Week 2: Lecture A
Message Integrity

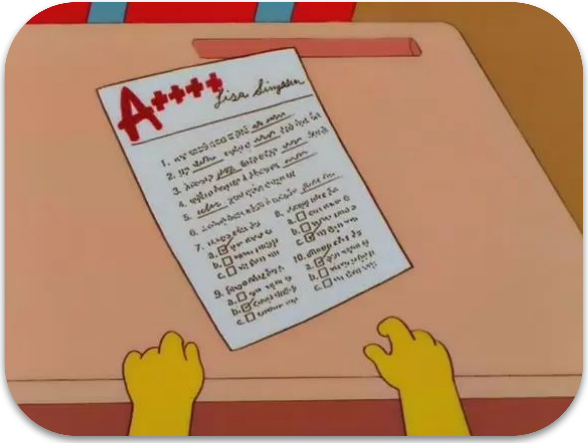
Tuesday, August 27, 2024

Reminders

- Be sure to join the course **Canvas** and **Piazza**
  - See links at top of course page
  - http://cs4440.eng.utah.edu
- Finish registering on **PollEverywhere**
  - Account must be `<yourUID>@utah.edu`
  - Location issues should be fixed
  - Sign in at https://pollev.com/cs4440
- Trouble accessing? See me after class!
  - Or email me at: snagy@cs.utah.edu

Reminders

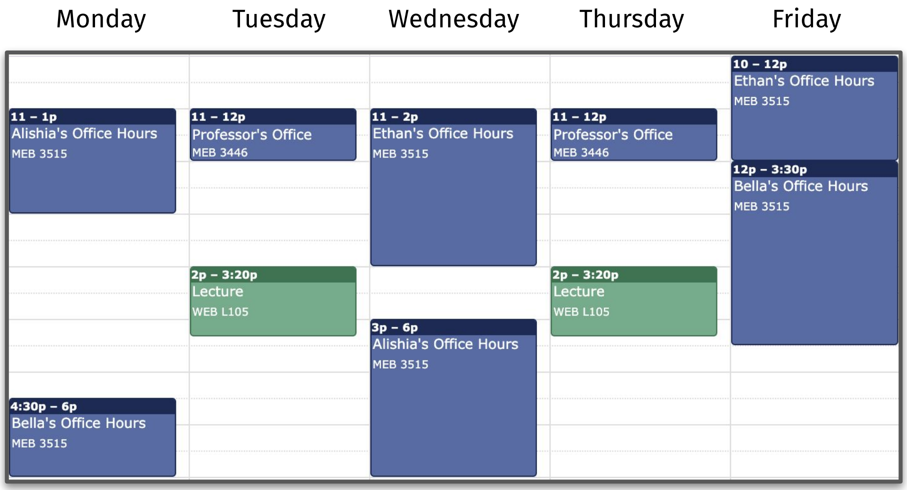
- First weekly **Lecture Quiz** was due last night
  - Next one opens **today** after lecture!
  - Due following **Monday by 11:59 PM**
  - Late submissions are not accepted



- You are welcome to consult your notes:
  - E.g., Wiki resources, the course VM, etc.
  - Designed to test understanding of key concepts
  - May see similar questions later in the semester 😃
  - **Lowest quiz score will be dropped**

Reminders

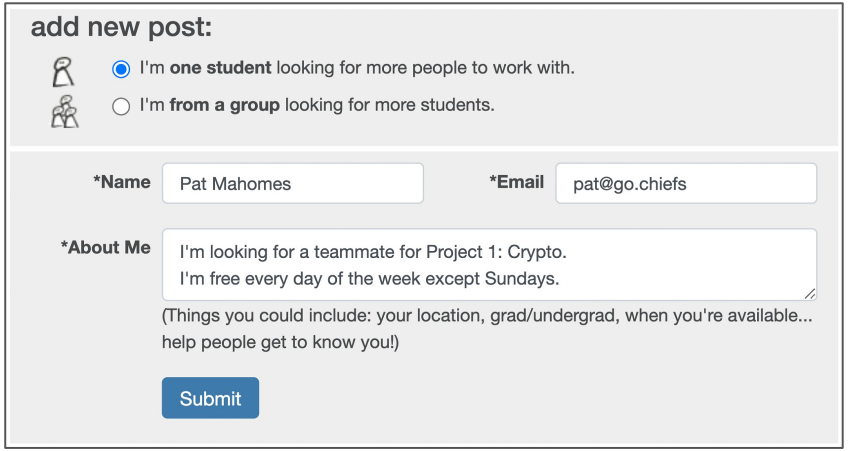
- Officers Hours schedule
  - http://cs4440.eng.utah.edu
  - Cancellations announced via **Piazza**
  - Busier near deadlines—**start early!**



