CS579: Foundations of Cryptography

Spring 2023

Hash Functions

Instructor: **Nikos Triandopoulos**

STEVENS INSTITUTE of TECHNOLOGY
THE INNOVATION UNIVERSITY®

Message & source authentication

- Encryption alone does not protect against an attacker maliciously modifying sent data or masquerading as another user
- Need a way to ensure that
  - data reaches the destination intact, i,e., in its original form as sent by the sender; and
  - it is indeed coming from an authenticated source, i.e., the intended sender

Hash functions

- Map a **message of an arbitrary length** to a **n-bit output**
  - output known as the **fingerprint** or the message **digest**
  - **compression function**
    - a hash function mapping **m-bit messages** to **n-bit outputs** **with m > n**
- Examples
  - a function that maps strings to integers in $[0,2^{32-1}]$
  - $F(x) = A\,x + b \bmod q$, where $x = 0,1,\ldots,T$ where $T >> q$
  - hash function used in the hash table data structure

Motivation: hash functions as MACs?

- One (unrealistic) approach
  - assume a hash function h
  - assume an authentic channel (i.e., attacker cannot tamper with) for short messages
  - transmit a message M over the normal (insecure) channel
  - transmit the message digest h(M) over the secure channel
  - when receiver receives both M' and h(M), check if message is not modified
    - compute h(M') and check if h(M') = h(M)
- Insecure, in general
  - a hash function is a **many-to-one function**, so **collisions can happen**
  - hash collisions generally result in/correspond to attacks

Non-crypto Hash (1)

- Data $X = (X_0, X_1, X_2, \ldots, X_{n-1})$, each X_i is a bit
- **hash**$(X) = X_0 + X_1 + X_2 + \ldots + X_{n-1}$
- What is the compression of this hash?
- How can be attacked?

Non-crypto Hash (2)

- Cyclic Redundancy Check (CRC)
- Essentially, CRC is the remainder in a long division calculation
- Finding collisions is easy
  - operating over a small search space (e.g., modulo x^8+1)
- CRC sometimes mistakenly used in crypto applications (e.g., WEP)

Cryptographic hash functions: Informal view

basic cryptographic primitive

- maps **"objects"** to a **fixed-length** binary strings
- core security property: mapping **avoids collisions**
  - **collision**: distinct objects ($x \neq y$) are mapped to the same hash value ($H(x) = H(y)$)
  - although collisions **necessarily exist**, they are **infeasible to find**

important role in modern cryptography

- lie between symmetric- and asymmetric-key cryptography
- capture different security properties of "idealized random functions"
- qualitative stronger assumption than PRF

Hash functions

Map messages to short digests

- security parameter 1^n
- a **general** hash function H() maps
  - a message of an **arbitrary length** to a ***l*(n)-bit** string

- a **compression** (hash) function h() maps
  - a **long** binary string to a **shorter** binary string
  - an ***l'*(n)-bit string** to a ***l*(n)-bit string**, with ***l'*(n) > *l*(n)**

Cryptographic hash functions : Formal view

A hash function with output length $l()$ is a pair of PPT algorithms (Gen, H) s.t.:

- PPT Gen: on input 1^n, outputs a **key** s (where 1^n is implicit in s)
- H: on input key s and string $x \in \{0,1\}^*$, outputs a string $H^s(x) \in \{0,1\}^{l(n)}$

Notes

- **fixed-length** extension: H is defined for inputs $x \in \{0,1\}^{l'(n)}$ with $l'(n) > l(n)$
- generally, H() compresses some (or all) of its inputs
  - without compression collision resistance is trivial
- "key" s is needed for technical reasons
  - typically, not randomly selected and **publicly known** (even to the adversary!)

