CS306: Introduction to IT Security

Fall 2020

Lecture 8: Crypto Light

Instructor: **Nikos Triandopoulos**

November 3, 2020

STEVENS INSTITUTE of TECHNOLOGY

THE INNOVATION UNIVERSITY®

Assignment Project Exam Help

https://powcoder.com

8.0 Announcements

Add WeChat powcoder

CS306: Announcements

- HW2 to come out this week
  - this time, for real!
  - covers MACs, hashing, PK-crypto, honeywords…
  - due in two weeks
- HW1 grades
  - talk to TAs
- Midterm grades
  - as soon as possible, given that they are graded by your instructor – end of week
- Labs resume this Thursday

CS306: Tentative Syllabus

Week	Date	Topics	Reading	Assignment
1	Sep 1	Introduction	Lecture 1	-
2	Sep 8	Symmetric-key encryption	Lecture 2	Lab 1
3	Sep 15	Perfect secrecy	Lecture 3	Lab 2, HW 1
4	Sep 22	Ciphers in practice I	Lecture 4	Lab 3, HW 1
5	Sep 29	Ciphers in practice II	Lecture 5	Lab 4
6	Oct 6	MACs & hashing	Lecture 6	Lab 5
-	Oct 13	No class (Monday schedule)		Lab 6
7	Oct 20	Public-key cryptography	Lecture 7	

CS306: Tentative Syllabus (continued)

Week	Date	Topics	Reading	Assignment
8	**Oct 27**	**Midterm**	**All materials covered**	
9	Nov 3	Crypto Light: Applications	Lecture 8	Lab 7, HW 2
10	Nov 10	Access Control & User Authentication		
11	Nov 17	Web & Network Security		
12	Nov 24	Cloud Security & Privacy		
13	Dec 1	Software/Database Security		
14	Dec 8	Economics, Legal & Ethical Issues		
15	Dec 15 (or later)	**Final** (closed “books”)	All materials covered*	

* w/ focus on what covered after midterm

Two weeks ago

- Public-key (PK) cryptography
  - Motivation, PK Infrastructure, PK encryption, digital signatures
  - Discrete log problem & ElGamal encryption, hybrid encryption
- Demo
  - The length-extension attack against naïve HMAC…

Today

- Crypto Light
  - Special topics on message authentication, cryptographic hashing, RSA
  - Final remarks on crypto tools w/ a focus on practical applications
  - Topics
    - authenticated encryption, HMAC
    - cryptographic hashing in practice & Honeywords
    - RSA problem & RSA crypto system

8.1 Authenticated encryption

Recall: Two distinct properties

Secrecy

- **sensitive** information has value
  - if **leaked**, it can be **risky**
- **specific** scope / **general** semantics
- **prevention**
- does **not** imply integrity
  - e.g., bit-flipping “attack”

Integrity

- **correct** information has value
  - if **manipulated**, it can **harmful**
  - **random Vs. adversarial** manipulation
- **wider** scope / **context-specific** semantics
  - source Vs. content authentication
  - replay attacks
- **detection**
- does **not** imply secrecy
  - e.g., user knows cookies’ “contents”

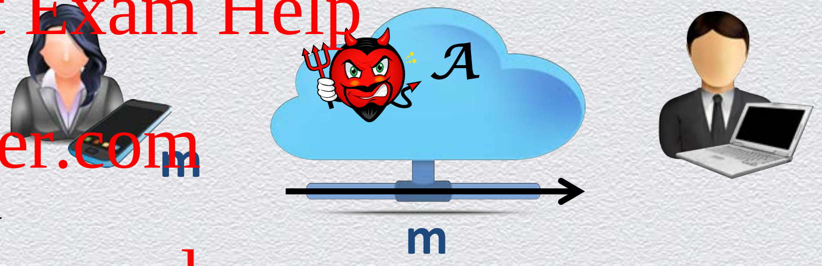
Recall: Yet, they are quite close…

Common setting

- communication (storage) over an "**open**," i.e., **unprotected**, channel (medium)

Fundamental security problems

- while in transit (at rest)
  - no message (file) should be **leaked** to $\mathcal{A}$
  - no message (file) should be **modified** by $\mathcal{A}$



Core cryptographic protections

- **encryption schemes** provide **secrecy / confidentiality**
- **MAC schemes** provide **integrity / unforgeability**

Can we achieve both at once in the symmetric-key setting? **Yes!**

Authenticated Encryption (AE): Catch 2 birds w/ 1 stone

Cryptographic primitive that realizes an "**ideally secure**" communication channel

- motivation
  - important in practice as real apps often **need both**
  - **good security hygiene**
    - even if a given app "asks" only/more for secrecy or integrity than the other, it's always better **to achieve both**!

Three generic AE constructions

Constructions of a **secure authenticated encryption** scheme Π_{AE}

- they all make use of
  - a **CPA-secure** encryption scheme Π_E = (Enc, Dec); and
  - a **secure MAC** Π_M = (Mac, Vrf)
  - which are instantiated using **independent** secret keys ke, km
  - …but the **order** with which these are used matters!

Generic AE constructions (1)

1. **encrypt-and-authenticate**

- $Enc_{ke}(m) \longrightarrow c$; $Mac_{km}(m) \longrightarrow t$; send ciphertext (c, t)
- if $Dec_{ke}(c) = m \neq$ `fail` and $Vrf_{km}(m,t)$ accepts, output m; else output `fail`
- **insecure scheme, generally**
  - e.g., MAC tag t may leak information about m
  - e.g., if MAC is deterministic (e.g., CBC-MAC) then Π_{AE} is not even CPA-secure
  - used in SSH

Generic AE constructions (2)

2. **authenticate-then-encrypt**

- $Mac_{km}(m) \longrightarrow t$; $Enc_{ke}(m||t) \longrightarrow c$; send ciphertext c
- if $Dec_{ke}(c) = m||t \neq$ `fail` and $Vrf_{km}(m,t)$ accepts, output m; else output `fail`
- **insecure scheme, generally**
  - used in TLS, IPsec

Generic AE constructions (3)

3. **encrypt-then-authenticate** (cf. "authenticated encryption")

- $\mathrm{Enc}_{ke}(m) \longrightarrow c$; $\mathrm{Mac}_{km}(c) \longrightarrow t$; send ciphertext (c, t)
- if $\mathrm{Vrf}_{km}(c,t)$ accepts then output $\mathrm{Dec}_{ke}(c) = m$, else output `fail`
- **secure scheme, generally** (as long as Π_M is a **"strong" MAC**)
  - used in TLS, SSHv2, IPsec

Application: Secure communication sessions

An AE scheme Π_{AE} = (Enc, Dec) enables two parties to **communicate securely**

- session: period of time during which sender and receiver maintain state
- idea: send any message m as $c = Enc_k(m)$ & ignore received c that don't verify
- security: **secrecy & integrity are protected**
- remaining possible attacks
  - **re-ordering** attack — counters can be used to eliminate reordering/replays
  - **reflection** attack — directional bit can be used to eliminate reflections
  - **replay** attack — $c = Enc_k(b_{A \to B} | ctr_{A,B} || m)$; $ctr_{A,B}$++

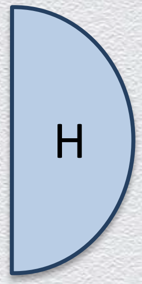
8.2 Applications of cryptographic hashing

Recall: Cryptographic hash functions

Basic cryptographic primitive

- maps **objects** to a **fixed-length** binary strings
- for all practical purposes, mapping **avoids collisions** (distinct objects x ≠ y mapped to same value H(x) = H(y))
  - **collision resistance**: no distinct inputs can be efficiently found that collide in the hash domain

input
arbitrarily long string



output
short digest, fingerprint, “secure” description

“any object can be **securely** mapped to a **practically unique** short digest”