Reminders

- Can work in **teams of up to two**
  - Find teammates on **Piazza**
  - Post on 

- Why work with someone else?
  - Pair programming
  - Divide and conquer
  - Two sets of eyes to solve problems
  - Teaching others helps you learn more
- Yes, you are free to work solo…
  - But we encourage you to team up!



Announcements

- **Project 1: Crypto** released (see **Assignments** page on course website)
  - **Deadline:** Thursday, September 19th by 11:59 PM

Project 1: Cryptography

Deadline: Thursday, September 19 by 11:59PM.

Before you start, review the course syllabus for the Lateness, Collaboration, and Ethical Use policies.

You may optionally work alone, or in teams of **at most two** and submit **one project per team**. If you have difficulties forming a team, post on **Piazza's Search for Teammates** forum. Note that the final exam will cover project material, so you and your partner should collaborate on each part.

The code and other answers your group submits must be entirely your own work, and you are bound by the University's Student Code. You may consult with other students about the conceptualization of the project and the meaning of the questions, but you may not look at any part of someone else's solution or collaborate with anyone outside your group. You may consult published references, provided that you appropriately cite them (e.g., in your code comments). **Don't risk your grade and degree by cheating!**

Complete your work in the **CS 4440 VM**—we will use this same environment for grading. You may not use any **external dependencies**. Use only default Python 3 libraries and/or modules we provide you.

Helpful Resources

- The CS 4440 Course Wiki
- VM Setup and Troubleshooting
- Terminal Cheat Sheet
- Python 3 Cheat Sheet
- PyMD5 Module Documentation
- PyRoots Module Documentation

Table of Contents:

Announcements

utahsec

See Discord for meeting info!

Announcements

- Due to the Utah football game, Thursday's class will be **hybrid**
  - Zoom link will be posted on **Piazza**
  - Feel free to join in-person if you can
  - We'll poll but **not record attendance**

Questions?

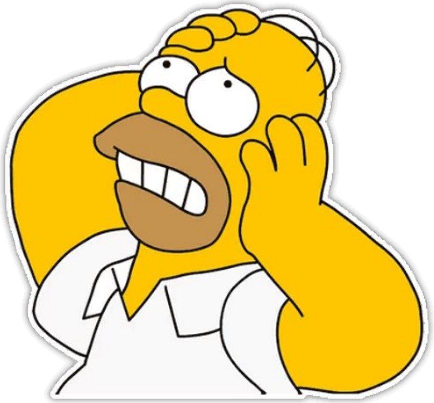
Last time on CS 4440…

Intro to Python
Debugging Code
Course VM Setup

Languages and Tools in CS 4440

- Projects cover a few languages and tools:
  - **Project1:** Python 3
  - **Project2:** C/C++, x86, GDB
  - **Project3:** SQL, HTML, JavaScript
  - **Project4:** Python 3, Wireshark

- This may seem daunting—but don't panic!



Languages and Tools in CS 4440

- Projects cover a few languages and tools:
  - **Project1:** Python 3
  - **Project2:** C/C++, x86, GDB
  - **Project3:** SQL, HTML, JavaScript
  - **Project4:** Python 3, Wireshark

- This may seem daunting—but don't panic!
  - Only using a **small subset** of their capabilities
  - We'll cover some basics in lecture as we go along
  - We'll post resources for you on the **CS 4440 Wiki**

Writing Python Scripts

- You'll be writing relatively simple scripts
  - No need for an IDE
  - IDEs can/will break things
- Recommended text editors:
  - VIM
  - Nano
  - Emacs
  - FeatherPad
  - **Many others—pick one you like!**