Collision resistance (CR): Definition

Hash-coll$_{\mathcal{A},\Pi}$**(n)** = 1 iff $x \neq x'$ & $H^s(x) = H^s(x')$

Hash function
$\Pi = (Gen, H)$

security parameter 1^n

We say that Π (or H) is **collision-resistant** if ∀PPT adversary $\mathcal{A}$, ∃a negligible function v(n) s.t.

$$\Pr[\text{Hash-coll}_{\mathcal{A},\Pi}(n) = 1] \leq v(n)$$

Weaker security notions

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

- **preimage resistant** (or **one-way**)
  - if given $y \in Y$ it is computationally infeasible to find a value $x \in X$ s.t. $H(x) = y$
- **2-nd preimage resistant** (or **weak collision resistant**)
  - if given a **uniform** $x \in X$ it is computationally infeasible to find a value $x' \in X$, s.t. $x' \neq x$ and $H(x') = H(x)$
- cf. **collision resistant** (or **strong collision resistant**)
  - if it is computationally infeasible to find two distinct values $x', x \in X$, s.t. $H(x') = H(x)$

Design framework for hashing

Domain extension via the Merkle-Damgård transform

- general design pattern for cryptographic hash functions
  - reduces collision resistance of general hash (no compression) functions to collision resistance of compression (hash) functions

  - thus, in practice, it suffices to realize a collision-resistant compression function h
  - compressing by 1 single bit is a least as hard as compressing by any number of bits!

Merkle-Damgård transform: Design

suppose that (Gen, h: $\{0,1\}^{2n} \rightarrow \{0,1\}^n$) is a collision-resistant compression function

consider the general hash function (Gen', H: $\mathcal{M} = \{x : |x| < 2^n\} \rightarrow \{0,1\}^n$), defined as below

Merkle-Damgård design

- Gen' is exactly as Gen
- H(x) is computed by applying $h^s()$ in a **"chained" manner** over n-bit message blocks

- pad x to define a number, say B, **message blocks $x_1, \ldots, x_B$**, with $|x_i| = n$
- set extra, final, message block **x_{B+1} as an n-bit encoding L of $|x|$**
- starting by initial digest **$z_0 = IV = 0^n$**, output **$H(x) = z_{B+1}$**, where **$z_i = h^s(z_{i-1} || x_i)$**

Merkle-Damgård transform: Security

suppose that (Gen, h: $\{0,1\}^{2n} \rightarrow \{0,1\}^n$) is a collision-resistant compression function

consider the general hash function (Gen', H: $\mathcal{M} = \{x : |x| < 2^n\} \rightarrow \{0,1\}^n$), defined as below

<u>Merkle-Damgård design</u>

<u>Thm</u>: If the compression function h is collision resistant, then the derived hash function H is collision resistant

Notes

- security holds independently of the exact compression factor of h (even other than 1/2)
- in practice, message space $\mathcal{M}$ is sufficiently large
  - contains messages of length that is exponentially large in security parameter

Proof (I)

Proof by reduction

- suppose H is not collision resistant; ∃PPT adversary $\mathcal{A}$ computing x ≠ x' s.t. H(x) = H(x')
- $x = x_1, \ldots, x_B, x_{B+1}=L$, $|x| = L$
- $x' = x'_1, \ldots, x'_{B'}, x'_{B'+1}=L'$, $|x'| = L'$
- case 1: L ≠ L'; since H(x) = H(x')
  - $h^s(z_B||L) = h^s(z'_{B'}||L')$, thus a collision of h

- case 2: L = L'; then B = B' and since x ≠ x', let j be the **largest** index in [1:B+1] s.t.
  - in its j-th invocation, function h^s is applied on **different** inputs $x_j||z_{j-1} \neq x'_j||z'_{j-1}$
  - thus, by j's maximality, it is $z_j = z'_j$, thus **a collision exists in the j-th "h^s-box"**

Schematic explanation of proof of reduction (I)

- hash function H employs as subroutine hash function h

Schematic explanation of proof of reduction (II)

- suppose that there exists $\mathcal{A}_{\mathcal{H}}$ that finds collision on H
- we construct $\mathcal{A}_h$ that finds a collision on h
  - by assumption, it doesn't exist

"Gen' is exactly as Gen"

argue as before

Cryptographic applications

Efficient MAC design

back to problem of designing **<u>secure MAC for messages of arbitrary lengths</u>**

- so far, we have seen two solutions
  - block-based "tagging"
    - based on PRFs
    - inefficient