Collision resistance implies two weaker security properties

- finding a collusion w.r.t. a given **random object x** is also infeasible
- finding any preimage of a **random hash value h** is also infeasible

Recall: Weaker security notions

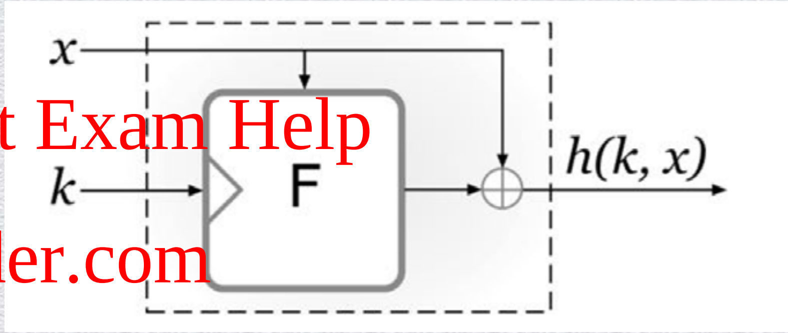
Given a hash function $H: X \rightarrow Y$, then we say that H is

- **preimage resistant** (or **one-way**)
  - given a **uniform** $y \in Y$, finding a value $x \in X$ s.t. $H(x) = y$ happens negligibly often
- **2-nd preimage resistant** (or **weak collision resistant**)
  - given a **uniform** $x \in X$, finding a value $x' \in X$, s.t. $x' \neq x$ and $H(x') = H(x)$ happens negligibly often

- cf. **collision resistant** (or **strong collision resistant**)
  - finding two distinct values $x', x \in X$, s.t. $H(x') = H(x)$ happens negligibly often

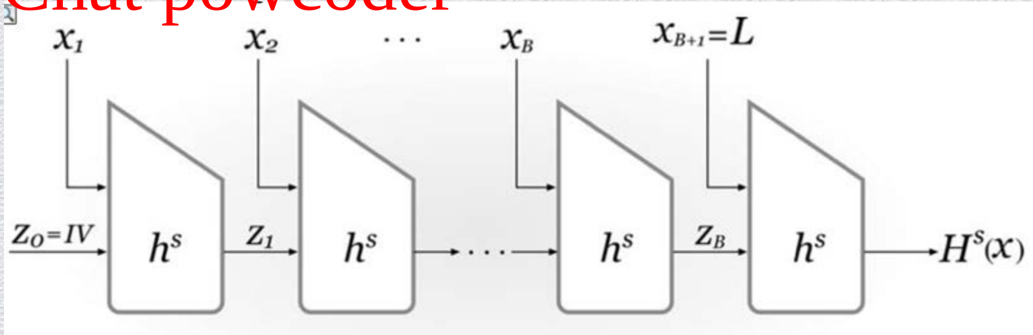
Recall: Davies-Meyer & Merkle-Damgård transforms

$h(x||k) = F_k(x)$ XOR x



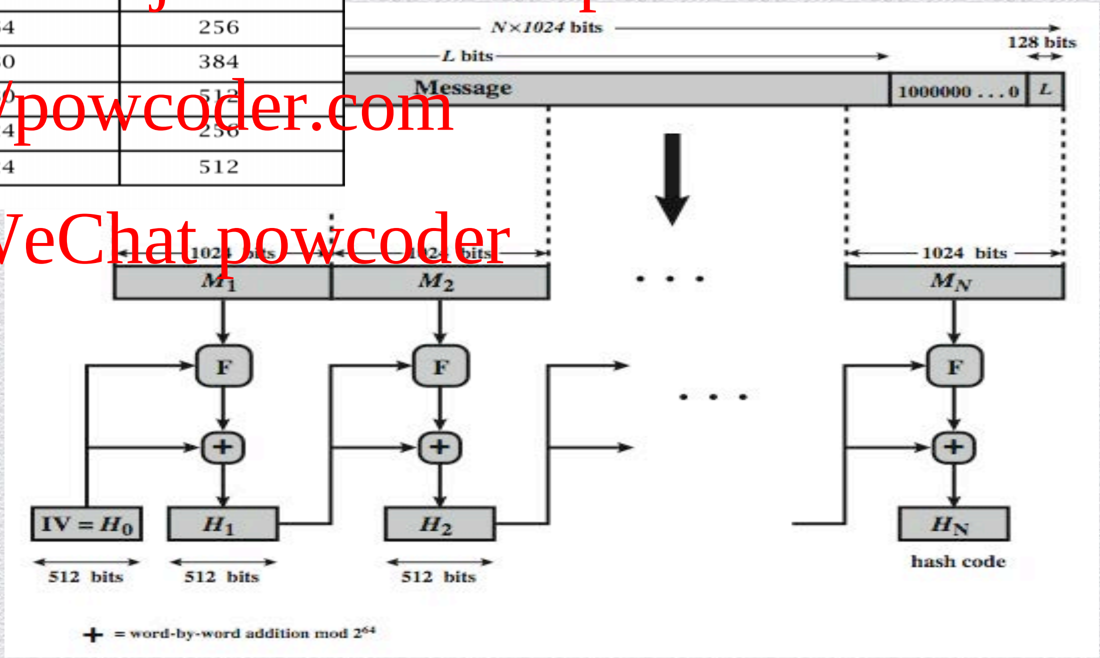
- h is a CR compression function,
  if F is an **ideal cipher (a more secure PRF)**

- H is a CR hash function, h is CR



Recall: Current hash standards & SHA2-512

Algorithm	Maximum Message Size (bits)	Block Size (bits)	Rounds	Message Digest Size (bits)
MD5	2^{64}	512	64	128
SHA-1	2^{64}	512	80	160
SHA-2-224	2^{64}	512	64	224
SHA-2-256	2^{64}	512	64	256
SHA-2-384	2^{128}	1024	80	384
SHA-2-512	2^{128}	1024	80	512
SHA-3-256	unlimited	1088	24	256
SHA-3-512	unlimited	576	24	512



Recall: Birthday attacks against collision resistance

Assume a CR function h producing hash values of size n

- **brute-force** attack
  - evaluate h on **$2^n + 1$** distinct inputs
  - by the "pigeon hole" **principle**, at least 1 collision **will be** found
- **birthday** attack
  - evaluate h on (much) **fewer** distinct inputs that hash to **random** values
  - by "balls-into-bins" **probabilistic analysis**, at least 1 collision will **likely** be found
  - when hashing **only half** distinct inputs, it's **more likely** to find a collision!
  - thus, in order to get **n-bit security**, we (**at least**) need **hash values of length 2n**

Recall: Security strength due to birthday attacks

- # hash evaluations for finding collisions on n-bit digests with probability p

Bits n	Possible outputs (2 s.f.) (H) m	Desired probability of random collision (2 s.f.) (p)									
		10^{-18}	10^{-15}	10^{-12}	10^{-9}	10^{-6}	0.1%	1%	25%	50%	75%
16	65,536	<2	<2	<2	<2	<2	11	36	190	300	430
32	4.3×10^9	<2	<2	<2	3	93	2900	9300	50,000	77,000	110,000
64	1.8×10^{19}	6	190	6100	190,000	6,100,000	1.9×10^8	6.1×10^8	3.3×10^9	5.1×10^9	7.2×10^9
128	3.4×10^{38}	2.6×10^{10}	8.2×10^{11}	2.6×10^{13}	8.2×10^{14}	2.6×10^{16}	8.3×10^{17}	2.6×10^{18}	1.4×10^{19}	2.2×10^{19}	3.1×10^{19}
256	1.2×10^{77}	4.8×10^{29}	1.5×10^{31}	4.8×10^{32}	1.5×10^{34}	4.8×10^{35}	1.5×10^{37}	4.8×10^{37}	2.6×10^{38}	4.0×10^{38}	5.7×10^{38}
384	3.9×10^{115}	8.9×10^{48}	2.8×10^{50}	8.9×10^{51}	2.8×10^{53}	8.9×10^{54}	2.8×10^{56}	8.9×10^{56}	4.8×10^{57}	7.4×10^{57}	1.0×10^{58}
512	1.3×10^{154}	1.6×10^{68}	5.2×10^{69}	1.6×10^{71}	5.2×10^{72}	1.6×10^{74}	5.2×10^{75}	1.6×10^{76}	8.8×10^{76}	1.4×10^{77}	1.9×10^{77}

