{1-1}{1-\omega^{x-y}} = 0$ using the formula for the sum of a geometric series. In other words, this is an *orthonormal* basis which means that the Fourier transform map $f \mapsto \hat{f}$ is a *unitary* operation.

What is so special about the Fourier basis? For one thing, if we identify vectors in \mathbb{C}^m with functions mapping \mathbb{Z}_m to \mathbb{C} , then it's easy to see that every function χ in the Fourier basis is a *homomorphism* from \mathbb{Z}_m to \mathbb{C} in the sense that $\chi(y+z) = \chi(y)\chi(z)$ for every $y, z \in \mathbb{Z}_m$. Also, every function χ is *periodic* in the sense that there exists $r \in \mathbb{Z}_m$ such that $\chi(y+r) = \chi(z)$ for every $y \in \mathbb{Z}_m$ (indeed if $\chi(y) = \omega^{xy}$ then we can take r to be ℓ/x where ℓ is the least common multiple of x and m). Thus, intuitively, if a function $f : \mathbb{Z}_m \rightarrow \mathbb{C}$ is itself periodic (or roughly periodic) then when representing f in the Fourier basis, the coefficients of basis vectors with periods agreeing

¹ In the context of Fourier transform it is customary and convenient to denote the x^{th} coordinate of a vector f by $f(x)$ rather than f_x .

with the period of f should be large, and so we might be able to discover f 's period from this representation. This does turn out to be the case, and is a crucial point in Shor's algorithm.

Fast Fourier Transform.

Denote by FT_m the operation that maps every vector $f \in \mathbb{C}^m$ to its Fourier transform \hat{f} . The operation FT_m is represented by an $m \times m$ matrix whose (x, y) th entry is ω^{xy} . The trivial algorithm to compute it takes m^2 operations. The famous *Fast Fourier Transform* (FFT) algorithm computes the Fourier transform in $O(m \log m)$ operations. We now sketch the idea behind the FFT algorithm as the same idea is used in the *quantum Fourier transform* algorithm.

Note that

$$\begin{aligned}\hat{f}(x) &= \frac{1}{\sqrt{m}} \sum_{y \in \mathbb{Z}_m} f(y) \omega^{xy} = \\ &\frac{1}{\sqrt{m}} \sum_{y \in \mathbb{Z}_m, y \text{ even}} f(y) \omega^{-2x(y/2)} + \omega^x \frac{1}{\sqrt{m}} \sum_{y \in \mathbb{Z}_m, y \text{ odd}} f(y) \omega^{2x(y-1)/2}.\end{aligned}$$

Now since ω^2 is an $m/2$ th root of unity and $\omega^{m/2} = -1$, letting W be the $m/2 \times m/2$ diagonal matrix with diagonal entries $\omega^0, \dots, \omega^{m/2-1}$, we get that

$$FT_m(f)_{low} = FT_{m/2}(f_{even}) + WFT_{m/2}(f_{odd})$$

$$FT_m(f)_{high} = FT_{m/2}(f_{even}) - WFT_{m/2}(f_{odd})$$

where for an m -dimensional vector \vec{v} , we denote by \vec{v}_{even} (resp. \vec{v}_{odd}) the $m/2$ -dimensional vector obtained by restricting \vec{v} to the coordinates whose indices have least significant bit equal to 0 (resp. 1) and by \vec{v}_{low} (resp. \vec{v}_{high}) the restriction of \vec{v} to coordinates with most significant bit 0 (resp. 1).

The equations above are the crux of the divide-and-conquer idea of the FFT algorithm, since they allow to replace a size- m problem with two size- $m/2$ subproblems, leading to a recursive time bound of the form $T(m) = 2T(m/2) + O(m)$ which solves to $T(m) = O(m \log m)$.

20.3.2 Quantum Fourier Transform over \mathbb{Z}_m

The *quantum Fourier transform* is an algorithm to change the state of a quantum register from $f \in \mathbb{C}^m$ to its Fourier transform \hat{f} .

Theorem 20.5 — Quantum Fourier Transform (Bernstein-Vazirani). For every

m and $m = 2^m$ there is a quantum algorithm that uses $O(m^2)$ elementary quantum operations and transforms a quantum register in state $f = \sum_{x \in \mathbb{Z}_m} f(x)|x\rangle$ into the state $\hat{f} = \sum_{x \in \mathbb{Z}_m} \hat{f}(x)|x\rangle$, where $\hat{f}(x) = \frac{1}{\sqrt{m}} \sum_{y \in \mathbb{Z}_m} \omega^{xy} f(y)$.

The crux of the algorithm is the FFT equations, which allow the problem of computing FT_m , the problem of size m , to be split into two identical subproblems of size $m/2$ involving computation of $FT_{m/2}$, which can be carried out recursively using the same elementary operations. (Aside: Not every divide-and-conquer classical algorithm can be implemented as a fast quantum algorithm; we are really using the structure of the problem here.)

We now describe the algorithm and the state, neglecting normalizing factors.

1. initial state: $f = \sum_{x \in \mathbb{Z}_m} f(x)|x\rangle$
2. Recursively run $FT_{m/2}$ on $m - 1$ most significant qubits (state: $(FT_{m/2}f_{even})|0\rangle + (FT_{m/2}f_{odd})|1\rangle$)
3. If LSB is 1 then compute W on $m - 1$ most significant qubits (see below). (state : $(FT_{m/2}f_{even})|0\rangle + (WFT_{m/2}f_{odd})|1\rangle$)
4. Apply Hadmard gate H to least significant qubit. (state: $(FT_{m/2}f_{even})(|0\rangle + |1\rangle) + (WFT_{m/2}f_{odd})(|0\rangle - |1\rangle) = (FT_{m/2}f_{even} + WFT_{m/2}f_{odd})|0\rangle + (FT_{m/2}f_{even} - WFT_{m/2}f_{odd})|1\rangle$)
5. Move LSB to the most significant position (state: $|0\rangle(FT_{m/2}f_{even} + WFT_{m/2}f_{odd}) + |1\rangle(FT_{m/2}f_{even} - WFT_{m/2}f_{odd}) = \hat{f}$)

The transformation W on $m - 1$ qubits can be defined by $|x\rangle \mapsto \omega^x = \omega^{\sum_{i=0}^{m-2} 2^i x_i}$ (where x_i is the i^{th} qubit of x). It can be easily seen to be the result of applying for every $i \in \{0, \dots, m - 2\}$ the following elementary operation on the i^{th} qubit of the register:

$$|0\rangle \mapsto |0\rangle \text{ and } |1\rangle \mapsto \omega^{2^i}|1\rangle.$$

The final state is equal to \hat{f} by the FFT equations (we leave this as an exercise)

20.4 Shor's Order-Finding Algorithm.

We now present the central step in Shor's factoring algorithm: a quantum polynomial-time algorithm to find the *order* of an integer a modulo an integer ℓ .

Theorem 20.6 — Order finding algorithm, restated. There is a polynomial-time quantum algorithm that on input A, N (represented in binary) finds the smallest r such that $A^r = 1 \pmod{N}$.

Let $t = \lceil 5 \log(A + N) \rceil$. Our register will consist of $t + \text{polylog}(N)$ qubits. Note that the function $x \mapsto A^x \pmod{N}$ can be computed in $\text{polylog}(N)$ time and so we will assume that we can compute the map $|x\rangle|y\rangle \mapsto |x\rangle|y \oplus (A^x \pmod{N})\rangle$ (where we identify a number $X \in \{0, \dots, N-1\}$ with its representation as a binary string of length $\log N$).² Now we describe the order-finding algorithm. It uses a tool of elementary number theory called *continued fractions* which allows us to approximate (using a classical algorithm) an arbitrary real number α with a rational number p/q where there is a prescribed upper bound on q (see below)

We now describe the algorithm and the state, this time *including* normalizing factors.

1. Apply Fourier transform to the first m bits. (state: $\frac{1}{\sqrt{m}} \sum_{x \in \mathbb{Z}_m} |x\rangle|0^n\rangle$)
2. Compute the transformation $|x\rangle|y\rangle \mapsto |x\rangle|y \oplus (A^x \pmod{N})\rangle$.
(state: $\frac{1}{\sqrt{m}} \sum_{x \in \mathbb{Z}_m} |x\rangle|A^x \pmod{N}\rangle$)
3. Measure the second register to get a value y_0 . (state: $\frac{1}{\sqrt{K}} \sum_{\ell=0}^{K-1} |x_0 + \ell r\rangle|y_0\rangle$ where x_0 is the smallest number such that $A^{x_0} = y_0 \pmod{N}$ and $K = \lfloor (m-1-x_0)/r \rfloor$.)
4. Apply the Fourier transform to the first register. (state: $\frac{1}{\sqrt{m}\sqrt{K}} \left(\sum_{x \in \mathbb{Z}_n} \sum_{\ell=0}^{K-1} \omega^{(x_0+\ell r)x} |x\rangle \right) |y_0\rangle$)

In the analysis, it will suffice to show that this algorithm outputs the order r with probability at least $\Omega(1/\log N)$ (we can always amplify the algorithm's success by running it several times and taking the smallest output).