  - CBC-MAC
    - also based on PRFs
    - more efficient

Hash-and-MAC: Design

generic method for designing **secure MAC for messages of arbitrary lengths**

- based on **CR hashing** and **any fix-length secure MAC**

- new MAC (Gen', Mac', Vrfy') **as the name suggests**
  - Gen': instantiate H, Mac, output (s, k)
  - Mac': **hash** message m into $h = H^s(m)$, output $\mathbf{Mac_k}$-tag t on h
  - Vrfy': **canonical** verification

Proof of security: Structure

by reduction

- MAC scheme Π' = (Gen', Mac', Vrfy') employs 2 crypto primitives
  - CR hash function Γ = (Gen_H, H)
  - secure MAC scheme Π = (Gen, Mac, Vrfy)
- assume that there exists adversary $\mathcal{A}'$ that breaks Π'
  - we will show that there exists
    - either, adversary $\mathcal{C}$ that breaks Γ
    - or, adversary $\mathcal{A}$ that breaks Π

Recall: MAC security

Game Mac-forge$_{\mathcal{A},\Pi}$(n) = 1 iff 1. $\mathrm{Vrfy}_k(m^*,t^*) = 1$ &
2. m^* not in $\mathcal{Q}$

MAC scheme
$\Pi = (\mathrm{Gen}, \mathrm{Mac}, \mathrm{Vrfy})$

security parameter 1^n

We say that Π is **secure** if for all PPT $\mathcal{A}$, there exists a negligible function negl so that

$$\Pr[\ \text{Mac-forge}_{\mathcal{A},\Pi}(n) = 1\ ] \leq \mathrm{negl}(n)$$

Recall: CR definition

Hash-coll$_{\mathcal{A},\Gamma}$**(n)** = 1 iff $x \neq x'$ & $H^s(x) = H^s(x')$

Hash function
$\Gamma = (Gen_H, H)$

security parameter 1^n

We say that Γ (or H) is **collision-resistant** if ∀PPT adversary $\mathcal{A}$, ∃a negligible function ν(n) s.t.

$$\Pr[\text{Hash-coll}_{\mathcal{A},\Pi}(n) = 1] \leq \nu(n)$$

Proof of security

assume that $\mathcal{A}'$ breaks Π'

Game Mac-forge$_{\mathcal{A}',\Pi'}$(n) = 1 iff 1. $\mathrm{Vrfy}'_k(m^*,t^*) = 1$ &
2. m^* not in $\mathcal{Q}$

MAC scheme
$\Pi' = (\mathrm{Gen}', \mathrm{Mac}', \mathrm{Vrfy}')$

security parameter 1^n

COLL: event that $H^s(m^*) = H^s(m_i)$ for some i

$\Pr[\mathcal{A}' \text{ wins}] = \Pr[\text{Mac-forge}_{\mathcal{A}',\Pi'}(n) = 1 \ \&\ \text{COLL}] + \Pr[\text{Mac-forge}_{\mathcal{A}',\Pi'}(n) = 1 \ \&\ \neg\text{COLL}]$
$\leq \Pr[\text{COLL}] + \Pr[\text{Mac-forge}_{\mathcal{A}',\Pi'}(n) = 1 \ \&\ \neg\text{COLL}]$

Proof of security

$\mathcal{C}$ is reduced to $\mathcal{A}'$

$$\Pr[\text{ Hash-coll}_{\mathcal{C},\Gamma}(n) = 1] = \Pr[\text{ COLL }] = \text{negligible}$$

COLL: event that $H^s(m^*) = H^s(m_i)$ for some i

Proof of security

$\mathcal{A}$ is reduced to $\mathcal{A}'$

$$\Pr[\text{Mac-forge}_{\mathcal{A},\Pi}(n) = 1] = \Pr[\text{Mac-forge}_{\mathcal{A}',\Pi'}(n) = 1 \ \& \ \neg\text{COLL}] = \text{negligible}$$

COLL: event that $H^s(m^*) = H^s(m_i)$ for some i

MAC based on hashing

- so far: MAC schemes are based on block ciphers
- can we construct a MAC based on CR hashing?
- first attempt: set tag $t = Mac_k(m) = H(k||m)$
  - intuition: given $H(k||m)$ it should be infeasible to compute $H(k||m')$, $m' \neq m$
  - insecure construction
  - most practical CR hash functions H() are based on the **Merkle-Damgård** design

- length-extension attack
  - knowledge of $H(m_1)$ makes it feasible to compute $H(m_1||m_2)$
  - assumes knowledge of length of message m_1; retrieve internal state s_k without k!