- for large $m = 2^n$, average # hash evaluations before finding the first collision is

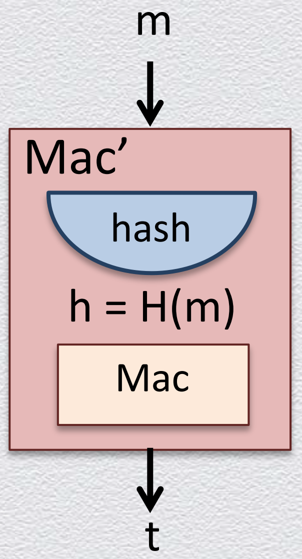
$1.25(m)^{1/2}$

8.2.1 Applications of cryptographic hashing in cryptography

[1] "Hash-and-MAC" & "hash-and-sign"

Hash-and-MAC construction based on

- a CR hash function H; and
- a secure fixed-message MAC



Similarly, digital signatures are used with hash functions

- the hash of a message is signed, instead of the message itself

Intuition

- since **H is CR**:
  authenticating **digest H(m)** is **a good as** authenticating **m itself**!

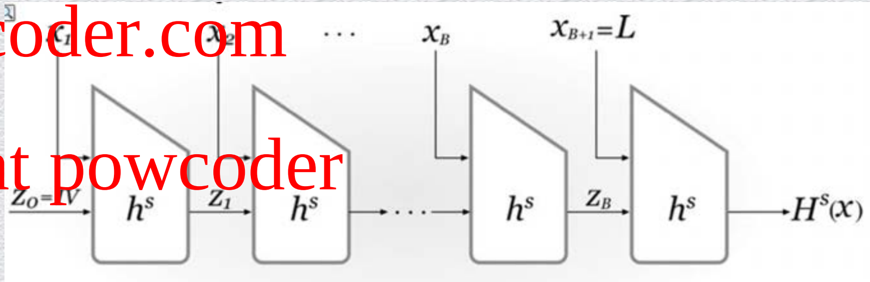
[2] Hash-based MAC (HMAC): A naïve, insecure, approach

Set tag t as:

$$\mathrm{Mac}_k(m) = \mathbf{H}(k||\mathbf{m})$$

- intuition: given H(k||m) it should be infeasible to compute H(k||m'), m' ≠ m

Insecure construction



- practical CR hash functions employ the Merkle-Damgård design

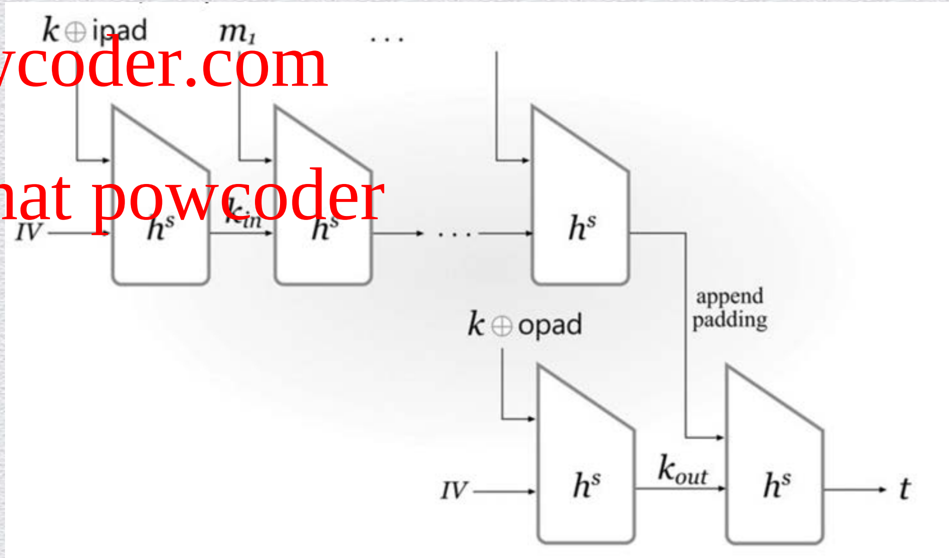
- **length-extension attack**
  - knowledge of $H(m_1)$ makes it feasible to compute $H(m_1||m_2)$
  - by knowing the length of m_1, one can learn internal state z_B even without knowing m_1!

[2] HMAC: Secure design

Set tag t as:

$$HMAC_k[m] = H[\ (k \oplus opad)\ ||\ H[\ (k \oplus ipad)\ ||\ m\ ]\ ]$$

- intuition: **instantiation of hash & sign paradigm**
- two layers of hashing H
  - **upper layer**
    - y = H($(k \oplus ipad)$ || m)
    - y = H’(m), i.e., “hash”
  - **lower layer**
    - t = H ($(k \oplus opad)$ || y’)
    - t = Mac’(k_{out}, y’), i.e., “sign”



8.2.2 Applications of cryptographic hashing in security

[1] Digital envelops

Commitment schemes

- two operations
- commit(x, r) = C
  - i.e., put message x into an envelop (using randomness r)
  - e.g., commit(x, r) = h(x || r)
  - **hiding property**: you cannot see through an (opaque) envelop
- open(C, m, r) = ACCEPT or REJECT
  - i.e., open envelop (using r) to check that it has not been tampered with
  - e.g., open(C, x, r): check if h(x || r) =? C
  - **binding property**: you cannot change the contents of a sealed envelop

[1] Security properties

Hiding: perfect opaqueness

- similar to indistinguishability; commitment reveals nothing about message
  - adversary selects two messages x_1, x_2 which he gives to challenger
  - challenger randomly selects bit b, computes (randomness and) commitment C_i of x_i
  - challenger gives C_b to adversary, who wins if he can find bit b (better than guessing)

Binding: perfect sealing

- similar to unforgeability; cannot find a commitment “collision”
  - adversary selects two distinct messages x_1, x_2 and two corresponding values r_1, r_2
  - adversary wins if commit(x_1, r_1) = commit(x_2, r_2)

[1] Example 1: Online auctions

Suppose Alice, Bob, Charlie are bidders in an online auction

- Alice plans to bid A, Bob B and Charlie C
  - they do not trust that bids will be secret
  - nobody is willing to submit their bid
- solution
  - Alice, Bob, Charlie submit **hashes** h(A), h(B), h(C) of their bids
  - all received hashes are posted online
  - then parties' bids A, B and C revealed
- analysis
  - "hiding:" hashes do not reveal bids (which property?)
  - "binding:" cannot change bid after hash sent (which property?)