20.4.1 Analysis: the case that $r|m$

We start by analyzing the algorithm in the case that $m = rc$ for some integer c . Though very unrealistic (remember that m is a power of $2!$) this gives the intuition why Fourier transforms are useful for detecting periods.

Claim: In this case the value x measured will be equal to ac for a random $a \in \{0, \dots, r-1\}$.

² To compute this map we may need to extend the register by some additional $\text{polylog}(N)$ many qubits, but we can ignore them as they will always be equal to zero except in intermediate computations.

The claim concludes the proof since it implies that $x/m = a/r$ where a is random integer less than r . Now for every r , at least $\Omega(r/\log r)$ of the numbers in $[r - 1]$ are co-prime to r . Indeed, the prime number theorem says that there are at least this many primes in this interval, and since r has at most $\log r$ prime factors, all but $\log r$ of these primes are co-prime to r . Thus, when the algorithm computes a rational approximation for x/m , the denominator it will find will indeed be r .

To prove the claim, we compute for every $x \in \mathbb{Z}_m$ the absolute value of $|x\rangle$'s coefficient before the measurement. Up to some normalization factor this is

$$\left| \sum_{\ell=0}^{c-1} \omega^{(x_0 + \ell r)x} \right| = \left| \omega^{x_0 c' c} \right| \left| \sum_{\ell=0}^{c-1} \omega^{\ell r x} \right| = 1 \cdot \left| \sum_{\ell=0}^{c-1} \omega^{\ell r x} \right|.$$

If c does not divide x then ω^r is a c^{th} root of unity, so $\sum_{\ell=0}^{c-1} \omega^{\ell r x} = 0$ by the formula for sums of geometric progressions. Thus, such a number x would be measured with zero probability. But if $x = cj$ then $\omega^{\ell r x} = \omega^{rcj\ell} = \omega^{Mj} = 1$, and hence the amplitudes of all such x 's are equal for all $j \in \{0, 1, \dots, r-1\}$.

The general case

In the general case, where r does not necessarily divide m , we will not be able to show that the measured value x satisfies $m|xr$. However, we will show that with $\Omega(1/\log r)$ probability, **(1)** xr will be “almost divisible” by m in the sense that $0 \leq xr \pmod m < r/10$ and **(2)** $\lfloor xr/m \rfloor$ is coprime to r .

Condition **(1)** implies that $|xr - cM| < r/10$ for $c = \lfloor xr/m \rfloor$. Dividing by rM gives $\left| \frac{x}{m} - \frac{c}{r} \right| < \frac{1}{10M}$. Therefore, $\frac{c}{r}$ is a rational number with denominator at most N that approximates $\frac{x}{m}$ to within $1/(10M) < 1/(4N^4)$. It is not hard to see that such an approximation is unique (again left as an exercise) and hence in this case the algorithm will come up with c/r and output the denominator r .

Thus all that is left is to prove the next two lemmas. The first shows that there are $\Omega(r/\log r)$ values of x that satisfy the above two conditions and the second shows that each is measured with probability $\Omega((1/\sqrt{r})^2) = \Omega(1/r)$.

Lemma 1: There exist $\Omega(r/\log r)$ values $x \in \mathbb{Z}_m$ such that:

1. $0 < xr \pmod m < r/10$
2. $\lfloor xr/m \rfloor$ and r are coprime

Lemma 2: If x satisfies $0 < xr \pmod{m} < r/10$ then, before the measurement in the final step of the order-finding algorithm, the coefficient of $|x\rangle$ is at least $\Omega(\frac{1}{\sqrt{r}})$.

Proof of Lemma 1 We prove the lemma for the case that r is coprime to m , leaving the general case to the reader. In this case, the map $x \mapsto rx \pmod{m}$ is a permutation of \mathbb{Z}_m^* . There are at least $\Omega(r/\log r)$ numbers in $[1..r/10]$ that are coprime to r (take primes in this range that are not one of r 's at most $\log r$ prime factors) and hence $\Omega(r/\log r)$ numbers x such that $rx \pmod{m} = xr - \lfloor xr/m \rfloor m$ is in $[1..r/10]$ and coprime to r . But this means that $\lfloor xr/m \rfloor$ can not have a nontrivial shared factor with r , as otherwise this factor would be shared with $rx \pmod{m}$ as well.

Proof of Lemma 2: Let x be such that $0 < xr \pmod{m} < r/10$. The absolute value of $|x\rangle$'s coefficient in the state before the measurement is

$$\frac{1}{\sqrt{K}\sqrt{m}} \left| \sum_{\ell=0}^{K-1} \omega^{\ell rx} \right|, \quad (20.2)$$

where $K = \lfloor (m - x_0 - 1)/r \rfloor$. Note that $\frac{m}{2r} < K < \frac{m}{r}$ since $x_0 < N \ll m$.

Setting $\beta = \omega^{rx}$ (note that since $m \nmid rx$, $\beta \neq 1$) and using the formula for the sum of a geometric series, this is at least $\frac{\sqrt{r}}{2M} \left| \frac{1-\beta^{\lceil m/r \rceil}}{1-\beta} \right| = \frac{\sqrt{r}}{2M} \frac{\sin(\theta \lceil m/r \rceil / 2)}{\sin(\theta/2)}$, where $\theta = \frac{rx \pmod{m}}{m}$ is the angle such that $\beta = e^{i\theta}$ (see Figure [quantum:fig:theta] for a proof by picture of the last equality). Under our assumptions $\lceil m/r \rceil \theta < 1/10$ and hence (using the fact that $\sin \alpha \sim \alpha$ for small angles α), the coefficient of x is at least $\frac{\sqrt{r}}{4M} \lceil m/r \rceil \geq \frac{1}{8\sqrt{r}}$

This completes the proof of [Theorem 20.6](#).

20.5 Rational approximation of real numbers

In many settings, including Shor's algorithm, we are given a real number in the form of a program that can compute its first t bits in $\text{poly}(t)$ time. We are interested in finding a close approximation to this real number of the form a/b , where there is a prescribed upper bound on b . Continued fractions is a tool in number theory that is useful for this.

A *continued fraction* is a number of the following form: $a_0 + \cfrac{1}{a_1 + \cfrac{1}{a_2 + \cfrac{1}{a_3 + \dots}}}$ for a_0 a non-negative integer and a_1, a_2, \dots positive in-

tegers.

Given a real number $\alpha > 0$, we can find its representation as an *infinite* fraction as follows: split α into the integer part $\lfloor \alpha \rfloor$ and fractional part $\alpha - \lfloor \alpha \rfloor$, find recursively the representation R of $1/(\alpha - \lfloor \alpha \rfloor)$, and then write

$$\alpha = \lfloor \alpha \rfloor + \frac{1}{R}. \quad (20.3)$$

If we continue this process for n steps, we get a rational number, denoted by $[a_0, a_1, \dots, a_n]$, which can be represented as $\frac{p_n}{q_n}$ with p_n, q_n coprime. The following facts can be proven using induction:

- $p_0 = a_0, q_0 = 1$ and for every $n > 1$, $p_n = a_n p_{n-1} + p_{n-2}$, $q_n = a_n q_{n-1} + q_{n-2}$.
- $\frac{p_n}{q_n} - \frac{p_{n-1}}{q_{n-1}} = \frac{(-1)^{n-1}}{q_n q_{n-1}}$

Furthermore, it is known that $\left| \frac{p_n}{q_n} - \alpha \right| < \frac{1}{q_n q_{n+1}}$ (*) which implies that $\frac{p_n}{q_n}$ is the *closest* rational number to α with denominator at most q_n . It also means that if α is extremely close to a rational number, say, $|\alpha - \frac{a}{b}| < \frac{1}{4b^4}$ for some coprime a, b then we can find a, b by iterating the continued fraction algorithm for $\text{polylog}(b)$ steps. Indeed, let q_n be the first denominator such that $q_{n+1} \geq b$. If $q_{n+1} > 2b^2$ then (*) implies that $\left| \frac{p_n}{q_n} - \alpha \right| < \frac{1}{2b^2}$. But this means that $\frac{p_n}{q_n} = \frac{a}{b}$ since there is at most one rational number of denominator at most b that is so close to α . On the other hand, if $q_{n+1} \leq 2b^2$ then since $\frac{p_{n+1}}{q_{n+1}}$ is closer to α than $\frac{a}{b}$, $\left| \frac{p_{n+1}}{q_{n+1}} - \alpha \right| < \frac{1}{4b^4}$, again meaning that $\frac{p_{n+1}}{q_{n+1}} = \frac{a}{b}$. It's not hard to verify that $q_n \geq 2^{n/2}$, implying that p_n and q_n can be computed in $\text{polylog}(q_n)$ time.