HMAC: Design

- hash-based secure MAC design (that eliminates length-extension attacks)
- $HMAC_k[m] = H^s[\ (k \oplus opad)\ ||\ H^s[\ (k \oplus ipad)\ ||\ m\ ]\ ]$
- two layers of hashing H^s intuition: **instantiation of hash & MAC paradigm**
- upper layer
  - $y = H^s(\ (k \oplus ipad)\ ||\ m\ )$
  - $y = H^{s\prime}(m)$
- lower layer
  - $t = H^s(\ (k \oplus opad)\ ||\ y'\ )$
  - $t = Mac'(k_{out}, y')$

HMAC: Details (1)

- hash-based secure MAC design (that eliminates length-extension attacks)
- $HMAC_k[m] = H^s[\ (k \oplus opad)\ ||\ H^s[\ (k \oplus ipad)\ ||\ \mathbf{m}\ ]\ ]$
- H^s is of Merkle-Damgård type
  - h^s compresses n + n' bits to n bits
  - $H^s(m)$ pads m s.t. last n' bits also encode |m| using d bits
  - n + d < n' s.t. tag t needs 2 h^s boxes

HMAC: Details (2)

- hash-based secure MAC design (that eliminates length-extension attacks)
- $HMAC_k[m] = H^s[\ (k \oplus opad)\ ||\ H^s[\ (k \oplus ipad)\ ||\ \mathbf{m}\ ]\ ]$
- the role of keys
- lower layer key needed for relaxing the assumption on H^s to being **<u>weak-CR</u>** function
  - oracle access to $H^s(IV,)$, random IV
- must use independent keys k_{in}, k_{out}
- opad, ipad: used to derive k_{in}, k_{out}

HMAC: Details (3)

- hash-based secure MAC design (that eliminates length-extension attacks)
- $HMAC_k[m] = H^s[\ (k \oplus opad)\ ||\ H^s[\ (k \oplus ipad)\ ||\ \mathbf{m}\ ]\ ]$
- the role of keys
- ipad
  - byte 0x36 repeated N' times (s.t. it is padded to n' bits)
- opad
  - byte 0x5C repeated N' times

HMAC security

- If used with a secure hash function (e.g., SHA-256) and according to the specification (key size, and use correct output), no known practical attacks against HMAC
  - recent attacks on MD5 did not affect the security of HMAC-MD5
  - because dependence on weak-CR, not CR!

Summary: Cryptographic hash functions (I)

- basic cryptographic primitive
  - general H
  - Vs. compression h

- main properties
  - collision resistance (CR): hard to find distinct inputs with same hash
  - Vs. 2nd-preimage / preimage resistance: hard to find specific collision / invert
- Merkle – Damgård transform
  - domain extension via chained compression
  - general design pattern

Summary: Cryptographic applications (II)

General design patterns for MACs

- "hash-and-MAC"
  - hybrid design
  - domain extension via fixed-length MACing
- HMAC
  - hash-based design
  - a special "hash-and-MAC" scheme
    - assumes Merkle – Damgård transform
    - fixes insecure h(k||m) scheme

Generic attacks

Birthday paradox

"In any group of 23 people (or more), it is (more) **likely** (than not) that **at least two** individuals have their birthday on the **same** day"

- based on probabilistic analysis of a random "balls-into-bins" experiment

  "k balls are each, independently and randomly, thrown into one out of m bins"

- what is the likelihood of event E = "**two balls land into the same bin**"?
  - by analysis: $\Pr[E] \approx 1 - e^{-k(k-1)/2m}$ (1)
  - if **$\Pr[E] = 1/2$**, Eq. (1) gives **$k \approx 1.17\, m^{1/2}$**
  - thus, for m = 365, k is around 23 (!)
    - assuming a uniform birth distribution

