EECS 388

Introduction to Computer Security

Lecture 3:

Randomness and Pseudorandomness

September 5, 2023
Prof. Halderman

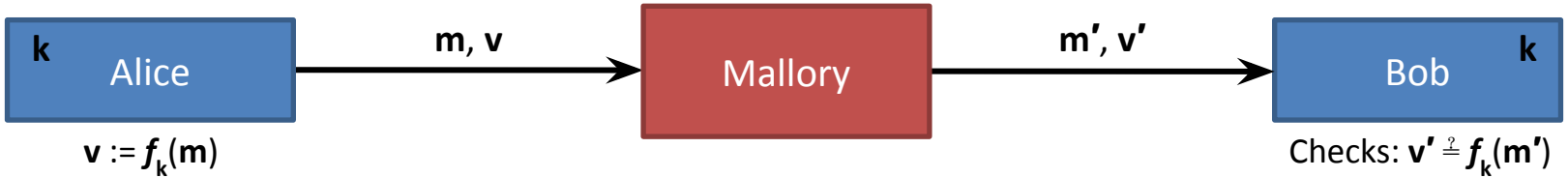
Review: Message Integrity

Problem: Integrity of message from Alice to Bob over an ***untrusted channel***

Approach: Alice must append bits to message that only Alice (or Bob) can make

Ideal solution: **Random functions**

Practical solution: **Pseudorandom functions (PRFs)**



$f_k()$ is a PRF if it's practically indistinguishable from a random function (unless you know **k**)

Today's lecture: What are some actual functions that (*we hope*) behave as PRFs?
Where do these random keys **k** come from?
What else are PRFs useful for?

Cryptographic Hashes

Cryptographic Hash Function

Fixed function ***H***(). ***No key*!**

Input: arbitrary length data

Output: fixed size digest (**n** bits)

Properties of strong hash functions:

Preimage resistance

Given output **h**, hard to find input **m** s.t. **h** = ***H***(**m**)

Second-preimage resistance

Given input $\mathbf{m}_1$, hard to different $\mathbf{m}_2$ s.t. ***H***($\mathbf{m}_1$) = **H**($\mathbf{m}_2$)

Collision resistance

Hard to find any pair of inputs $\mathbf{m}_1$,$\mathbf{m}_2$ s.t. ***H***($\mathbf{m}_1$)=**H**($\mathbf{m}_2$)

Note: Collisions exist [why?], but should be hard to find

Computing hashes with **OpenSSL:**

```
$ echo "hello world 1" | openssl dgst -sha256
07bafbe63a0e7c57c572aedf1c228022
b537e28785013d7be017fc78731a8cc5
$ echo "hello world 0" | openssl dgst -sha256
8bfa9a398e97152beaaf385847808ad2
d828c1c7251f1a45bc7697723827e7e7
```

Observe how changing even a single bit of the input produces output that appears completely unrelated.

Annoying question:

Are existing hashes *actually strong*?

Annoying answer:

We don't know.

Candidates: ~~MD5~~, ~~SHA-1~~, **SHA-256**, SHA-512, SHA-3

Constructing SHA-256

SHA-256 is a widely used hash function that is currently thought to be strong

Input: arbitrary length data
Output: 256-bit digest

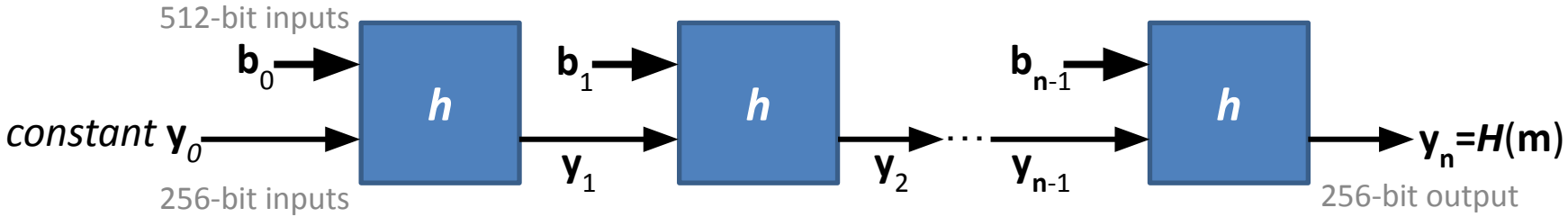
Built from a **compression function *h***:

Inputs: (256 bits, 512 bits), **Output:** 256 bits

"compresses" 768 bits to 256 bits using a complex internal function (details out of scope for 388)

Uses the **Merkle–Damgård (MD) construction** (illustrated below) to accept arbitrary-length input by repeatedly applying ***h***():