[1] Example 1: Online auctions (II)

A general issue with concealing private data via hashing

- due to the small search space, this protocol is not secure!
- a forward search attack is possible
  - e.g., Bob computes h(A) for the most likely bids A
- how to prevent this?
  - increase search space
  - e.g., Alice computes h(A||R), where R is randomly chosen
    - at the end, Alice must reveal A and R
    - but before he chooses B, Bob cannot try all A and R combination

[1] Example 2: Fair decision via coin flipping

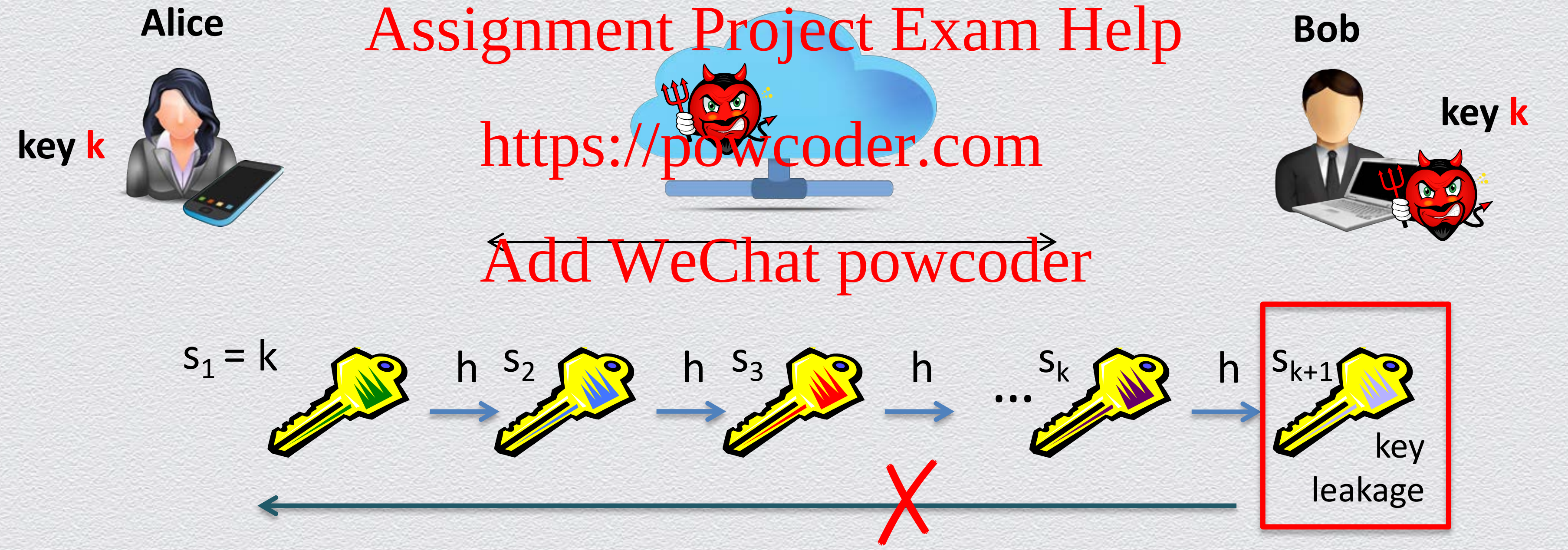
Alice is to “call” the coin flip and Bob is to flip the coin

- to decide who will do the dishes…
- problem: Alice may change her mind, Bob may skew the result
- protocol
  - Alice "calls" the coin flip but only tells Bob a commitment to her call
  - Bob flips the coin and reports the result
  - Alice reveals what she committed to
  - Bob verifies that Alice's call matches her commitment
  - If Alice's revelation matches the coin result Bob reported, Alice wins
- hiding: Bob does not get any advantage by seeing Alice commitment
- binding: Alice cannot change her mind after the coin is flipped

[2] Forward-secure key rotation

Alice and Bob secretly communicate using symmetric encryption

- Eve intercepts their messages and later breaks into Bob's machine to steal the shared key



[3] Hash values as file identifiers

Consider a cryptographic hash function H applied on a file F

- the hash (or digest) H(M) of F serves as a **unique** identifier for F
  - "uniqueness"
    - if another file F' has the same identifier, this contradicts the security of H
  - thus
    - the hash H(F) of F is like a fingerprint
    - one can check whether two files are equal by comparing their digests

Many real-life applications employ this simple idea!

Examples

3.1 Virus fingerprinting

- When you perform a virus scan over your computer, the virus scanner application tries to identify and block or quarantine programs or files that contain viruses
- This search is primarily based on comparing the digest of your files against a database of the digests of already known viruses
- The same technique is used for confirming that is safe to download an application or open an email attachment

3.2 Peer-to-peer file sharing

- In distributed file-sharing applications (e.g., systems allowing users to contribute contents that are shared amongst each other), both shared files and participating peer nodes (e.g., their IP addresses) are uniquely mapped into identifiers in a hash range
- When a given file is added in the system it is consistently stored at peer nodes that are responsible to store files those digests fall in a certain sub-range
- When a user looks up a file, routing tables (storing values in the hash range) are used to eventually locate one of the machines storing the searched file

Example 3.3: Data deduplication

Goal: Elimination of duplicate data

- Consider a cloud provider, e.g., Gmail or Dropbox, storing data from numerous users.
- A vast majority of stored data are duplicates; e.g., think of how many users store the same email attachments, or a popular video…
- Huge cost savings result from deduplication:
  - a provider stores identical contents possessed by different users once!
  - this is completely transparent to end users!

Idea: Check redundancy via hashing

- Files can be reliably checked whether they are duplicates by comparing their digests.
- When a user is ready to upload a new file to the cloud, the file's digest is first uploaded.
- The provider checks to find a possible duplicate, in which case a pointer to this file is added.
- Otherwise, the file is being uploaded literally
- This approach saves both storage and bandwidth!

[4] Password hashing

Goal: User authentication

- Today, passwords are the dominant means for user authentication, i.e., the process of verifying the identity of a user (requesting access to some computing resource).
- This is a "something you know" type of user authentication, assuming that only the legitimate user knows the correct password.
- When you provide your password to a computer system (e.g., to a server through a web interface), the system checks if your submitted password matches the password that was initially stored in the system at setup.

Problem: How to protect password files

- If password are stored at the server in the clear, an attacker can steal the password file after breaking into the authentication server – this type of attack happens routinely nowadays…
- Password hashing involved having the server storing the hashes of the users passwords.
- Thus, even if a password file leaks to an attacker, the onewayness of the used hash function can guarantee some protections against user-impersonation simply by providing the stolen password for a victim user.

[4] Password storage

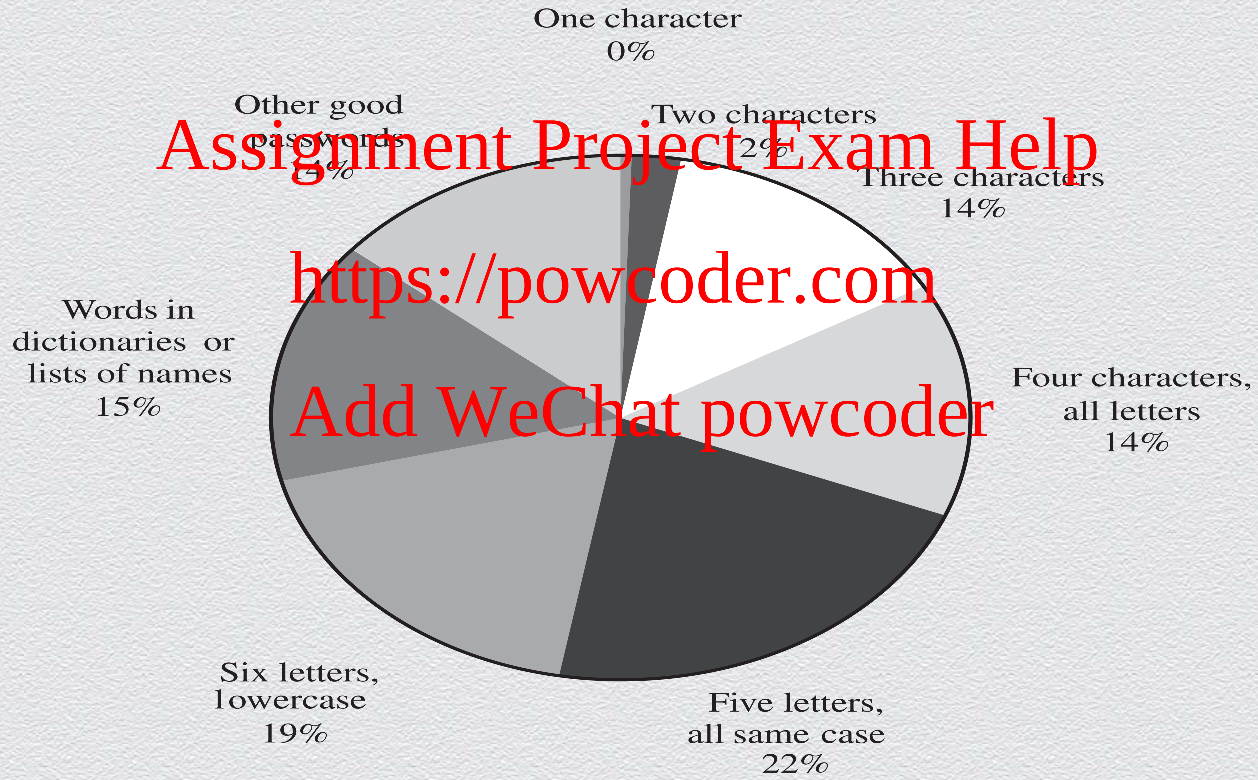
Identity	Password
Jane	qwerty
Pat	aaaaaa
Phillip	oct31witch
Roz	aaaaaa
Herman	guessme
Claire	aq3wm$oto4