Birthday attack: Core idea

- applies "birthday paradox" against cryptographic hashing
- exploits the likelihood of finding collisions for hash function h using a **randomized** search, rather than an **exhausting** search
- analogy
  - k balls: distinct messages chosen to hash
  - m bins: number of possible hash values
  - <u>independent & random</u> throwing
    - how is this achieved?
    - message selection, hash mapping

Birthday attack: Algorithm

Find a collision for a cryptographic hash function h, as follows:

- randomly generate a sequence of messages $X_1, X_2, X_3, \ldots$
- for each X_i compute and store $y_i = h(X_i)$, and test whether $y_i = y_j$ for some $j < i$
- stop as soon as a collision has been found

Birthday attack: Probabilistic analysis

- for m hash values, message X_i h-collides with none of $X_1,\ldots,X_{i-1}$ with probability

$$1 - (i - 1)/m$$

- after k hashes, the attack fails (no collisions are found) with probability

$$F_k = (1 - 1/m)\,(1 - 2/m)\,(1 - 3/m)\,\ldots\,(1 - (k - 1)/m)$$

- using the standard approximation $1 - x \approx e^{-x}$

$$F_k \approx e^{-(1/m + 2/m + 3/m + \ldots + (k-1)/m)} = e^{-k(k-1)/2m}$$

- attack succeeds with probability at least (lower bound for uniform bin selection)

$$p = 1 - e^{-k(k-1)/2m}$$

  - (1) for $p=1/2$, $k \approx 1.17\, m^{1/2}$; and (2) for $m = 365$, $p=1/2$, k is around 24

Collision finding Vs. ball throwing: Reconcile discrepancies

Do independent & random throws model hashing via a specific function h?

Not perfectly; two possible ways to justify that our analysis is "accurate enough"

- consider an **idealized** hashing model
  - h behaves as a **truly random** function
  - hashed messages give random values
- choose messages **uniformly at random**
  - if h is **CR**, values h(x) are "**evenly distributed**"
- birthday attack (in real life)
  - randomly choose a (distinct) M_i, i = 1, 2,…
  - compute $y_i = h(M_i)$; check if $y_i = y_j$ for some $j < i$
    - stop as soon as a collision has been found

What birthday attacks mean in practice…

- approximate number of hash evaluations for finding hash collisions with prob. p for various digest lengths (or hash ranges H)

Bits	Possible outputs (2 s.f.) (H)	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}

- additionally, for large enough |H|=m, it can be approximated that **the first hash collision** will be found **on average** after **$t(m) = 1.25(m)^{1/2}$** hash evaluations

Overall: Generic attacks against cryptographic hashing

- assume a CR function h producing hash values of size l(n)
- **brute-force** attack
  - evaluate h on $\mathbf{2^{l(n)} + 1}$ 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 **SQT(#distinct inputs)**, it's **more likely** to find a collision!
  - thus, in order to get **k-bit security**, we (**at least**) need **hash values of length 2k**

Implementations of cryptographic hash functions

The Merkle-Damgård design framework (theory)

The Merkle-Damgård design framework

- preprocessing: message is divided into fixed-size blocks and padded
- hashing: applying a compression function f over the blocks using chaining
  - the compression function f combines the current message block with the hash of the previous message block to produce the hash of the current message block
- finalization: the digest of the message is the final hash output

Merkle's meta method

- any collision resistant compression function f can be efficiently extended to derive a collision-resistant hash function h
  - n bit output, r bit chaining variable
  - collision for h would imply collision for f for some stage i

Constructing hash functions in practice

- typically, using the Merkle-Damgård transform

  - (this precludes practical schemes being random oracles!)
  - reduces problem to design of CR compression functions
  - generic PRF-based compression schemes exist

The Davies-Meyer scheme

- assume PRF w/ key length n & block length l
- define h: $\{0,1\}^{n+l} \rightarrow \{0,1\}^{l}$ as **$h(k||x) = F_k(x)$ XOR x**
- h is CR, if F is an **ideal cipher**
  - idealized model that treats a PRF as a **random keyed permutation**
  - stronger than random oracle
  - some known block ciphers e.g., DES and triple-DES, are known not to be ideal ciphers!