1. Pad input **m** to the next multiple of 512 bits (adds at least 1 bit, uses fixed algorithm [why?]) and split into 512-bit blocks: $\mathbf{b}_0, \mathbf{b}_1, \ldots \mathbf{b}_{n-1}$
2. $\mathbf{y}_0$:= <*256-bit constant*>
   $\mathbf{y}_1 := \boldsymbol{h}(\mathbf{y}_0,\mathbf{b}_0) \ldots \mathbf{y}_i := \boldsymbol{h}(\mathbf{y}_{i-1},\mathbf{b}_{i-1})$
3. Return $\mathbf{y}_n$ which is defined to be **SHA-256(m)**



MD Hash Pitfall: Length Extension Attacks

Merkle–Damgård hash functions are susceptible to **length extension attacks**:

Given **y** = ***H***(**x**) for some unknown **x**, attackers can calculate

z = ***H***(**x** // *padding* // **s**)

for arbitrary **s**.

concatenation

That is, given:

y = **SHA-256**(???)

An attacker can produce:

z = **SHA-256**(??? | *original pad* | suffix)

Note that this doesn't violate preimage, second-preimage, or collision resistance.

[But why is it a problem?]

Suppose Alice and Bob use this as a verifier:

v := **SHA-256**(**k** // **m**)

1. Alice sends **m** = "Please go to the bank."
2. Mallory doesn't know **k**, but *can* apply length extension to (**m**, **v**) to calculate **v'** for:
   m' = "Please go to the bank.[*original_pad*] Then transfer $10,000 to Mallory."
   (*original_pad* is some bytes beyond Mallory's control, but which the recipient might ignore)
3. Since **v'** is the correct verifier for **m'**, Bob will accept the modified message as valid

You'll explore how this is done in Project 1!

Practical Solution: HMAC

Message Authentication Code (MAC)

Designed to be used as a secure verifier:

Inputs: key, arbitrary length data
Output: fixed size digest (**n** bits)

HMAC construction turns any secure hash function ***H***() into a MAC:

$$\mathrm{HMAC}_k(m) = H(k \oplus c_1 \parallel H(k \oplus c_2 \parallel m))$$

XOR — constant 363636… — constant 5c5c5c… — concatenation

Design protects against length extension!

Example: HMAC-SHA-256 is an HMAC constructed using **SHA-256** for ***H***()

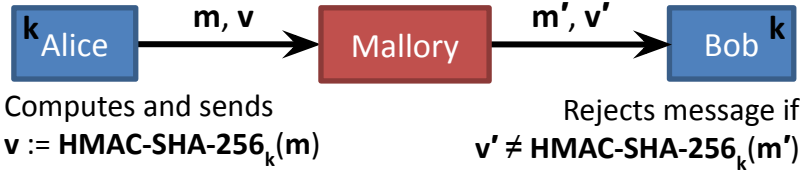
For practical purposes, we believe we can treat **HMAC-SHA-256** as a PRF

Can *reduce* PRF security of HMAC-SHA-256 to a (weaker) security property of SHA-256's compression function

```
$ echo "hi" | openssl dgst -sha256 -hmac Secr3t
8074cdfd007e5cfdc71c2c1cd393a5fe
fa890d7702956a1366a155d79d1cbe77
```

At last, this gives Alice and Bob a suitable approach for protecting message integrity:

v := HMAC-SHA-256$_k$(m)



Randomness and Pseudorandomness

How should we choose a "secret" key **k**?

Select a uniform random value [Why?]

Careful: People are often sloppy about what is "random"

True Randomness

Output of a *physical process* that is inherently unpredictable

Inherent in all physical systems (Heisenberg)

Absent in abstraction of a Turing machine

True randomness in software is scarce. Getting it requires careful engineering.

Can't just look at the *output* of a process and determine that it was random

Requires assessing the ***generation procedure***

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

Example: **Extract an unbiased random bit from a biased coin** (Von Neumann's method)

Since true randomness is expensive, often want to take a *small amount of it* and create a longer sequence that's "as good as random"

Pseudorandom Generator

Definition of a PRG

Review: **Pseudorandom Function (PRF)**

$f_k : \{0,1\}_m \rightarrow \{0,1\}_n$

m-bit input and **n**-bit output

Cannot practically distinguish $f_k(\mathbf{x})$ from random without knowing **k**

Pseudorandom Generator (PRG)

$g_k : \perp \rightarrow \{0,1\}_n$ for **n**=poly(|**k**|)

k is a truly random seed

No other inputs

Output is much larger than the input

(Can think of output as *stream of bits*)

Cannot practically distinguish $g_k()$ from a random stream of bits without knowing **k**

Security definition: (Similar to PRF definition)

1. Let **k** be a secret seed
2. Toss a coin (in secret) to get bit **b**
3. If ***b***=0, ***s*** := a truly random stream
   If **b**=1, **s** := $\mathbf{g_k}()$
4. Output **s** to Mallory
5. Mallory guesses **b** *in polynomial time**

We say ***g***() is a **secure PRG** if Mallory can't do meaningfully better than random guessing

*** Usual complexity-theoretic caveats…** no brute-forcing

Annoying question again:

Do PRGs *actually exist*?