20.5.1 Quantum cryptogrpahy

There is another way in which quantum mechanics interacts with cryptography. These “spooky actions at a distance” have been suggested by Weisner and Bennet-Brassard as a way in which parties can create a secret shared key over an insecure channel. On one hand, this concept does not require as much control as general-purpose quantum computing, and so it has in fact been **demonstrated physically**. On the other hand, unlike transmitting standard digital information, this “insecure channel” cannot be an arbitrary media such as wifi etc.. but rather one needs fiber optics, lasers, etc.. Unlike quantum computers, where we only need one of those to break RSA, to actually use key exchange at scale we need to setup these type of networks, and so it is unclear if this approach will ever dominate

the solution of Alice sending to Bob a Brink's truck with the shared secret key. People have proposed some other ways to use the interesting properties of quantum mechanics for cryptographic purposes including **quantum money** and **quantum software protection**.

21

Software Obfuscation

Let us stop and think of the notions we have seen in cryptography. We have seen that under reasonable computational assumptions (such as LWE) we can achieve the following:

- CPA secure *private key encryption* and *Message Authentication codes* (which can be combined to get CCA security or authenticated encryption)- this means that two parties that share a key can have virtual secure channel between them. An adversary cannot get *any additional information* beyond whatever is her prior knowledge given an encryption of a message sent from Alice to Bob. Even if Moreover, she cannot modify this message by even a single bit. It's lucky we only discovered these results from the 1970's onwards—if the Germans have used such an encryption instead of ENIGMA in World War II there's no telling how many more lives would have been lost.
- *Public key encryption* and *digital signatures* that enable Alice and Bob to set up such a virtually secure channel without *sharing a prior key*. This enables our “information economy” and protects virtually every financial transaction over the web. Moreover, it is the crucial mechanism for supplying “over the air” software updates which smart devices whether its phones, cars, thermostats or anything else. Some had predicted that this invention will change the nature of our form of government to **crypto anarchy** and while this may be hyperbole, governments everywhere are **worried** about this invention.
- *Hash functions* and *pseudorandom function* enable us to create authentication tokens for deriving one-time passwords out of shared keys, or deriving long keys from short passwords. They are also useful as a tool in *password based key exchange*, which enables two parties to communicate securely (with fairly good is not over-

whelming probability) when they share a 6 digit PIN, even if the adversary can easily afford much more than 10^6 computational cycles.

- *Fully homomorphic encryption* allows computing over encrypted data. Bob could prepare Alice's taxes without knowing what her income is, and more generally store all her data and perform computations on it, without knowing what the data is.
- *Zero knowledge proofs* can be used to prove a statement is true without revealing *why* its true. In particular since you can use zero knowledge proofs to prove that you posses X bitcoins without giving any information about their identity, they have been used to obtain **fully anonymous electronic currency**.
- *Multiparty secure computation* are a fully general tool that enable Alice and Bob (and Charlie, David, Elana, Fran,...) to perform any computation on their private inputs, whether it is to compute the result of a vote, a second-price auction, privacy-preserving data mining, perform a cryptographic operation in a distributed manner (without any party ever learning the secret key) or simply play poker online without needing to trust any central server.

(BTW all of the above points are notions that you should be familiar and be able to explain what are their security guarantees if you ever need to use them, for example, in the unlikely event that you ever find yourself needing to take a cryptography final exam...)

While clearly there are issues of efficiency, is there anything more in terms of *functionality* we could ask for? Given all these riches, can we be even more greedy?

It turns out that the answer is *yes*. Here are some scenarios that are still not covered by the above tools:

21.1 Witness encryption

Suppose that you have uncovered a conspiracy that involves very powerful people, and you are afraid that something bad might happen to you. You would like an "insurance policy" in the form of writing down everything you know and making sure it is published in the case of your untimely death, but are afraid these powerful people could find and attack any trusted agent. Ideally you would publish an encrypted form of your manuscript far and wide, and make sure the decryption key is automatically revealed if anything happens to you, but how could you do that? A *UA-secure encryption*

(which stands for secure against an [Underwood](#)) attack) gives an ability to create an encryption c of a message m that is CPA secure but such that there is an algorithm D such that on input c and any string w which is a (digitally signed) New York Times obituary for Janine Skorsky will output m .

The technical term for this notion is *witness encryption* by which we mean that for every circuit F we have an algorithm E that on input F and a message m creates a ciphertext c that in a CPA secure, and there is an algorithm D that on input c and some string w , outputs m if $F(w) = 1$. In other words, instead of the key being a unique string, the key is *any* string w that satisfies a certain condition. Witness encryption can be used for other applications. For example, you could encrypt a message to future members of humanity, that can be decrypted only using a valid proof of the Riemann Hypothesis.

21.2 Deniable encryption

Here is another scenario that is seemingly not covered by our current tools. Suppose that Alice uses a public key system (G, E, D) to encrypt a message m by computing $c = E_e(m; r)$ and sending c to Bob that will compute $m = D_d(c)$. The ciphertext is intercepted by Bob's archenemy Freddie Baskerville Ignatius (or FBI for short) who has the means to force Alice to reveal the message and as proof reveal the randomness used in encryption as well. Could Alice find, for any choice of m' , some string r' that is pseudorandom and still c equals $E_e(m', r')$? An encryption scheme with this property is called *deniable*, since we Alice can deny she sent m and claim she sent m' instead.¹

21.3 Functional encryption

It's not just individuals that don't have all their needs met by our current tools. Think of a large enterprise that uses a public key encryption (G, E, D) . When a ciphertext $c = E_e(m)$ is received by the enterprise's servers, it needs to be decrypted using the secret key d . But this creates a single point of failure. It would be much better if we could create a "weakened key" d_1 that, for example, can only decrypt messages related to sales that were sent in the date range X-Y, a key d_2 that can only decrypt messages that contain certain keywords, or maybe a key d_3 that only allows to detect whether the message encoded by a particular ciphertext satisfies a certain regular expression.

¹ One could also think of a deniable witness encryption, and so if Janine in the scenario above is forced to open the ciphertexts she sent by reveal the randomness used to create them, she can credibly claim that she didn't encrypt her knowledge of the conspiracy, but merely wanted to make sure that her family secret recipe for pumpkin pie is not lost when she passes away.

This will allow us to give the key d_1 to the manager of the sales department (and not worry about her taking the key with her if she leaves the company), or more generally give every employee a key that corresponds to his or her role. Furthermore, if the company receives a subpoena for all emails relating to a particular topic, it could give out a cryptographic key that reveals precisely these emails and nothing else. It could also run a spam filter on encrypted messages without needing to give the server performing this filter access to the full contents of the messages (and so perhaps even outsource spam filtering to a different company).

The general form of this is called a *functional encryption*. The idea is that for every function $f : \{0,1\}^* \rightarrow \{0,1\}^*$ we can create a decryption key d_f such that on input $c = E_e(m)$, $D_{d_f}(c) = f(m)$ but d_f cannot be used to gain any other information on the message except for $f(m)$, and even if several parties holding d_{f_1}, \dots, d_{f_k} collude together, they can't learn more than simply $f_1(m), \dots, f_k(m)$. Note that using fully homomorphic encryption we can easily transform an encryption of m to an encryption of $f(m)$ but what we want here is the ability to *selectively decrypt* only some information about the message.

The formal definition of functional encryption is the following:

Definition 21.1 — Functional Encryption. A tuple $(G, E, D, KeyDist)$ is a *functional encryption scheme* if:

- For every function $f : \{0,1\}^\ell \rightarrow \{0,1\}$, if $(d, e) = G(1^n)$ and $d_f = KeyDist(d, f)$, then for every message m , $D_{d_f}(E_e(m)) = f(m)$.
- Every efficient adversary Eve wins the following game with probability at most $1/2 + negl(n)$:
 1. We generate $(d, e) \leftarrow_R G(1^n)$.
 2. Eve is given e and for $i = 1, \dots, T = poly(n)$ repeatedly chooses f_i and receives d_{f_i} .
 3. Eve chooses two messages m_0, m_1 such that $f_i(m_0) = f_i(m_1)$ for all $i = 1, \dots, T$.
 4. For $b \leftarrow_R \{0,1\}$, Eve receives $c^* = E_e(m_b)$ and outputs b' .
 5. Eve *wins* if $b' = b$.

21.4 The software patch problem

It's not only exotic forms of encryption that we're missing. Here is another application that is not yet solved by the above tools. From time to time software companies discover a vulnerability in their products. For example, they might discover that if fed an input x of some particular form (e.g., satisfying a regular expression R) to a server running their software could give an adversary unlimited access to it. In such a case, you might want to release a patch that modifies the software to check if $R(x) = 1$ and if so rejects the input. However the fear is that hackers who didn't know about the vulnerability before could discover it by examining the patch and then use it to attack the customers who are slow to update their software. Could we come up for a regular expression R with a program P such that $P(x) = 1$ if and only if $R(x) = 1$ but examining the code of P doesn't make it any easier to find some x satisfying R ?

21.5 Software obfuscation

All these applications and more could in principle be solved by a single general tool known as *virtual black-box (VBB) secure software obfuscation*. In fact, such an obfuscation is a general tool that can also be directly used to yield public key encryption, fully homomorphic encryption, zero knowledge proofs, secure function evaluation, and many more applications.