Well known hash functions

- MD5 (designed in 1991)
  - output 128 bits, collision resistance completely broken by researchers in 2004
  - today (controlled) collisions can be found in less than a minute on a desktop PC
- SHA1 – the Secure Hash Algorithm (series of algorithms standardized by NIST)
  - output 160 bits, considered insecure for collision resistance
  - broken in 2017 by researchers in CWI
- SHA2 (SHA-224, SHA-256, SHA-384, SHA-512)
  - outputs 224, 256, 384, and 512 bits, respectively, no real security concerns yet
  - based on Merkle-Damgård + Davies-Meyer generic transforms
- SHA3 (Kessac)
  - completely new philosophy (sponge construction + un-keyed permutations)

Message Digest Algorithm 5 – MD5

- developed by Ron Rivest in 1991
- uses 128-bit hash values
- still widely used in legacy applications although considered insecure
- various severe vulnerabilities discovered
- collisions found by Marc Stevens, Arjen Lenstra and Benne de Weger

SHA-2 overview

Current hash standards

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

The random oracle model

Recall: Provable security in modern cryptography

Formal treatment for systematic design & analysis of security solutions

- security goals, threat model, assumptions, rigorous analysis
- 3 principles: **(A) formal definitions (B) precise assumptions (C) formal proofs**
  - given specific **assumptions**, a **concrete candidate scheme S** (implementation) is **proved secure** according to a **well-defined security concept**
  - (A) what is a (secure) MAC scheme? (B) what can/cannot an adversary $\mathcal{A}$ do? (C) for CR h and secure MAC, does "hash & MAC" securely implement a MAC?
- general proof structure: **reduction**
  - *unless S is secure, one of the made hardness assumptions is violated by* $\mathcal{A}$
  - e.g., if "hash & MAC" is insecure, then $\mathcal{A}$ can either break CR or MAC

Provable security: A possible challenge

- general proof structure: **reduction**
  - *unless S is secure, one of the made* hardness assumptions *is violated* by $\mathcal{A}$
    - type I: assumptions on underlying primitives used by S
    - type II: assumptions on threat model
- problem: often the above proof pattern cannot be used!
  - no reasonable assumptions of type I or II can be made to provide a reduction
  - real world Vs. abstract reduction model
    - discrepancies between **real-world** implementations of primitives or threats and **abstraction** in which reduction-based security proofs can be carried out
- **canonical example**: scheme S uses a **specific implementation** of hash function h but **no CR-type (or other) assumption** on h can provide a security proof for S

Provable security: Idealized models

- challenge in proving security of scheme S that employs scheme S'
  - no reasonable assumption on S' or $\mathcal{A}$ can provide a security proof for S
- naïve approach: look for other schemes or use scheme S (if S' looks "secure")
- middle-ground approach: fully rigorous proof Vs. heuristic proofs
  - employ **idealized** models that **impose** assumptions on S', $\mathcal{A}$
  - formally prove security of S in this idealized model
  - better than nothing…
- **canonical example**: employ the **random-oracle model** when using hashing
  - a cryptographic hash function h is treated as a **truly random** function

The random-oracle model

treats a cryptographic hash function h as a “black box” realizing a **random** function

- models h as a “secret service” that is publicly available for querying
  - anyone can provide input x and get output h(x)
  - nobody knows the exact functionality of the “box”
  - queries are assumed to be private
- interpretation of internal processing
  - if query x is new, then record and return a **random** value h(x) in the hash range
  - otherwise, answer **consistently** with previous queries on x

Using a random oracle h: Properties

- models h as a "secret service" that is publicly available for querying
  - **black-box access**: information leaks only via its API
  - **consistent** & **private querying**
  - **random hashing**
- in proofs by reduction (reduction $\mathcal{A}'$ using adversary $\mathcal{A}$)
  - probability is taken (also) over random choice of uniform h
  - in simulating oracle h (accessed by $\mathcal{A}$) $\mathcal{A}'$ can exploit the above properties
    - if x has not been queried before, h(x) is uniform (cf. PRG value G(x))
    - if $\mathcal{A}$ queries h on x, $\mathcal{A}'$ learns x (**extractability**)
    - $\mathcal{A}'$ can select answer h(x) to query x as long as it's uniform (**programmability**)