Plaintext

Identity	Password
Jane	0x471aa2d2
Pat	0x13b9c32f
Phillip	0x01c142be
Roz	0x13b9c32f
Herman	0x5202aae2
Claire	0x488b8c27

Concealed via hashing

[4] Distribution of password types



[4] Dictionary attacks

- "online" brute-force or dictionary attack against passwords
  - employs only the authentication system
  - the attacker tries to impersonate a victim by trying
    - all possible (short length) passwords or
    - passwords coming from a known dictionary
- “offline” brute-force or dictionary attack
  - employs a leaked file of hashed passwords

[4] Countermeasures

Password salting

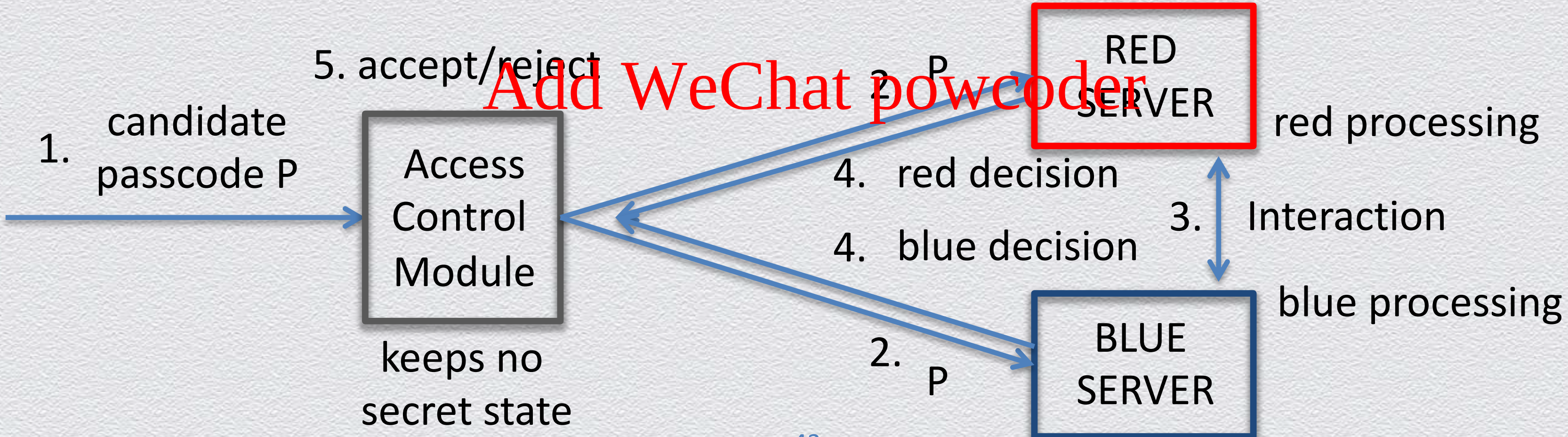
- to slow down dictionary attacks
  - a user-specific **salt** is appended to a user's password before it is being hashed
  - each salt value is stored in the clear along with its corresponding hashed password
  - if two users have the same password, they will have different hashed passwords
  - example: Unix uses a 12 bit salt

Hash strengthening

- to slow down dictionary attacks
  - a password is hashed k times before being stored

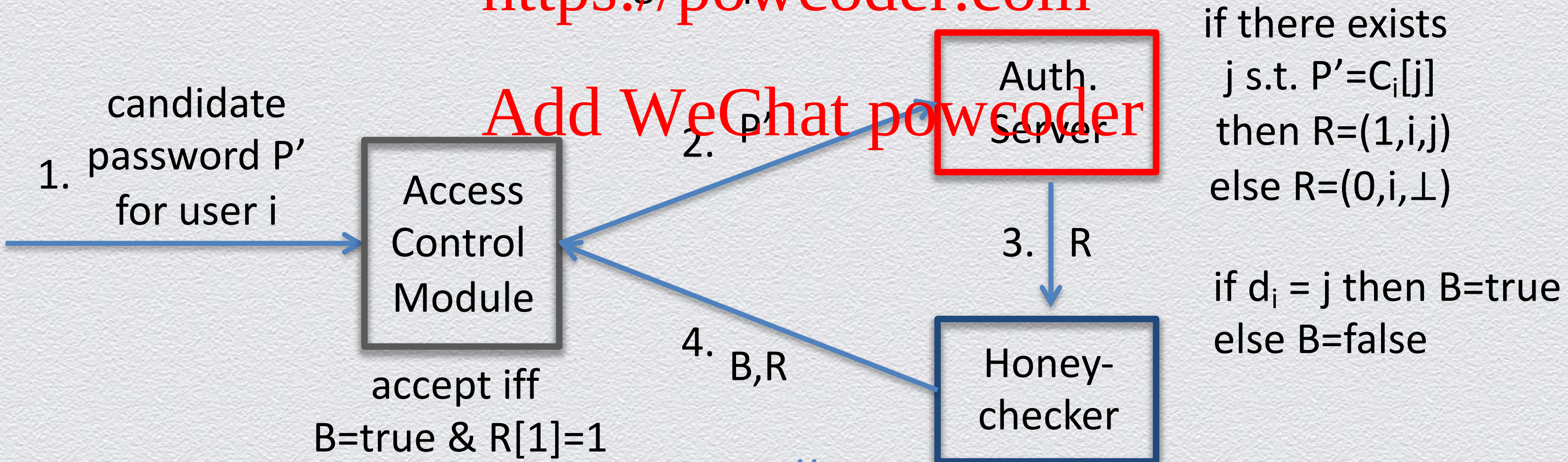
[4] A promising approach: Split verification into two servers

- <u>Key idea</u>: Distribute password verification across two servers
  - Red server verifies "half" the credentials; blue server verifies other "half"
  - Authentication decision relies on both outputs
    - Compromise of one server gives no/little advantage to attacker



[4] Honeywords

- Based on decoy passwords, aka honeywords
  - Red stores user's i real password P_i and k-1 fake ones in unlabeled set C_i
  - Blue server stores the index d_i of P_i in set C_i
  - Password verification through sequential verifications