We will now give the definition of VBB secure obfuscation and prove the central result about it, which is unfortunately that secure VBB obfuscators do not exist. We will then talk about the relaxed notion of *indistinguishability obfuscators* (IO) - this object turns out to be good enough for many of the above applications and whether it exists is one of the most exciting open questions in cryptography at the moment. We will survey some of the research on this front.

Lets define a *compiler* to be an efficient (i.e., polynomial time) possibly probabilistic map \mathcal{O} that takes a Boolean circuit C on n bits of input and outputs a Boolean circuit C' that also takes n input bits and computes the same function; i.e., $C(x) = C'(x)$ for every $x \in \{0,1\}^n$. (If \mathcal{O} is probabilistic then this should happen for every choice of its coins.) This might seem a strange definition, since it even allows the trivial compiler $\mathcal{O}(C) = C$. That is OK, since later we will require additional properties such as the following:

Definition 21.2 — VBB secure obfuscation. A compiler \mathcal{O} is a *virtual black box (VBB) secure obfuscator* if it satisfies the following property: for every efficient adversary A mapping $\{0,1\}^*$ to $\{0,1\}$, there exists an efficient simulator S such that for every circuit C the following random variables are computationally indistinguishable:

- $A(\mathcal{O}(C))$
- $S^C(1^{|C|})$ where by this we mean the output of S when it is given the length of C and access to the function $x \mapsto C(x)$ as a black box (aka oracle access).

(Note that the distributions above are of a single bit, and so being indistinguishable simply means that the probability of outputting 1 is equal in both cases up to a negligible additive factor.)

21.6 Applications of obfuscation

The writings of Diffie and Hellman, James Ellis, and others that thought of public key encryption, shows that one of the first approaches they considered was to use obfuscation to transform a private-key encryption scheme into a public key one. That is, given a private key encryption scheme (E, D) we can transform it to a public key encryption scheme (G, E', D) by having the key generation algorithm select a private key $k \leftarrow_R \{0,1\}^n$ that will serve as the decryption key, and let the encryption key e be the circuit $\mathcal{O}(C)$ where \mathcal{O} is an obfuscator and C is a circuit mapping c to $D_k(d)$. The new encryption algorithm E' takes e and c and simply outputs $e(c)$.

These days we know other approaches for obtaining public key encryption, but the obfuscation-based approach has significant additional flexibility. To turn this into a fully homomorphic encryption, we simply publish the obfuscation of $c, c' \mapsto D_k(c)NAND_k(c')$. To turn this into a functional encryption, for every function f we can define d_f as the obfuscation of $c \mapsto f(D_k(c))$.

We can also use obfuscation to get a witness encryption, to encrypt a message m to be opened using any w such that $F(w) = 1$, we can obfuscate the function that maps w to m if $F(w) = 1$ and outputs error otherwise. To solve the patch problem, for a given regular expression we can obfuscate the function that maps x to $R(x)$.

21.7 Impossibility of obfuscation

So far, we've learned that in cryptography no concept is too fantastic to be realized. Unfortunately, VBB secure obfuscation **is an exception**:

Theorem 21.3 — Impossibility of Obfuscation. Under the PRG assumption, there does not exist a VBB secure obfuscating compiler.

21.7.1 Proof of impossibility of VBB obfuscation

We will now show the proof of [Theorem 21.3](#). For starters, note that obfuscation is trivial for *learnable* functions. That is, if F is a function such that given black-box access to F we can recover a circuit that computes it, then we can obfuscate it. Given a circuit C , the obfuscator \mathcal{O} will simply use it as a black box to learn a circuit C' that computes the same function and output it. Since \mathcal{O} itself only uses black box access to C , it can be trivially simulated perfectly. (Verifying that this is indeed the case is a good way to make sure you followed the definition.)

However, this is not so useful, since it's not hard to see that all the examples above where we wanted to use obfuscation involved functions that were unlearnable. But it already suggests that we should use an unlearnable function for our negative result. Here is an extremely simple unlearnable function. For every $\alpha, \beta \in \{0,1\}^n$, we define $F_{\alpha,\beta} : \{0,1\}^n \rightarrow \{0,1\}^n$ to be the function that on input x outputs β if $x = \alpha$ and otherwise outputs 0^n .

Given black box access for this function for a random α, β , it's extremely unlikely that we would hit α with a polynomial number of queries and hence will not be able to recover β and so in particular will not be able to learn a circuit that computes $F_{\alpha,\beta}$.²

This function already yields a counterexample for a stronger version of the VBB definition. We define a *strong VBB obfuscator* to be a compiler \mathcal{O} that satisfies the above definition for adversaries that can output not just one bit but an arbitrary long string. We can now prove the following:

Lemma 21.4 There does not exist a strong VBB obfuscator.

Proof. Suppose towards a contradiction that there exists a strong VBB obfuscator \mathcal{O} . Let $F_{\alpha,\beta}$ be defined as above, and let A be the adversary that on input a circuit C' simply outputs C' . We claim that for every S there exists some α, β and an efficient algorithm $D_{\alpha,\beta}$

² Pseudorandom functions can be used to construct examples of functions that are unlearnable in the much stronger sense that we cannot achieve the machine learning goal of outputting some circuit that *approximately predicts* the function.

$$\left| \mathbb{P}[D_{\alpha,\beta}(A(\mathcal{O}(F_{\alpha,\beta})) = 1] - \mathbb{P}[D_{\alpha,beta}(S^{F_{\alpha,\beta}}(1^{10n})) = 1] \right| > 0.9 \quad (*)$$

these probabilities are over the coins of \mathcal{O} and the simulator S .

Note that we identify the function $F_{\alpha,\beta}$ with the obvious circuit of size at most $10n$ that computes it.

Clearly $(*)$ implies that these two distributions are not indistinguishable, and so proving $(*)$ will finish the proof. The algorithm $D_{\alpha,\beta}$ on input a circuit C' will simply output 1 iff $C'(\alpha) = \beta$. By the definition of a compiler and the algorithm A , for every α, β ,

$$\mathbb{P}[D_{\alpha,\beta}(A(\mathcal{O}(F_{\alpha,\beta})) = 1] = 1.$$

On the other hand, for $D_{\alpha,\beta}$ to output 1 on $C' = S^{F_{\alpha,\beta}}(1^{10n})$, it must be the case that $C'(\alpha) = \beta$. We claim that there exists some α, β such that this will happen with negligible probability. Indeed, assume S makes $T = \text{poly}(n)$ queries and pick α, β independently and uniformly at random from $\{0,1\}^n$. For every $i = 1, \dots, T$, let E_i be the event that the i^{th} query of S is the first in which it gets a response other than 0^n . The probability of E_i is at most 2^{-n} because as long as S got all responses to be 0^n , it got no information about α and so the choice of S 's i^{th} query is independent of α which is chosen at random in $\{0,1\}^n$. By a union bound, the probability that S got any response other than 0^n is negligible. In which case if we let C' be the output of S and let $\beta' = C'(\alpha)$, then β' is independent of β and so the probability that they are equal is at most 2^{-n} . ■

The adversary in the proof of [Lemma 21.4](#) does not seem very impressive. After all, it merely printed out its input. Indeed, the definition of strong VBB security might simply be an overkill, and “plain” VBB is enough for almost all applications. However, as mentioned above, plain VBB is impossible to achieve as well. We’ll prove a slightly weaker version of [Theorem 21.3](#):

Theorem 21.5 — Impossibility of Obfuscation from FHE. If fully homomorphic encryption exists then there is no VBB secure obfuscating compiler.

(To get the original theorem from this, note that if VBB obfuscation exist then we can transform any private key encryption into a fully homomorphic public key encryption.)

Proof. Let $(G, E, D, EVAL)$ be a fully homomorphic encryption scheme. For strings $d, e, c, \alpha, \beta, \gamma$, we will define the function $F_{e,c,\alpha,\beta,\gamma}$ as follows: for inputs of the form $00x$, it will output β if and only if $x = \alpha$, and otherwise output 0^n . For inputs of the form $01c'$, it will

output γ iff $D_d(c') = \beta$ and otherwise output 0^n . And for the input 1^n , it will output c . For all other inputs it will output 0^n .

We will use this function family where d, e are the keys of the FHE, and $c = E_e(\alpha)$. We now define our adversary A . On input some circuit C' , A will compute $c = C'(1^n)$ and let C'' be the circuit that on input x outputs $C'(00x)$. It will then let $c'' = \text{EVAL}_e(C'', c)$. Note that if c is an encryption of α and C' computes $F = F_{d,e,c,\alpha,\beta,\gamma}$ then c'' will be an encryption of $F(00\alpha) = \beta$. The adversary A will then compute $\gamma' = C'(01c')$ and output γ_1 .