Same annoying answer:

We don't know. (Would imply P≠NP.)

Building a PRG from a PRF

Given a secure PRF *f*(), we can construct a PRG ***g***() and prove that it's secure.*

Construction:

For some random key **k** and PRF ***f***(), define $g_k() := f_k(0) \;//\; f_k(1) \;//\; f_k(2) \;//\; \ldots$

Theorem (informal):

If ***f***() is a secure PRF and ***g***() is built from ***f***() by this construction, then ***g***() is a secure PRG.

You're not responsible for the proof, but here it is (*at right*), if you're interested:

(*Hard exercise: Build a PRF from a PRG)

Proof by contradiction: If ***g***() as constructed is *not* a secure PRG, ***f***() *can't be* a secure PRF:

1. If ***g***() is not a secure PRG, there exists an algorithm **M*** that can distinguish its output from random. (This is from the definition of a secure PRG.)

2. We can apply **M*** to construct an algorithm **M** to distinguish ***f***() from a random function ***r***():
   a. Query ***h***() with inputs 0, 1, 2, …
   b. Let **s** := ***h***(0) // ***h***(1) // ***h***(2) // …
   c. Apply **M*** to **s** and return the result.

3. If ***h***() is ***r***(), **s** is a random stream, so **M** outputs 0.
   If ***h***() is ***f***(), **s** is ***g***() (by construction), so **M** outputs 1.
   Thus, **M** wins the PRF game for ***f***().

This contradicts our assumption that ***f***() is a secure PRF. Therefore, ***g***() must be a secure PRG.

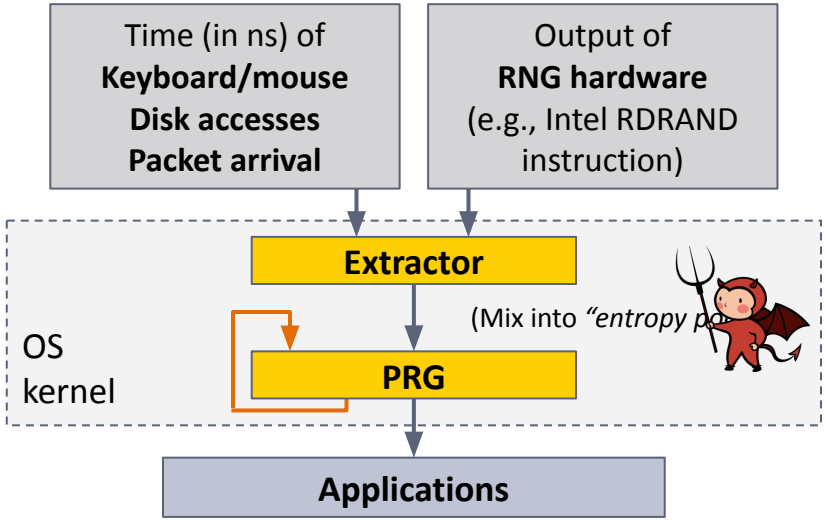
Getting Randomness

Randomness is **an input** to your program

Typically provided by the OS, via special APIs

OS continuously gathers inputs from physical sources that are hard for adversary to predict, "extracts" uniform bits from them.

[What if attacker can predict *some* of them?]



[What if an attacker learns internal state?]

If compromise is **transient**, can recover by adding more randomness.

How quickly can we recover, though?