[4] Example

- User nikos: hel00w0rld, **heloow0rld,** hel00w1rld, hel11w0rld
- Red server: nikos, hel00w0rld, heloow0rld, hel00w1rld, hel11w0rld
- Blue server: nikos, 2

- User provides hel11w0rld
- Red: if password is contained in list at position 4, then output (nikos, 4)
  - Else reject
- Blue: if match, then output OK, else ALERT

8.3 Number theory background

Multiplicative inverses

The residues modulo a positive integer n comprise set $Z_n = \{0,1,2,\ldots,n - 1\}$

- let x and y be two elements in Z_n such that $x\ y \bmod n = 1$
  - we say: y is the multiplicative inverse of x in Z_n
  - we write: $y = x^{-1}$

- example:
  - multiplicative inverses of the residues **modulo 11**

x	0	1	2	3	4	5	6	7	8	9	10
x^{-1}		1	6	4	3	9	2	8	7	5	10

Multiplicative inverses (cont'ed)

Theorem

An element x in Z_n has a multiplicative inverse iff x, n are relatively prime

- e.g., the only elements of Z_{10} having a multiplicative inverse are 1, 3, 7, 9

x	0	1	2	3	4	5	6	7	8	9
x^{-1}		1		7				3		9

Corollary

If p is prime, every non-zero residue in Z_p has a multiplicative inverse

Theorem

A variation of Euclid's GCD algorithm computes the multiplicative inverse of an element x in Z_n or determines that it does not exist

Computing multiplicative inverses

Fact

- given two numbers **a** and **b**, there exist integers x, y s.t.

$$\mathbf{x\ a + y\ b = gcd(a, b)}$$

which can be computed efficiently by the extended Euclidean algorithm.

Thus

- the multiplicative inverse of a in Z_b exists iff gcd(a, b) = 1
- i.e., iff the extended Euclidean algorithm computes x and y s.t. **x a + y b = 1**
- in this case, the multiplicative inverse of a in Z_b is **x**

Euclid's GCD algorithm

Computes the greater common divisor by repeatedly applying the formula

gcd(a, b) = gcd(b, a mod b)

- example
  - gcd(412, 260) = 4

Algorithm EuclidGCD(a, b)
Input integers **a** and **b**
Output gcd(a, b)

if b = 0
return a
else
return EuclidGCD(b, a mod **b)**

a	412	260	152	108	44	20	4
b	260	152	108	44	20	4	0

Extended Euclidean algorithm

Theorem

If, given positive integers **a** and **b**, **d** is the smallest positive integer s.t. **d** = **ia** + **jb**, for some integers **i** and **j**, then **d** = gcd(**a, b**)

- example
  - **a** = 21, **b** = 15
  - **d** = 3, **i** = 3, **j** = -4
  - 3 = 3·21 + (-4)·15 = 63 - 60 = 3

Algorithm Extended-Euclid(**a, b**)
  Input integers **a** and **b**
  Output gcd(**a**, **b**), i and j s.t. ia+jb = gcd(a,b)
  if **b** = 0
    return (a,1,0)
  (d′, x′, y′) = Extended-Euclid(**b, a** mod **b**)
  (d, x, y) = (d′, y′, x′ - [a/b]y′)
  return (d, x, y)

Multiplicative group

A set of elements where multiplication • is defined

- closure, associativity, identity & inverses
- multiplicative groups Z^*_n, defined w.r.t. Z_n (residues modulo n)
  - subsets of Z_n containing all integers that are relative prime to n
  - **CASE 1**: if n is a prime number, then all non-zero elements in Z_n have an inverse
    - $Z^*_7 = \{1,2,3,4,5,6\}$, n = 7
    - 2 • 4 = 1 (mod 7), 3 • 5 = 1 (mod 7), 6 • 6 = 1 (mod 7), 1 • 1 = 1 (mod 7)
  - **CASE 2**: if n is not prime, then not all integers in Z_n have an inverse
    - $Z^*_{10} = \{1,3,7,9\}$, n = 10
    - 3 • 7 = 1 (mod 10), 9 • 9 = 1 (mod 10), 1 • 1 = 1 (mod 10)

Order of a multiplicative group

Order of a group = cardinality of the group

- multiplicative groups for Z^*_n
- the totient function $\phi(n)$ denotes the order of Z^*_n, i.e., $\phi(n) = |Z^*_n|$
  - if **n = p is prime**, then the order of $Z^*_p=\{1,2,\ldots,p-1\}$ is p-1, i.e., $\phi(n) = p-1$
    - e.g., $Z^*_7 = \{1,2,3,4,5,6\}$, n = 7, $\phi(7) = 6$
  - if **n is not prime**, $\phi(n) = n(1-1/p_1)(1-1/p_2)\ldots(1-1/p_k)$, where $n = p^{e1}_1 p^{e2}_2 \ldots p^{ek}_k$
    - e.g., $Z^*_{10} = \{1,3,7,9\}$, n = 10, $\phi(10) = 4$

- if n = p q, where p and q are distinct primes, then $\phi(n) = (p-1)(q-1)$ **Factoring problem**
  - difficult problem: given n = pq, where p, q are primes, find p and q or $\phi(n)$

Fermat's Little Theorem

Theorem

If **p is a prime**, then for each nonzero residue x in Z_p, we have $x^{p-1} \bmod p = 1$

- example (p = 5):

$1^4 \bmod 5 = 1$

$2^4 \bmod 5 = 16 \bmod 5 = 1$

$3^4 \bmod 5 = 81 \bmod 5 = 1$

$4^4 \bmod 5 = 256 \bmod 5 = 1$

Corollary

If **p is a prime**, then the multiplicative inverse of each x in Z^*_p is $x^{p-2} \bmod p$

- proof: $x(x^{p-2} \bmod p) \bmod p = xx^{p-2} \bmod p = x^{p-1} \bmod p = 1$

Euler's Theorem

Theorem

For each element x in Z^*_n, we have $x^{\phi(n)}$ mod n = 1

- example (**n = 10**)
  - $Z^*_{10} = \{1,3,7,9\}$, n = 10, $\phi(10) = 4$
  - $3^{\phi(10)}$ mod 10 = 3^4 mod 10 = 81 mod 10 = 1
  - $7^{\phi(10)}$ mod 10 = 7^4 mod 10 = 2401 mod 10 = 1
  - $9^{\phi(10)}$ mod 10 = 9^4 mod 10 = 6561 mod 10 = 1

Computing in the exponent

For the multiplicative group Z^*_n, we can reduce the exponent modulo ϕ(n)

- $x^y \bmod n = x^{k\,\phi(n) + r} \bmod n = (x^{\phi(n)})^k x^r \bmod n = x^r \bmod n = x^{y \bmod \phi(n)} \bmod n$

Corollary: For Z^*_p, we can reduce the exponent modulo p-1

- example
  - $Z^*_{10} = \{1,3,7,9\}$, n = 10, ϕ(10) = 4
  - $3^{1590} \bmod 10 = 3^{1590 \bmod 4} \bmod 10 = 3^2 \bmod 10 = 9$
- example
  - $Z^*_p = \{1,2,\ldots,p - 1\}$, p = 19, ϕ(19) = 18
  - $15^{39} \bmod 19 = 15^{39 \bmod 18} \bmod 19 = 15^3 \bmod 19 = 12$

Powers

Let p be a prime

- the sequences of successive powers of the elements in Z^*_p exhibit repeating subsequences
- the sizes of the repeating subsequences and the number of their repetitions are the divisors of p – 1
- example, p = 7

x	x^2	x^3	x^4	x^5	x^6
1	1	1	1	1	1
2	4	1	2	4	1
3	2	6	4	5	1
4	2	1	4	2	1
5	4	6	2	3	1
6	1	6	1	6	1