We claim that for every simulator S , there exist some tuple $(d, e, c, \alpha, \beta, \gamma)$ and a distinguisher D such that

$$\left| \mathbb{P}[D(A(\mathcal{O}(F_{d,e,c,\alpha,\beta,\gamma}))) = 1] - \mathbb{P}[S^{F_{d,e,c,\alpha,\beta,\gamma}}(1^{|F_{d,e,c,\alpha,\beta,\gamma}|})] \right| \geq 0.1 \quad (21.1)$$

Indeed, the distinguisher D will depend on γ and on input a bit b will simply output 1 iff $b = \gamma_1$. Clearly, if (d, e) are keys of the FHE and $c = E_e(\alpha)$ then no matter what circuit C' the obfuscator \mathcal{O} outputs on input $F_{d,e,c,\alpha,\beta,\gamma}$, the adversary A will output γ_1 on C' and hence D will output 1 with probability one on A 's output. ■

In contrast if we let S be a simulator and generate $(d, e) = G(1^n)$, pick α, β, γ independently at random in $\{0, 1\}^n$ and let $c = E_e(\alpha)$, we claim that the probability that S will output γ_1 will be equal to $1/2 \pm \text{negl}(n)$. Indeed, suppose otherwise, and define the event E_i to be that the i^{th} query is the first query (apart from the query 1^n whose answer is c) on which S receives an answer other than 0^n . Now there are two cases: > **Case 1:** The query is equal to 00α . > **Case 2:** The query is equal to $01c'$ for some c' such that $D_d(c') = \beta$. > Case 2 only happens with negligible probability because if S only received the value e (which is independent of β) and did not receive any other non 0^n response up to the i^{th} point then it did not learn any information about β . Therefore the value β is independent of the i^{th} query and the probability that it decrypts to β is at most 2^{-n} . > Case 1 only happens with negligible probability because otherwise S is an algorithm that on input an encryption of α (and a bunch of answers of the form 0^n , which are of course not helpful) manages to output α with non-negligible probability, hence violating the CPA security of the encryption scheme. > Now if neither case happens, then S does not receive any information about γ , and hence the probability that its output is γ_1 is at most $1/2$.

P This proof is simple but deserves a second read. A crucial point here is to use FHE to allow the adversary to essentially “feed C' to itself” so it can obtain from an encryption of α an encryption of β , even though that would not be possible using black box access only.

21.8 Indistinguishability obfuscation

The proof can be generalized to give private key encryption for which the transformation to public key encryption would be insecure, and many other such constructions. So, this result might (and indeed to a large extent did) seem like a death blow to general-purpose obfuscation. However, already in that paper we noticed that there was a variant of obfuscation that we could not rule out, and this is the following:

Definition 21.6 — Indistinguishability Obfuscation. We say a compiler \mathcal{O} is an *indistinguishability obfuscator* (IO) if for every two circuits C, C' that have the same size and compute the same function, the random variables $\mathcal{O}(C)$ and $\mathcal{O}(C')$ are computationally indistinguishable.

It is a good exercise to understand why the proof of the impossibility result above does not apply to rule out IO. Nevertheless, a reasonable guess would be that:

1. IO is impossible to achieve.
2. Even if it was possible to achieve, it is not good enough for most of the interesting applications of obfuscation.

However, it turns out that this guess is (most likely) wrong. New results have shown that IO is extremely useful for many applications, including those outlined above. They also gave some evidence that it might be possible to achieve. We’ll talk about those works in the next lecture.

22

More obfuscation, exotic encryptions

Fully homomorphic encryption is an extremely powerful notion, but it does not allow us to obtain fine control over the access to information. With the public key you can do all sorts of computation on the encrypted data, but you still do not learn it, while with the private key you learn everything. But in many situations we want *fine grained access control*: some people should get access to some of the information for some of the time. This makes the “all or nothing” nature of traditional encryptions problematic. While one could still implement such access control by interacting with the holder(s) of the secret key, this is not always possible.

The most general notion of an encryption scheme allowing fine control is known as *functional encryption*, as was described in the previous lecture. This can be viewed as an object dual to Fully Homomorphic Encryption, and incomparable to it. For every function f , we can construct an f -restricted decryption key d_f that allows recovery of $f(m)$ from an encryption of m but not anything else.

In this lecture we will focus on a weaker notion known as *identity based encryption (IBE)*. Unlike the case of full fledged functional encryption, there are fairly efficient constructions known for IBE.

22.1 Slower, weaker, less secure

In a sense, functional encryption or IBE is all about selective *leaking* of information. That is, in some sense we want to modify an encryption scheme so that it actually is “less secure” in some very precise sense, so that it would be possible to learn something about the plaintext even without knowing the (full) decryption key.

There is actually a history of cryptographic technique meant to

support such operations. Perhaps the “mother” of all such “quasi encryption” schemes is the modular exponentiation operation $x \mapsto g^x$ for some discrete group G . The map $x \mapsto g^x$ is not exactly an encryption of x - for one thing, we don’t know how to decrypt it. Also, as a deterministic map, it cannot be semantically secure. Nevertheless, if x is random, or even high entropy, in groups such as cyclic subgroup of a multiplicative group modulo some prime, we don’t know how to recover x from g^x . However, given g^{x_1}, \dots, g^{x_k} and a_1, \dots, a_k we can find out if $\sum a_i x_i = 0$, and this can be quite useful in many applications.

More generally, even in the private key setting, people have studied encryption schemes such as

- **Deterministic encryption** : an encryption scheme that maps x to $E(x)$ in a deterministic way. This cannot be semantically secure in general but can be good enough if the message x has high enough entropy or doesn’t repeat and allows to check if two encryptions encrypt the same object. (We can also do this by publishing a hash of x under some secret salt.)
- **Order preserving encryption**: is an encryption scheme mapping numbers in some range $\{1, \dots, N\}$ to ciphertexts so that given $E(x)$ and $E(y)$ one can efficiently compare whether $x < y$. This is quite problematic for security. For example, given $\text{poly}(t)$ random such encryptions you can more or less know where they lie in the interval up to $(1 \pm 1/t)$ multiplicative factor..
- **Searchable encryption**: is a generalization of deterministic encryption that allows some more sophisticated searchers (such as not only exact match).

Some of these constructions can be quite efficient. In particular the system [CryptDB](#) developed by Popa et al uses these kinds of encryptions to automatically turn a SQL database into one that works on encrypted data and still supports the required queries. However, the issue of how dangerous the “leakage” can be is somewhat subtle. See this [paper](#) and [blog post](#) claiming weaknesses in practical use cases for CryptDB, as well as this [response](#) by the CryptDB authors.

While the constructions of IBE and functional encryption often use maps such as $x \mapsto g^x$ as subroutines, they offer a stronger control over the leakage in the sense that, in the absence of publishing a (restricted) decryption key, we always get at least CPA security.

22.2 How to get IBE from pairing based assumptions.

The standard exponentiation mapping $x \mapsto g^x$ allows us to compute *linear functions in the exponent*. That is, given any linear map L of the form $L(x_1, \dots, x_k) = \sum a_i x_i$, we can efficiently compute the map $g^{x_1}, \dots, g^{x_k} \mapsto g^{L(x_1, \dots, x_k)}$. But can we do more? In particular, can we compute *quadratic* functions? This is an issue, as even computing the map $g^x, g^y \mapsto g^{xy}$ is exactly the Diffie Hellman problem that is considered hard in many of the groups we are interested in.

Pairing based cryptography begins with the observation that in some elliptic curve groups we can use a map based on the so called Weil or Tate pairings. The idea is that we have an efficiently computable isomorphism from a group G_1 to a group G_2 mapping g to \hat{g} such that we can efficiently map the elements g^x and g^y to the element $\varphi(g^x, g^y) = \hat{g}^{xy}$. This in particular means that given g^{x_1}, \dots, g^{x_k} we can compute $\hat{g}^{Q(x_1, \dots, x_k)}$ for every quadratic Q . Note that we cannot repeat this to compute, say, degree 4 functions in the exponent, since we don't know how to invert the map φ .

The **Pairing Diffie Hellman Assumption** is that we can find two such groups G_1, G_2 and generator g for G such that there is no efficient algorithm A that on input g^a, g^b, g^c (for random $a, b, c \in \{0, \dots, |\mathbb{G}| - 1\}$) computes \hat{g}^{abc} . That is, while we can compute a quadratic in the exponent, we can't compute a cubic.

We now show an IBE construction due to Boneh and Franklin¹ how we can obtain from the pairing diffie hellman assumption an identity based encryption:

- **Master key generation:** We generate G_1, G_2, g as above, choose a at random in $\{0, \dots, |\mathbb{G}| - 1\}$. The master private key is a and the master public key is $G_1, G_2, g, h = g^a$. We let $H : \{0, 1\}^* \rightarrow G_1$ and $H' : G_2 \mapsto \{0, 1\}^\ell$ be two hash functions modeled as random oracles.
- **Key distribution:** Given an arbitrary string $id \in \{0, 1\}^*$, we generate the decryption key corresponding to id , as $d_{id} = H(id)^a$.
- **Encryption:** To encrypt a message $m \in \{0, 1\}^\ell$ given the public parameters and some id id , we choose $c \in \{0, \dots, |\mathbb{G}| - 1\}$, and output $g^c, H'(id \parallel \varphi(h, H(id))^c) \oplus m$
- **Decryption:** Given the secret key d_{id} and a ciphertext h', y , we output $H'(id \parallel \varphi(d_{id}, h')) \oplus x$

Correctness: We claim that $D_{d_{id}}(E_{id}(m)) = m$. Indeed, write

¹ The construction we show was first published in the CRYPTO 2001 conference. The Weil and Tate pairings were used before for cryptographic attacks, but were used for a positive cryptographic result by Antoine Joux in his 2000 paper getting a three-party Diffie Hellman protocol and then Boneh and Franklin used this to obtain an identity based encryption scheme, answering an open question of Shamir. At approximately the same time as these papers, Sakai, Ohgishi and Kasahara presented a paper in the SCIS 2000 conference in Japan showing an identity-based key exchange protocol from pairing. Also Clifford Cocks (who as we mentioned above in the 1970's invented the RSA scheme at GCHQ before R,S, and A did), also came up in 2001 with a different identity-based encryption scheme using the quadratic residuosity assumption.