Recall: PRF – security

b = 0 when $\mathcal{D}$ thinks that its oracle is f()

b = 1 when $\mathcal{D}$ thinks that its oracle is $F_k()$

$$| \Pr[\mathcal{D}^{F(k,)}(1^n) = 1] - \Pr[\mathcal{D}^{f()}(1^n) = 1] | \leq negl(n)$$

$\mathcal{D}$ behaves the same no matter what its oracle is!

Random-oracle model Vs. PRF

- random-oracle model
  - models publicly-known & deterministic cryptographic hashing
  - used as black box in constructions (& analysis)
  - in practice, instantiated by a concrete scheme
- PRF
  - models keyed functions that produce pseudorandom values if keys are secret
  - oracle access to a uniform f is used as a means to define security of PRFs
  - PRFs are generally not random oracles

Recall: PRG – security

b = 0 when $\mathcal{D}$ thinks that its input X is random

b = 1 when $\mathcal{D}$ thinks that its input X is pseudorandom

$$| \Pr[\mathcal{D}(G(s)) = 1] - \Pr[\mathcal{D}(r) = 1] | \leq negl(n)$$

$\mathcal{D}$ behaves the same no matter what its input is!

Power of random oracles

consider a random oracle h

- h can be used as a PRG (assuming h expands its input)
  - $| \Pr[D(h(s)) = 1] - \Pr[D(r) = 1] | \leq negl(n)$
  - querying for h(s) happens with negligible probability
- h is a CR hash function (assuming h compressed its input)
  - why?
- h can provide a PRF (assuming inputs and outputs of 2n and n, respectively)
  - $F_k(x) = h(k||x)$
  - $| \Pr[D^{h(),F(k,\ )}(1^n) = 1] - \Pr[D^{h(),f()}(1^n) = 1] | \leq negl(n)$
  - why?

Random-oracle methodology

1. design & analyze using random oracle h; 2. instantiate h with specific function h'

- how sound is such an approach? on-going debate in cryptographic community
- pros (proof in random-oracle model better than no proof at all)
  - leads to significantly **more efficient** (thus practical) schemes
  - design is **sound**, subject to limitations in instantiating h to h'
  - at present, only **contrived** attacks against schemes proved in this model are known
- cons (proofs in the standard model are preferable)
  - random oracles **may not exist** (cannot deterministically realize random functions)
  - real-life $\mathcal{A}$s see the code of h' (e.g., may find a shortcut for some hash values)
  - can construct scheme S, s.t. S is proven secure using h, but is insecure using h'
  - note: "h' is CR" Vs. "h' is a random oracle"

Constructing hash functions in practice

- typically, using the Merkle-Damgård transform

- (this precludes practical schemes being random oracles!)
- reduces problem to design of CR compression functions
- generic PRF-based compression schemes exist

The Davies-Meyer scheme

- assume PRF w/ key length n & block length l
- define h: $\{0,1\}^{n+l} \rightarrow \{0,1\}^{l}$ as **$h(k||x) = F_k(x)$ XOR x**
- h is CR, if F is an **ideal cipher**
  - idealized model that treats a PRF as a **random keyed permutation**
  - stronger than random oracle
  - some known block ciphers e.g., DES and triple-DES, are known not to be ideal ciphers!

Applications of cryptographic hashing

Message digest

- Short secure description of data primarily used to detect changes to a block of data

Message digest

- Provided through cryptographic hash functions which have multiple uses
  - message authentication (e.g., integrity checking)
  - user authentication (e.g., password protection), in communication protocols
  - in conjunction with public-key algorithms for both encryption and digital signatures
- Main properties of modern cryptographic hash functions
  - one way: fast to convert input to a digest but infeasible to infer input of a digest value
  - collision-resistance: many collisions exist but they are not obvious to find
    - i.e., it is infeasible to find a pair of inputs that produce the same digest

Application 1: Hash values as file identifiers

Consider a cryptographic hash function H applied on a file F

- the hash (or digest) H(F) 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

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

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: 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!