8.4 The RSA algorithm

The RSA algorithm (for encryption)

General case

Setup (run by a given user)

- $\mathbf{n} = \mathbf{p} \cdot \mathbf{q}$, with **p** and **q** primes
- **e** relatively prime to $\phi(n) = (\mathbf{p} - 1)(\mathbf{q} - 1)$
- **d** inverse of **e** in $\mathbf{Z}_{\phi(n)}$

Keys

- public key is $\mathbf{K_{PK}} = (\mathbf{n}, \mathbf{e})$
- private key is $\mathbf{K_{SK}} = \mathbf{d}$

Encryption

- $\mathbf{C} = \mathbf{M}^{\mathbf{e}} \bmod \mathbf{n}$ for plaintext **M** in $\mathbf{Z_n}$

Decryption

- $\mathbf{M} = \mathbf{C}^{\mathbf{d}} \bmod \mathbf{n}$

Example

Setup

- $\mathbf{p} = 7$, $\mathbf{q} = 17$, $\mathbf{n} = 7 \cdot 17 = 119$
- $\mathbf{e} = 5$, $\boldsymbol{\phi}(\mathbf{n}) = 6 \cdot 16 = 96$
- $\mathbf{d} = 77$

Keys

- public key is (119, 5)
- private key is 77

Encryption

- $\mathbf{C} = 19^5 \bmod 119 = 66$ for $\mathbf{M} = 19$ in $\mathbf{Z_{119}}$

Decryption

- $\mathbf{M} = 66^{77} \bmod 119 = 19$

Another complete example

- Setup
  - **p** = 5, **q** = 11, **n** = 5 · 11 = 55
  - **ϕ**(**n**) = 4 · 10 = 40
  - **e** = 3, **d** = 27 (3·27 = 81 = 2·40 + 1)
- Encryption
  - **C** = **M**^{3} mod 55 for **M** in $\mathbf{Z}_{55}$
- Decryption
  - **M** = **C**^{27} mod 55

M	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
C	1	8	27	9	15	51	13	17	14	10	11	23	52	49	20	26	18	2
M	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36
C	39	25	21	33	12	19	5	31	48	7	24	50	36	43	22	34	30	16
M	37	38	39	40	41	42	43	44	45	46	47	48	49	50	51	52	53	54
C	53	37	29	35	6	3	32	44	45	41	38	42	4	40	46	28	47	54

*Correctness of RSA

Given

Setup

- $\mathbf{n} = \mathbf{p} \cdot \mathbf{q}$, with **p** and **q** primes
- **e** relatively prime to $\phi(n) = (\mathbf{p} - 1)(\mathbf{q} - 1)$
- **d** inverse of **e** in $\mathbf{Z}_{\phi(n)}$ **(1)**

Encryption

- $\mathbf{C} = \mathbf{M}^{e} \bmod \mathbf{n}$ for plaintext **M** in $\mathbf{Z}_n$

Decryption

- $\mathbf{M} = \mathbf{C}^{d} \bmod \mathbf{n}$

Fermat's Little Theorem **(2)**

- for prime p, non-zero x: $x^{p-1} \bmod p = 1$

Analysis

Need to show

- $M^{ed} = M \bmod p \cdot q$

Use **(1)** and apply **(2)** for prime p

- $M^{ed} = M^{ed-1} M = (M^{p-1})^{h(q-1)} M$
- $\mathbf{M}^{ed} = 1^{h(q-1)} M \bmod p = \mathbf{M \bmod p}$

Similarly (w.r.t. prime q)

- $\mathbf{M^{ed} = M \bmod q}$

Thus, since p, q are co-primes

- $\mathbf{M^{ed} = M \bmod p \cdot q}$

A useful symmetry

[1] RSA setting

- modulo $\mathbf{n} = \mathbf{p} \cdot \mathbf{q}$, p & q are **primes**, public & private keys (e,d): $\mathbf{d \cdot e = 1 \bmod (p-1)(q-1)}$

[2] RSA operations involve **exponentiations**, thus they are **interchangeable**

- $\mathbf{C} = \mathbf{M^e} \bmod \mathbf{n}$ (encryption of plaintext $\mathbf{M}$ **in** $\mathbf{Z_n}$)
- $\mathbf{M} = \mathbf{C^d} \bmod \mathbf{n}$ (decryption of ciphertext $\mathbf{C}$ **in** $\mathbf{Z_n}$)

Indeed, their order of execution does not matter: $\mathbf{(M^e)^d = (M^d)^e \bmod n}$

[3] RSA operations involve exponents that **"cancel out"**, thus they are **complementary**

- $\mathbf{x^{(p-1)(q-1)}} \bmod \mathbf{n = 1}$ (Euler's Theorem)

Indeed, they invert each other: $\mathbf{(M^e)^d = (M^d)^e = M^{ed} = M^{k(p-1)(q-1)+1} \bmod n}$

$\mathbf{= (M^{(p-1)(q-1)})^k \cdot M = 1^k \cdot M = M \bmod n}$

Signing with RSA

RSA functions are complementary & interchangeable w.r.t. order of execution

- **core property**: $M^{ed} = M \bmod p \cdot q$ for any message **M in** $\mathbf{Z_n}$

RSA cryptosystem lends itself to a **signature scheme**

- 'reverse' use of keys is possible : $(M^d)^e = M \bmod p \cdot q$

- signing algorithm **Sign(M,d,n)**: $\sigma = M^d \bmod n$ for message **M** in $\mathbf{Z_n}$
- verifying algorithm **Vrfy(σ,M,e,n)**: return $M == \sigma^e \bmod n$

The RSA algorithm (for signing)

General case

Setup (run by a given user)

- $\mathbf{n} = \mathbf{p} \cdot \mathbf{q}$, with $\mathbf{p}$ and $\mathbf{q}$ primes
- $\mathbf{e}$ relatively prime to $\boldsymbol{\phi}(\mathrm{n}) = (\mathbf{p} - 1)(\mathbf{q} - 1)$
- $\mathbf{d}$ inverse of $\mathbf{e}$ in $\mathbf{Z}_{\boldsymbol{\phi}(\mathbf{n})}$

Keys (same as in encryption)

- public key is $\mathbf{K}_{\mathbf{PK}} = (\mathbf{n}, \mathbf{e})$
- private key is $\mathbf{K}_{\mathbf{SK}} = \mathbf{d}$

Sign

- $\boldsymbol{\sigma} = \mathbf{M}^{\mathbf{d}} \bmod \mathbf{n}$ for message $\mathbf{M}$ in $\mathbf{Z}_{\mathbf{n}}$

Verify

- Check if $\mathbf{M} = \boldsymbol{\sigma}^{\mathbf{e}} \bmod \mathbf{n}$

Example

Setup

- $\mathbf{p} = 7$, $\mathbf{q} = 17$, $\mathbf{n} = 7 \cdot 17 = 119$
- $\mathbf{e} = 5$, $\boldsymbol{\phi}(\mathbf{n}) = 6 \cdot 16 = 96$
- $\mathbf{d} = 77$

Keys

- public key is (119, 5)
- private key is 77

Signing

- $\boldsymbol{\sigma} = 66^{77} \bmod 119 = 19$ for $\mathbf{M} = 66$ in $\mathbf{Z}_{119}$

Verification

- Check if $\mathbf{M} = 19^5 \bmod 119 = 66$

Digital signatures & hashing

Very often digital signatures are used with hash functions

- the hash of a message is signed, instead of the message itself

Signing message M

- let h be a cryptographic hash function, assume RSA setting (n, d, e)
- compute signature σ on message M as: $\sigma = h(M)^d \bmod n$
- send σ, M

Verifying signature σ

- use public key (e, n) to compute (candidate) hash value $H = \sigma^e \bmod n$
- if H = h(M) output ACCEPT, else output REJECT

Security of RSA

Based on difficulty of **factoring** large numbers (into large primes), i.e., $n = p \cdot q$ into p, q

- note that for RSA to be secure, both p and q must be large primes
- widely believed to hold true
  - since 1978, subject of extensive cryptanalysis without any serious flaws found
  - best known algorithm takes exponential time in security parameter (key length $|n|$)
- how can you break RSA if you can factor?