$h_{id} = H(id)$ and let $b = \log_g h_{id}$. Then an encryption of m has the form $h' = g^c, H'(id\| \varphi(g^a, h_{id})^c) \oplus m$, and so the second term is equal to $H'(id\| \hat{g}^{abc}) \oplus m$. However, since $d_{id} = h_{id}^a = g^{ab}$, we get that $\varphi(h', d_{id}) = \hat{g}^{abc}$ and hence decryption will recover the message. QED

Security: To prove security we need to first present a *definition* of IBE security. The definition allows the adversary to request keys corresponding to arbitrary identities, as long as it does not ask for keys corresponding to the target identity it wants to attack. There are several variants, including CCA type of security definitions, but we stick to a simple one here:

Definition: An IBE scheme is said to be CPA secure if every efficient adversary Eve wins the following game with probability at most $1/2 + negl(n)$:

- The keys are generated and Eve gets the master public key.
- For $i = 1, \dots, T = poly(n)$, Eve chooses an identity $id_i \in \{0,1\}^*$ and gets the key d_{id} .
- Eve chooses an identity $id^* \notin \{id_1, \dots, id_T\}$ and two messages m_0, m_1 .
- We choose $b \leftarrow_R \{0,1\}$ and Eve gets the encryption of m_b with respect to the identity id^* .
- Eve outputs b' and *wins* if $b' = b$.

Theorem: If the pairing Diffie Hellman assumption holds and H, H' are random oracles, then the scheme above is CPA secure.

Proof: Suppose for the sake of contradiction that there exists some time $T = poly(n)$ adversary A that succeeds in the IBE-CPA with probability at least $1/2 + \epsilon$ for some non-negligible ϵ . We assume without loss of generality that whenever A makes a query to the key distribution function with id id or a query to H' with prefix id , it had already previously made the query id to H . (A can be easily modified to have this behavior)

We will build an algorithm B that on input $\mathbb{G}_1, \mathbb{G}_2, g, g^a, g^b, g^c$ will output \hat{g}^{abc} with probability $poly(\epsilon, 1/T)$.

The algorithm B will guess $i_0, j_0 \leftarrow_R \{1, \dots, T\}$ and simulate A “in its belly” giving it the public key g^a , and act as follows:

- When A makes a query to H with id , then for all but the i_0^{th} queries, B will choose a random $b_{id} \in \{0, \dots, |\mathbb{G}|\}$ (as usual we'll assume $|\mathbb{G}|$ is prime), choose $e_{id} = g^{b_{id}}$ and define $H(id) = e_{id}$.

Let id_0 be the i_0^{th} query A made to the oracle. We define $H(id_0) = g^b$ (where g^b is the input to B - recall that B does not know b .)

- When A makes a query to the key distribution oracle with id then if $id \neq id_0$ then B will then respond with $d_{id} = (g^a)^{b_{id}}$. If $id = id_0$ then B aborts and fails.
- When A makes a query to the H' oracle with input $id' \parallel \hat{h}$ then for all but the j_0^{th} query B answers with a random string in $\{0,1\}^\ell$. In the j_0^{th} query, if $id' \neq id_0$ then B stops and fails. Otherwise, it outputs \hat{h} .
- B does stops the simulation and fails if we get to the challenge part.

It might seem weird that we stop the simulation before we reach the challenge part, but the correctness of this reduction follows from the following claim:

Claim: In the actual attack game, with probability at least $\epsilon/10$ A will make the query $id_* \parallel \hat{g}^{abc}$ to the H' oracle, where $H(id_*) = g^b$ and the public key is g^a .

Proof: If A does not make this query then the message in the challenge is XOR'ed by a completely random string and A cannot distinguish between m_0 and m_1 in this case with probability better than $1/2$. QED

Given this claim, to prove the theorem we just need to observe that, assuming it does not fail, B provides answers to A that are identically distributed to the answers A receives in an actual execution of the CPA game, and hence with probability at least $\epsilon/(10T^2)$, B will guess the query i_0 when A queries $H(id_*)$ and set the answer to be g^b , and then guess the query j_0 when A queries $id_* \parallel \hat{g}^{abc}$ in which case B 's output will be correct. QED

22.3 Beyond pairing based cryptography

Boneh and Silverberg asked the question of whether we could go beyond quadratic polynomials and get schemes that allow us to compute higher degree. The idea is to get a *multilinear map* which would be a set of isomorphic groups $\mathbb{G}_1, \dots, \mathbb{G}_d$ with generators g_1, \dots, g_d such that we can map g_i^a and g_j^b to g_{i+j}^{ab} .

This way we would be able to compute any degree d polynomial in the exponent given $g_1^{x_1}, \dots, g_1^{x_k}$.

We will now show how using such a multilinear map we can get a construction for a witness encryption scheme. We will only show the construction, without talking about the security definition, the assumption, or security reductions.

Given some circuit $C : \{0, 1\}^n \rightarrow \{0, 1\}$ and some message x we want to “encrypt” x in a way that given w such that $C(w) = 1$ it would be possible to decrypt x , and otherwise it should be hard. It should be noted that the encrypting party itself does not know any such w and indeed (as in the case of the proof of Riemann hypothesis) might not even know if such a w exists. The idea is the following. We use the fact that the Exact Cover problem is NP complete to map C into collection of subsets S_1, \dots, S_m of the universe U (where $m, |U| = \text{poly}(|C|, n)$) such that there exists w with $C(w) = 1$ if and only if there exists d sets S_{i_1}, \dots, S_{i_d} that are a partition of U (i.e., every element in U is covered by exactly one of these sets), and moreover there is an efficient way to map w to such a partition and vice versa. Now, to encrypt the message x we take a degree d instance of multilinear maps $(\mathbb{G}_1, \dots, \mathbb{G}_d, g_1, \dots, g_d)$ (with all groups of size p) and choose random $a_1, \dots, a_{|U|} \leftarrow_R \{0, \dots, p - 1\}$. We then output the ciphertext $g_1^{\prod_{j \in S_1} a_j}, \dots, g_1^{\prod_{j \in S_m} a_j}, H(g_d^{\prod_{j \in U} a_j}) \oplus x$. Now, given a partition S_{i_1}, \dots, S_{i_d} of the universe U , we can use the multilinear operations to compute $g_d^{\prod_{j \in U} a_j}$ and recover the message. Intuitively, since the numbers are random, that would be the only way to come up with computing this value, but showing that requires formulating precise security definitions for both multilinear maps and witness encryption and of course a proof.

The first candidate construction for a multilinear map was given by [Garg, Gentry and Halevi](#). It is based on computational questions on lattices and so (perhaps not surprisingly) it involves significant complications due to *noise*. At a very high level, the idea is to use a fully homomorphic encryption scheme that can evaluate polynomials up to some degree d , but release a “hobbled decryption key” that contains just enough information to provide what’s known as a *zero test*: check if an encryption is equal to zero. Because of the homomorphic properties, that means that we can check given encryptions of x_1, \dots, x_n and some degree d polynomial P , whether $P(x_1, \dots, x_d) = 0$. Moreover, the notion of security this and similar construction satisfy is rather subtle and indeed not fully understood. Constructions of indistinguishability obfuscators are built based on this idea, but are significantly more involved than the construction of a witness encryption. One central tool they use is the observation that FHE reduces the task of obfuscation to essentially obfuscating

a decryption circuit, which can often be rather shallow. But beyond that there is significant work to be done to actually carry out the obfuscation.

23

Anonymous communication

Encryption is meant to protect the contents of communication, but sometimes the bigger secret is that the communication existed in the first place. If a whistleblower wants to leak some information to the New York Times, the mere fact that she sent an email would reveal her identity. There are two main concepts aimed at achieving anonymity:

- *Anonymous routing* is about ensuring that Alice and Bob can communicate without that fact being revealed.
- *Steganography* is about having Alice and Bob hiding an encrypted communication in the context of an seemingly innocuous conversation.