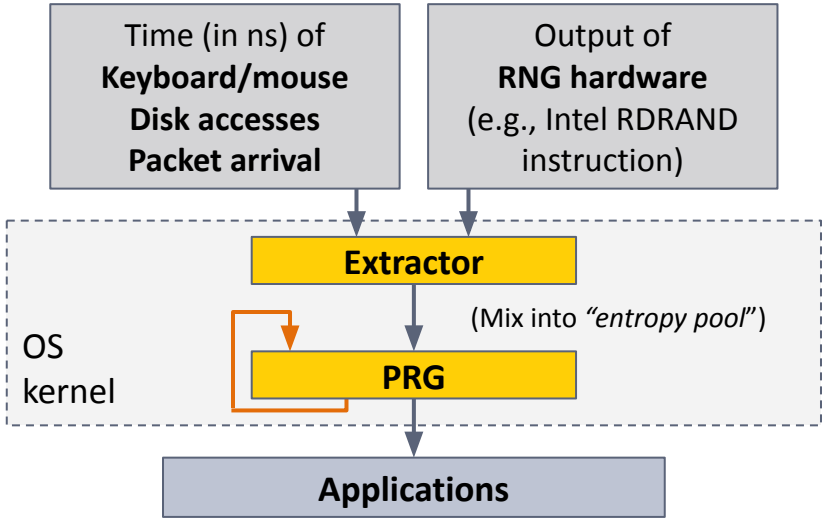
Getting Randomness

Randomness is **an input** to your program

Typically provided by the OS, via special APIs

OS continuously gathers inputs from physical sources that are hard for adversary to predict, "extracts" uniform bits from them.

[What if attacker can predict *some* of them?]



Generating 16 random bytes with **openssl**:

```
$ openssl rand -hex 16
4345b3cccecb66bef87f9289d72b8d2a
$ openssl rand -hex 16
578fbc36ad4d88ab7d98d7fd16e3737d
```

Caution! Not all "random" APIs are secure (or even unpredictable!) C's `rand()` function is notoriously bad, as are typical math packages.

Use APIs specified as suitable for cryptography:

C (Linux):
```
#include <sys/random.h>
getrandom(buf, size, 0);
```

Python:
```
import secrets
data = secrets.randbits(256);
```

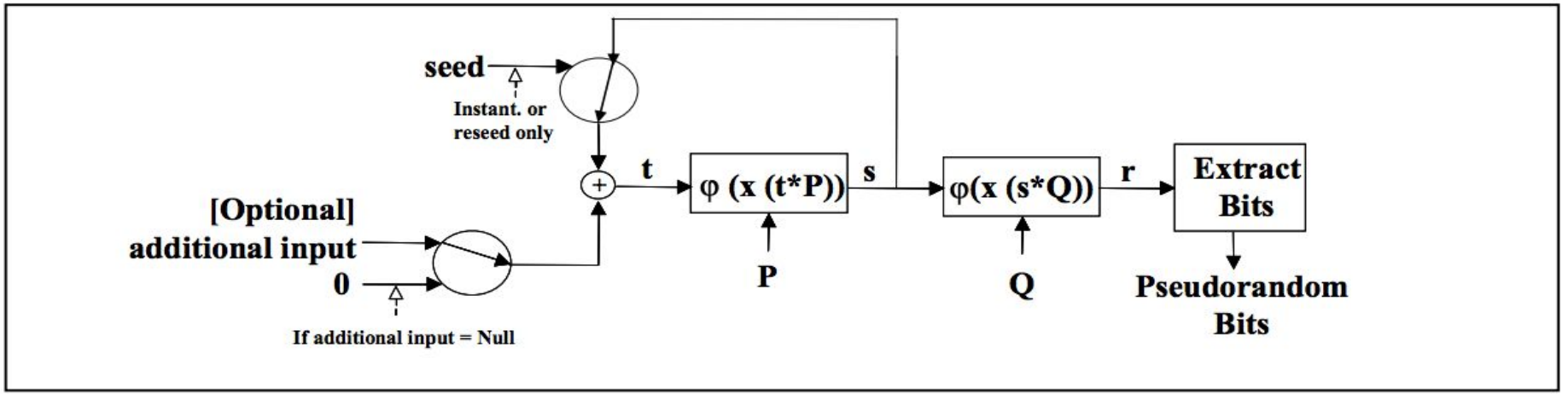
JavaScript:
```
const array = new Uint8Array(32);
self.crypto.getRandomValues(array);
```

Randomness as an Attack Target

Good randomness is needed everywhere in cryptography. RNG is very good attack target!

Dual-EC DRBG: 2006 NIST standard that NSA (allegedly) **backdoored**. Evidence in Snowden documents.

Construction allows for the existence of a secret backdoor key that can be used to recover the internal RNG state (and determine future output) given knowledge of small amount of past output.



Coming Up

Reminders:

Quiz on Canvas after every lecture
Lab Assignment 1 due Thursday at 6 p.m.
Project 1, Part 1 due Sept. 14 at 6 p.m.

Thursday

Confidentiality

Simple ciphers,
AES,
block cipher modes

Next Week

Wrap Up Crypto Unit

Combining confidentiality and integrity; Diffie-Hellman,
RSA encryption, digital signatures