Application 2: The Merkle tree

- an alternative (to Merkle-Damgård) method to achieve domain extension

$\mathbf{d} = h(h_{14} \,||\, h_{48})$

$h(h_{12} \,||\, h_{34}) =$ $\mathbf{h_{14}}$

$\mathbf{h_{58}} = h(h_{56} \,||\, h_{78})$

$h(h_1 \,||\, h_2) =$ $\mathbf{h_{12}}$

let $h_i = h(F_i)$, $1 \le i \le n$

Motivation: Secure cloud storage

- Bob has files $f_1, f_2, \ldots, f_n$
- Bob sends to Amazon S3 (cloud storage service)
  - the hashes $h(r||f_1), h(r||f_2), \ldots, h(r||f_n)$
  - files $f_1, f_2, \ldots, f_n$
- Bob stores randomness r (and keeps it secret)
- Every time Bob **reads** a file f_1, he also reads $h(r||f_i)$ and verifies f_1 integrity
- Any problems with **writes**?

Cloud storage model

(1)

files = (F1, F2, …, F7, F8)

(2)

Cloud storage model

- attack by malicious server

(3)

files = (F1, F2, …, F7, F8)

(4)

F1' = #@$@!#^@$^… (altered)

Secure cloud storage model
- integrity protection via hashing

(5)

files = F = (F1, F2, …, F7, F8)

1. pre-process files
using CR hash function h

(6)

digest d is computed over all files

$|d| << |F|$

Secure cloud storage model
- how verification works

(7)

files = (F1, F2, …, F7, F8)

(8)

4. verification

"is F1' intact?"

Secure cloud storage model
- verification via hashing

(9)

verification

"is F1' intact?"

- user has
  - authentic digest d (locally stored)
  - file F1' (to be checked/verified as it can be altered)
  - **proof** (to help checking integrity, but it can be maliciously chosen)
- verification involves (performed locally at user)
  - combine the file F1' with the proof to re-compute candidate digest d'
  - check if d' = d
  - if yes, then F1 is intact; otherwise tampering is detected!

Overall: Data authentication via the Merkle tree

$F = (F_1, F_2, \ldots, F_8)$

$\mathbf{d} = h(h_{14} \,||\, h_{48})$

$h(h_{12} \,||\, h_{34}) = \mathbf{h_{14}}$

$\mathbf{h_{58}} = h(h_{56} \,||\, h_{78})$

$h(h_1 \,||\, h_2) = \mathbf{h_{12}}$

let $h_i = h(F_i)$, $1 \le i \le n$

Application 3: Digital envelops

Commitment schemes

- 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, x, 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

Security definition for commitment schemes

- 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 computes (randomness and) commitments C_1, C_2 of x_1, x_2
    - challenger randomly selects bit b
    - challenger gives C_b to adversary
    - adversary 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)

Motivation: Online auction

- 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?)

Online auction (cont'ed)

- 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

Example: 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

Application 4: Forward-secure key rotation

- Alice and Bob secretly communicate using symmetric key encryption
- Eve intercepts their messages and later breaks into Bob's machine to steal the shared key

Application 5: 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.

Password storage

Identity	Password
Jane	qwerty
Pat	aaaaaa
Phillip	oct31witch
Roz	aaaaaa
Herman	guessme
Claire	aq3wm$oto!4

Plaintext

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

Concealed

Distribution of password types

Hashing passwords is not enough

An immediate control against password leakage through stolen password files, involves concealing passwords stored at the authentication server via hashing

Why are offline dictionary attacks quite effective using leaked hashed passwords in practice?

- Because the password space is actually small and predictable

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

Salting and iterative hashing

How do salts help?

- Since salts are not known in advance, no precomputation of hashed dictionaries helps

What does repeated password hashing (say 1000 times) offer?

- A trade-off between security and efficiency

Application 6: Digital signatures & hashing (looking ahead)

- Very often digital signatures are used with hash functions
  - the hash of a message is signed, instead of the message itself
  - "hash & sign" (similar to "hash & MAC")

Signing message M

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

Verifying signature σ

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