23.1 Steganography

The goal in a stegnaographic communication is to hide cryptographic (or non cryptographic) content without being detected. The idea is simple: let's start with the *symmetric case* and assume Alice and Bob share a shared key k and Alice wants to transmit a bit b to Bob. We assume that Alice and has a choice of t words w_1, \dots, w_t that would be reasonable for her to send at this point in the conversation. Alice will choose a word w_i such that $f_k(w_i) = b$ where $\{f_k\}$ is a pseudorandom function collection. With probability $1 - 2^{-t}$ there will be such a word. Bob will decode the message using $f_k(w_i)$. Alice and Bob can use an error correcting code to compensate for the probability 2^{-t} that Alice is forced to send the wrong bit.

In the *public key setting*, suppose that Bob publishes a public key e for an encryption scheme that has *pseudorandom ciphertexts*. That is,

to a party that does not know the key, an encryption is indistinguishable from a random string. To send some message m to Bob, Alice computes $c = E_e(m)$ and transmits it to Bob one bit at a time. Given the t words w_1, \dots, w_t , to transmit the bit c_j Alice chooses a word w_i such that $H(w_i) = c_j$ where $H : \{0,1\}^* \rightarrow \{0,1\}$ is a hash function modeled as a random oracle. The distribution of words output by Alice w^1, \dots, w^ℓ is uniform conditioned on $(H(w^1), \dots, H(w^\ell)) = c$. But note that if H is a random oracle, then $H(w^1), \dots, H(w^\ell)$ is going to be uniform, and hence indistinguishable from c .

23.2 Anonymous routing

- **Low latency communication:** Aqua, Crowds, LAP, ShadowWalker, Tarzan, Tor
- **Message at a time, protection against timing / traffic analysis:** Mix-nets, e-voting, Dining Cryptographer network (DC net), Dissent, Herbivore, Riposte

23.3 Tor

Basic arhitecture. Attacks

23.4 Telex

23.5 Riposte

24

Ethical, moral, and policy dimensions to cryptography

This will not be a lecture but rather a discussion on some of the questions that arise from cryptography. I would like you to read some of the sources below (and maybe others) and reflect on the following questions:

The discussion is often framed as weighing privacy against security, but I encourage you to look critically at both issues. It is often instructive to try to compare the current situation with both the historical past as well as some ideal desired world. It is also worthwhile to consider cryptography in the broader contexts. Some people on both the pro regulation and anti regulation camps exaggerate the role of cryptography.

On one hand, cryptography is likely not to bring about the “crypto anarchy” regime hoped for in the crypto anarchist manifesto. For example, more than the growth of bitcoin, we are seeing a turn away from cash into credit cards and other forms of much more traceable and *less* anonymous forms of payments (interestingly, these forms of payments are often enabled by cryptography). On the other hand, despite the fears raised by government agencies of “going dark” there are powerful commercial incentives to collect vast amounts of data and store them at search-warrant friendly servers. Clearly technology is shifting the landscape of relationships among individuals, as well as between individuals and large organizations and governments. Cryptography is an important component in these technologies but not the only one, and more than that, the ways technologies end up *used* often has more to do with social and commercial factors than with the technologies themselves.

All that said, significant changes often pose non trivial dangers, and it is important to have an informed and reasoned discussion of the ways cryptography can help or harm the general and private

good.

Some questions that are worth considering are:

- Is communicating privately a basic **human right**? Should it extend to communicating at a distance? Should this be absolute privacy that cannot be violated even with a legal warrant? If there was a secure way to implement wiretapping only with a legal warrant, would it be morally just?
- Is privacy a basic good in its own right? Or a necessary condition for the freedom of expression, and peaceful assembly and association?
- Are we less or more secure today than in the past? In what ways did the balance between government and individuals shift in the last few decades? Do governments have more or less data and tools for monitoring individuals at their disposal? Do individuals and non-governmental groups have more or less ability to inflict harm (and hence need to be protected against)?
- Do we have more or less privacy today than in the past? Do cryptography regulation play a big part in that?
- What would be the balance between security and privacy in an ideal world?
- Is the focus on encryption misguided in that the main issue affecting privacy and security is the so called *meta data*? Can cryptographic techniques protect such meta data? Even if they could, is there a commercial interest in doing so?
- One argument against the regulation of cryptography is that, given the mathematics of cryptography is not secret, the “bad guys” will always be able to access it. Is this a valid argument? Note that similar arguments are made in the context of gun control. Also, perhaps the “true dissidents” will also be able to access cryptography as well and so regulation will effect the masses or “run of the mill” private good and not-so-good citizens?
- What would be the practical impact of regulations forbidding the use of end-to-end crypto without access by governments?
- Rogaway argues that cryptography is inherently political, and research should acknowledge this and be directed at achieving beneficial political goals. Has cryptography research failed the public? What more could be done?
- Are some cryptographic (or crypto related) tools inherently

morally problematic? Rogaway suggests that this may be true for fully homomorphic encryption and differential privacy- do you agree?

- What are the most significant scenarios where cryptography can impact positively or negatively? Large scale terror attacks? “Ordinary” crimes (that still claim the lives of many more people than terror attacks)? Attacks against cyber infrastructure or personal data? Political dissidents in oppressive regimes? Mass government or corporate surveillance?
- How are these issues different in the U.S. as opposed to other countries? Is the debate too U.S. centric?

24.1 Reading prior to lecture:

- **Moral Character of Cryptographic Work** - please read at least parts 1-3 (pages 1-30 in the footnoted version) - it's long and should not be taken uncritically, but is a very good and thought provoking read.
- **“Going Dark” Berkman report** - this is a report written by a committee, and as such not as exciting (though arguably more sober) than Rogaway’s paper. Please read at least the introduction and you might also find the personal statements in Appendix A interesting.
- **Digital Equilibrium project** - optional reading - this is a group of very senior current and former officials, in particular in government, and as such would tend to fall on the more “establishment” or “pro regulation” side. Their “foundational paper” has even more of a “written by committee” feel but is still worthwhile reading.
- **Crypto anarchist manifesto** - optional reading - very much not “written by committee” can be an interesting read even if it sounds more like science fiction than describing actual current or near future reality.

24.2 Case studies.

Since such a discussion might be sometimes hard to hold in the abstract, let us consider some actual cases:

24.2.1 *The Snowden revelations*

The impetus for the current iteration of the security vs privacy debate were the [Snowden revelations](#) on the massive scale of surveillance by the NSA on citizens in the U.S. and around the globe. Concurrently, in plain sight, companies such as Apple, Google, Facebook, and others are also collecting massive amounts of information on their users. Some of the backlash to the Snowden revelations was increased pressure on companies to support stronger “end-to-end” encryption such as some data does not reside on companies’ servers, that have become suspect. We’re now seeing some “backlash to the backlash” with law enforcement and government officials around the globe trying to ban such encryption technology or mandate government backdoors.

24.2.2 *FBI vs Apple case*

We’ve mentioned [this case](#) in the past. (I also [blogged](#) about it.) The short summary is that an iPhone belonging to one of the San Bernardino terrorists was found by the FBI. The iPhone’s memory was encrypted by a key k that is obtained as $H(uid \parallel passcode)$ where $passcode$ is the six digit passcode of the user and uid is a secret 128 bit key that is hardwired into the processor. The processor will only allow ten attempts at guessing the passcode before erasing all memory. The FBI wanted Apple’s help in creating a digitally signed software update that essentially run a brute force search over the 10^6 passcodes and output the key k . The software update could be restricted to run only on that particular iPhone. Eventually, the FBI managed to extract the information out of the iPhone without Apple’s help. The method they used is unknown, but it may be possible to physically extract the uid from the processor. It might also be possible to prevent erasure of the memory by disconnecting it from the processor, or rewriting it after erasure. Would such cases change your position on this question?

Some questions that one could ask:

- Given that the FBI had a legal warrant for the information on the iPhone, was it wrong of Apple to refuse to provide the help required?
- Was it wrong for Apple to have designed their iPhone so that they are unable to easily extract information out of it? Should they be required to make sure that such devices can be searched as a result of a legal warrant?

- If the only way for the FBI to get the information was to get Apple's master signature key (that allows to completely break into any iPhone, and even turn it into a recording/surveillance device), would it have been OK for them to do it? Should Apple design their device in a way that even their master signature key cannot break them? Is that even possible, given that software updates are crucial for proper functioning of such devices? (It was recently claimed that Canadian police has had access to the master decryption key of Blackberry since 2010.)

In the San Bernardino case, the utility of breaking into the phone was questioned, given that both perpetrators were killed and there was no evidence of them receiving any assistance. But there are cases where things are more complicated. [Brittney Mills](#) was 29 years old and 8 months pregnant when she was shot and killed in April 2015 in Baton Rouge, Louisiana. Her baby was delivered via emergency C section but also died a week later. There was no sign of forced entry and so it is quite likely she knew her assailant. Her family believes that the clues to her murderer's identity could be found in her iPhone, but since it is locked they have no way of extracting this information. One can imagine other cases as well. Recently a mother found her kidnapped daughter using the [Find my iPhone](#) procedure. It is not hard to conceive of a case where unlocking a phone is the key to saving someone's life. Would such cases change your view of the above questions?