Current practice is using 2,048-bit long RSA keys (617 decimal digits)

- estimated computing/memory resources needed to factor an RSA number within one year

Length (bits)	PCs	Memory
430	1	128MB
760	215,000	4GB
1,020	342×10^6	170GB
1,620	1.6×10^{15}	120TB

RSA challenges

Challenges for breaking the RSA cryptosystem of various key lengths (i.e., |n|)

- known in the form RSA-`key bit length' expressed in bits or decimal digits
- provide empirical evidence/confidence on strength of specific RSA instantiations

Known attacks

- RSA-155 (**512-bit**) factored in **4 mo**. using 35.7 CPU-years or 8000 Mips-years (**1999**) and 292 machines
  - 160 175-400MHz SGI/Sun, 8 250MHz SGI/Origin, 120 300-450MHz Pent. II, 4 500MHz Digital/Compaq

- RSA-**640** factored in **5 mo**. using 30 2.2GHz CPU-years (**2005**)
- RSA-220 (**729-bit**) factored in **5 mo**. using 30 2.2GHz CPU-years (**2005**)
- RSA-232 (**768-bit**) factored in **2 years** using **parallel** computers 2K CPU-years (1-core 2.2GHz AMD Opteron) (**2009**)

Most interesting challenges

- prizes for factoring RSA-**1024**, RSA-**2048** is \$100K, \$200K – estimated at 800K, 20B Mips-centuries

Deriving an RSA key pair

- public key is pair of integers (e,n), secret key is (d, n) or d
- the value of n should be quite large, a product of two large primes, p and q
- often p, q are nearly 100 digits each, so n ~= 200 decimal digits (~512 bits)
  - but 2048-bit keys are becoming a standard requirement nowadays
- the larger the value of n the harder to factor to infer p and q
  - but also the slower to process messages
- a relatively large integer e is chosen
  - e.g., by choosing e as a prime that is larger than both (p − 1) and (q − 1)
  - why?
- d is chosen s.t. e · d = 1 mod (p − 1)(q − 1)
  - how?

Discussion on RSA

- Assume **p** = 5, **q** = 11, **n** = 5 · 11 = 55, **ϕ**(**n**) = 40, **e** = 3, **d** = 27
  - why encrypting small messages, e.g., **M** = 2, 3, 4 is tricky?
  - recall that the ciphertext is $\mathbf{C} = \mathbf{M}^3$ mod 55 for **M** in $\mathbf{Z}_{55}$

M	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
C	1	8	27	9	15	51	13	17	14	10	11	23	52	49	20	26	18	2
M	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36
C	39	25	21	33	12	19	5	31	48	7	24	50	36	43	22	34	30	16
M	37	38	39	40	41	42	43	44	45	46	47	48	49	50	51	52	53	54
C	53	37	29	35	6	3	32	44	45	41	38	42	4	40	46	28	47	54

Discussion on RSA

- Assume **p** = 5, **q** = 11, **n** = 5 · 11 = 55, **ϕ**(**n**) = 40, **e** = 3, **d** = 27
  - why encrypting small messages, e.g., **M** = 2, 3, 4 is tricky?
  - recall that the ciphertext is $\mathbf{C} = \mathbf{M}^3 \bmod 55$ for **M** in $\mathbf{Z}_{55}$

- Assume n = 20434394384355534343545428943483434356091 = p · q
  - can e be the number 434325345343453 6?

- Are there problems with applying RSA in practice?
  - what other algorithms are required to be available to the user?

- Are there problem with respect to RSA security?
  - does it satisfy CPA security?

Algorithmic issues

The implementation of the RSA cryptosystem requires various algorithms

- Main issues
  - representation of integers of arbitrarily large size, and
  - arithmetic operations on them, namely computing modular powers
- Required algorithms (at setup)
  - generation of **random numbers** of a given number of bits (to compute candidates **p**, **q**)
  - **primality testing** (to check that candidates **p**, **q** are prime)
  - computation of the **GCD** (to verify that **e** and **ϕ**(**n**) are relatively prime)
  - computation of the **multiplicative inverse** (to compute **d** from **e**)

Modular powers

Repeated squaring algorithm

- speeds up computation of $\mathbf{a^p}$ mod $\mathbf{n}$
- write the exponent $\mathbf{p}$ in binary
  - $\mathbf{p} = \mathbf{p}_{b-1}\,\mathbf{p}_{b-2} \ldots \mathbf{p}_1\,\mathbf{p}_0$
- start with $\mathbf{Q}_1 = \mathbf{a}^{\mathbf{p}_{b-1}}$ mod $\mathbf{n}$
- repeatedly compute
  $\mathbf{Q}_i = ((\mathbf{Q}_{i-1})^2 \text{ mod } \mathbf{n})\mathbf{a}^{\mathbf{p}_{b-i}}$ mod $\mathbf{n}$
- obtain $\mathbf{Q}_b = \mathbf{a^p}$ mod $\mathbf{n}$

In total **O** (log **p**) arithmetic operations

Example

- 3^{18} mod 19 (18 = 10010)
- $\mathbf{Q}_1 = 3^1$ mod 19 = 3
- $\mathbf{Q}_2 = (3^2$ mod $19)3^0$ mod 19 = 9
- $\mathbf{Q}_3 = (9^2$ mod $19)3^0$ mod 19 = 81 mod 19 = 5
- $\mathbf{Q}_4 = (5^2$ mod $19)3^1$ mod 19 =
  (25 mod 19)3 mod 19 = 18 mod 19 = 18
- $\mathbf{Q}_5 = (18^2$ mod $19)3^0$ mod 19 = (324 mod 19)
  mod 19 = 17·19 + 1 mod 19 = 1

Pseudo-primality testing

Testing whether a number is prime (**primality testing**) is a difficult problem

An integer **n** ≥ 2 is said to be a base-**x** **pseudo-prime** if

- $\mathbf{x}^{\mathbf{n}-1} \bmod \mathbf{n} = 1$ (Fermat's little theorem)

- Composite base-**x** pseudo-primes are rare
  - a random 100-bit integer is a composite base-2 pseudo-prime with probability less than 10^{-13}
  - the smallest composite base-2 pseudo-prime is 341
- Base-**x** pseudo-primality testing for an integer **n**
  - check whether $\mathbf{x}^{\mathbf{n}-1} \bmod \mathbf{n} = 1$
  - can be performed efficiently with the repeated squaring algorithm

Security properties

- Plain RSA is deterministic
  - why is this a problem?
- Plain RSA is also homomorphic
  - what does this mean?
  - multiply ciphertexts to get ciphertext of multiplication!
  - $[(m_1)^e \bmod N][(m_2)^e \bmod N] = (m_1 m_2)^e \bmod N$
  - however, not additively homomorphic

Real-world usage of RSA

- Randomized RSA
  - to encrypt message M under an RSA public key (e,n), generate a new random session AES key K, compute the ciphertext as [K^e mod n, $AES_K(M)$]
  - prevents an adversary distinguishing two encryptions of the same M since K is chosen at random every time encryption takes place
- Optimal Asymmetric Encryption Padding (OAEP)
  - roughly, to encrypt M, choose random r, encode M as $M' = [X = M \oplus H_1(r)\ ,\ Y = r \oplus H_2(X)\ ]$ where H_1 and H_2 are cryptographic hash functions, then encrypt it as $(M')^e$ mod n