24.2.3 Juniper backdoor case and the OPM break-in

We've also mentioned the case of the [Juniper backdoor case](#). This was a break in to the firewalls of Juniper networks by an unknown party that was crucially enabled by backdoor allegedly inserted by the NSA into the Dual EC pseudorandom generator. (see also [here](#) and [here](#) for more).

Because of the nature of this break in, whomever is responsible for it could have decrypted much of the traffic without leaving any traces, and so we don't know the damage caused, but such hacks can have much more significant consequences than forcing people to change their credit card numbers. When the [federal office of personell management was hacked](#) sensitive information about millions of people who have gone through the security clearance was extracted. This includes fingerprints, extensive personal information from interviews and polygraph sessions, and much more. Such information can help then gain access to more information, whether

it's using the fingerprint to unlock a phone or using the extensive knowledge of social connections, habits and interests to launch very targeted attacks to extract information from particular individuals.

Here one could ask if stronger cryptography, and in particular cryptographic tools that would have enabled an individual to control access to his or her own data, would have helped prevent such attacks.

25

Course recap

It might be worthwhile to recall what we learned in this course:

- Perhaps first and foremost, that it is possible to *mathematically define* what it means for a cryptographic scheme to be secure. In the cases we studied such a definition could always be described as a “security game”. That is, we really define what it means for a scheme to be *insecure* and then a scheme is secure if it is not insecure. The notion of “insecurity” is that there exists some adversarial strategy that succeeds with higher probability than what it should have. We normally don’t limit the *strategy* of the adversary but only his or her *capabilities*: its computational power and the type of access it has to the system (e.g., chosen plaintext, chosen ciphertext, etc..). We also talked how the notion of *secrecy* requires *randomness* and how many real-life failures of cryptosystems amount to faulty assumptions on the sources of randomness.
- We saw the importance of being *conservative* in security definitions. For example, how despite the fact that the notion of chosen ciphertext attack (CCA) security seems too strong to capture any realistic scenario (e.g., when do we let an adversary play with a decryption box?), there are many natural cases where the using a CPA instead of a CCA secure encryption would lead to an attack on the overall protocol.
- We saw how we can prove security by *reductions*. Suppose we have a scheme S that achieves some security notion X (for example, S might be a function that achieves the security notion of being a pseudorandom generator) and we use it to build a scheme T that we want to achieve a security notion Y (for example, we want T to be a message authentication code). Then the way we prove security is that we show how we can transform an adversary B that wins against T in the security game of Y into an adversary A that

wins against S in the security game of X . Typically the adversary A will run B “in its belly” simulating for B the security game Y with respect to T . This can be somewhat confusing so please re-read the last three sentences and make sure you understand this crucial notion.

- We also saw some of the concrete wonderful things we can do in cryptography:
- In the world of *private key cryptography*, we saw that based on the PRG conjecture we can get a CPA secure private key encryption (which in particular has key shorter than message), pseudorandom functions, message authentication codes, CCA secure encryption, commitment schemes, and even zero knowledge proofs for NP complete languages.
- We saw that assuming the existence of *collision resistant hash functions*, we can get message authentication codes (and digital signatures) where the key is shorter than the message. We talked about the heuristic of how we can model hash functions as a *random oracle*, and use that for “proofs of work” in the context of bitcoin and password derivation, as well as many other settings.
- We also discussed practical constructions of private key primitives such as the AES block ciphers, and how such block ciphers are modeled as pseudorandom permutations and how we can use them to get CPA or CCA secure encryption via various modes such as CBC or GCM. We also discussed the Merkle and Davis-Meyer length extension construction for hash functions, and how the Merkle tree construction can be used for secure storage.
- We saw the revolutionary notion of *public key encryption*, that two people can talk without having coordinated in advance. We saw constructions for this based on discrete log (e.g., the Diffie-Hellman protocol), factoring (e.g., the Rabin and RSA trapdoor permutations), and the *learning with errors* (LWE) problem. We saw the notion of digital signatures, and gave several different constructions. We saw how we can use digital signatures to create a “chain of trust” via certificates, and how the TLS protocol, which protects web traffic, works.
- We talked about some advances notions and in particular saw the construction of the surprising concept of a *fully homomorphic encryption (FHE)* scheme which has been rightly called by **Bryan Hayes** “one of the most amazing magic tricks in all of computer science”. Using FHE and zero knowledge proofs, we can get multiparty secure computation, which basically means that in the

setting of interactive protocols between several parties, we can establish a “virtual trusted third party” (or, as I prefer to call it, a “virtual Chuck Norris”).

- We also saw other variants of encryption such as *functional encryption*, *witness encryption* and *identity based encryption*, which allow for “selective leaking” of information. For functional encryption and witness encryption we don’t yet have clean constructions under standard assumptions but only under obfuscation, but we saw how we could get identity based encryption using the random oracle heuristic and the assumption of the difficulty of the discrete logarithm problem in a group that admits an efficient *pairing* operation.
- We talked about the notion of obfuscation, which can be thought as the one tool that if it existed would imply all the others. We saw that virtual black box obfuscation does not exist, but there might exist a weaker notion known as “indistinguishability obfuscation” and we saw how it can be useful via the example of a witness encryption and a digital signature scheme. We mentioned (without proof) that it can also be used to obtain a functional encryption scheme.
- We talked about how quantum computing can change the landscape of cryptography, making lattice based constructions our main candidate for public key schemes.
- Finally we discussed some of the ethical and policy issues that arise in the applications of cryptography, and what is the impact cryptography has now, or can have in the future, on society.

25.1 Some things we did not cover

- We did not cover what is arguably the other “fundamental theorem of cryptography”, namely the equivalence of one-way functions and pseudorandom generators. A one-way function is an efficient map $F : \{0,1\}^* \rightarrow \{0,1\}^*$ that is hard to invert on a random input. That is, for any efficient algorithm A if A is given $y = F(x)$ for uniformly chosen $x \leftarrow_R \{0,1\}^n$, then the probability that A outputs x' with $F(x') = y$ is negligible. It can be shown that one-way functions are *minimal* in the sense that they are *necessary* for a great many cryptographic applications including pseudorandom generators and functions, encryption with key shorter than the message, hash functions, message authentication codes, and many more. (Most of these results are obtained via the work of

Impagliazzo and Luby who showed that if one-way functions do not exist then there is a *universal posterior sampler* in the sense that for every probabilistic process F that maps x to y , there is an efficient algorithm that given y can sample x' from a distribution close to the posterior distribution of x conditioned on $F(x) = y$. This result is typically known as the equivalence of standard one-way functions and distributional one-way functions.) The fundamental result of Hastad, Impagliazzo, Levin and Luby is that one-way functions are also *sufficient* for much of private key cryptography since they imply the existence of pseudorandom generators.

- Related to this, although we mentioned this briefly, we did not go in depth into “Impagliazzo’s Worlds” of algorithmica, heuristica, pessiland, minicrypt, cryptomania (and the new one of “obfustopia”). If this piques your curiosity, please read [this 1995 survey](#).
- We did not go in detail into the design of private key cryptosystems such as the AES. Though we discussed modes of operation of block ciphers we did not go into a full description of all modes that are used in practice. We also did not discuss cryptanalytic techniques such as linear and differential cryptanalysis. We also not discuss all technical issues that arise with length extention and padding of encryptions in practice. In particular we did not talk
- While we talked about bitcoin, the TLS protocol, two factor authentication systems, and some aspects of pretty good privacy, we restricted ourselves to abstractions of these systems and did not attempt a full “end to end” analysis of a complete system. I do hope you have learned the tools that you’d be able to understand the full operation of such a system if you need to.
- While we talked about Shor’s algorithm, the algorithm people actually use today to factor numbers is the *number field sieve*. It and its predecessor, the quadratic sieve, are well worth studying. The (freely available online) [book of Shoup](#) is an excellent source not just for these algorithms but general algorithmic group/number theory.
- We talked about some attacks on practical systems, but there many other attacks that teach us important lessons, not just about these particular systems, but also about security and cryptography in general (as well some human tendencies to repeat certain types of mistakes).

25.2 *What I hope you learned*

I hope you got an appreciation for cryptography, and an understanding of how it can surprise you both in the amazing security properties it can deliver, as well in the subtle, but often devastating ways, that it can fail. Beyond cryptography, I hope you got out of this course the ability to think a little differently- to be paranoid enough to see the world from the point of view of an adversary, but also the lesson that sometimes if something sounds crazy but is not downright impossible it might just be feasible.

But if these philosophical ramblings don't speak to you, as long as you know the difference between CPA and CCA and I won't catch you reusing a one-time pad, you should be in good shape :)

I did not intend this course to teach you how to implement cryptographic algorithms, but I do hope that if you need to use cryptography at any point, you now have the skills to read up what's needed, and be able to argue intelligently about the security of real-world systems. I also hope that you have now sufficient background to not be scared by the technical jargon and the abundance of adjectives in cryptography research papers, and be able to read up on what you need to follow any paper that is interesting to you.

Mostly, I just hope you enjoyed this last term and felt like this course was a good use of your time. I certainly did.

Bibliography