Variables

- Types you'll likely see:
  - Integer (`int`)
  - Float (`float`)
  - String (`str`)
  - Boolean (`bool`)
  - Custom classes (e.g., `md5`)
- Variable assignment:
  - Assignment uses the "=" sign
  - Value changed? **So does type!**

```
>>> x = 5
>>> print(type(x))
<class 'int'>

>>> x = "cs4440"
>>> print(type(x))
<class 'str'>
```

Variables

- Casting:
  - Pick a desired data type
  - “Wrap” your variable in it
  - **Re-casting** will change type!

```
>>> x = 5
>>> print(x, type(x))
5 <class 'int'>

>>> x = float(x)
>>> print(x, type(x))
5.0 <class float>
```

Strings

- You will use **strings** in many exercises
  - Super flexible to use and manipulate
  - We'll cover some basic conventions

- Basic string manipulation:
  - Length
  - Appending
  - Substrings

```
>>> x = "odoyle"
>>> print(len(x))
6

>>> print(x + "rules")
odoylerules

>>> print("odoy" in x)
True
```

Strings

- Other string manipulations:
  - Splitting by a delimiter
  - Stripping characters
  - Repeating characters

```
>>> x = "cs4440:fa23"
>>> print(x.split(':'))
['cs4440', 'fa23']

>>> print(x.strip(':'))
cs4440fa23

>>> print('A'*10)
AAAAAAAAAA
```

Byte Strings

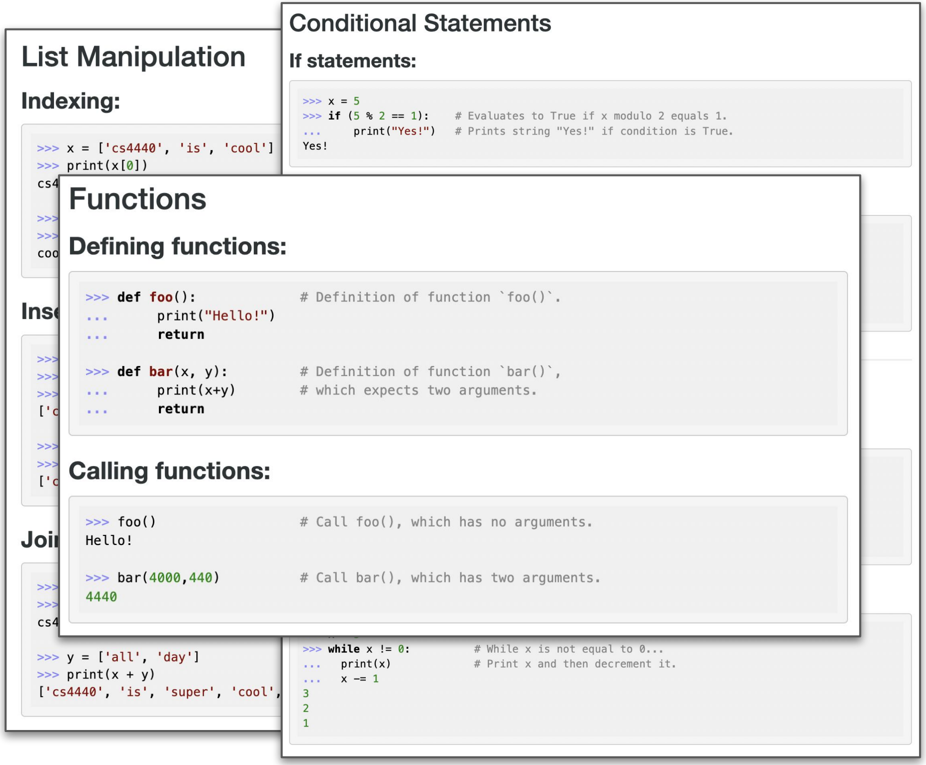
- Sometimes you will work with data as **bytes**
  - In Python, **byte strings** appear as `b'data'`
- Examples:
  - **Encoding** to a byte string
  - **Decoding** a byte string
  - Must keep the same codec (e.g., `utf-8`)
- Conceptually can be a little confusing
  - Functions `print()` and `type()` are your friends!

```
>>> x = "cs4440"
>>> x = x.encode('utf-8'))
>>> print(x, type(x))
b'cs4440' <class 'bytes'>

>>> y = x.decode('utf-8'))
>>> print(y, type(y))
cs4440 <class 'str'>
```

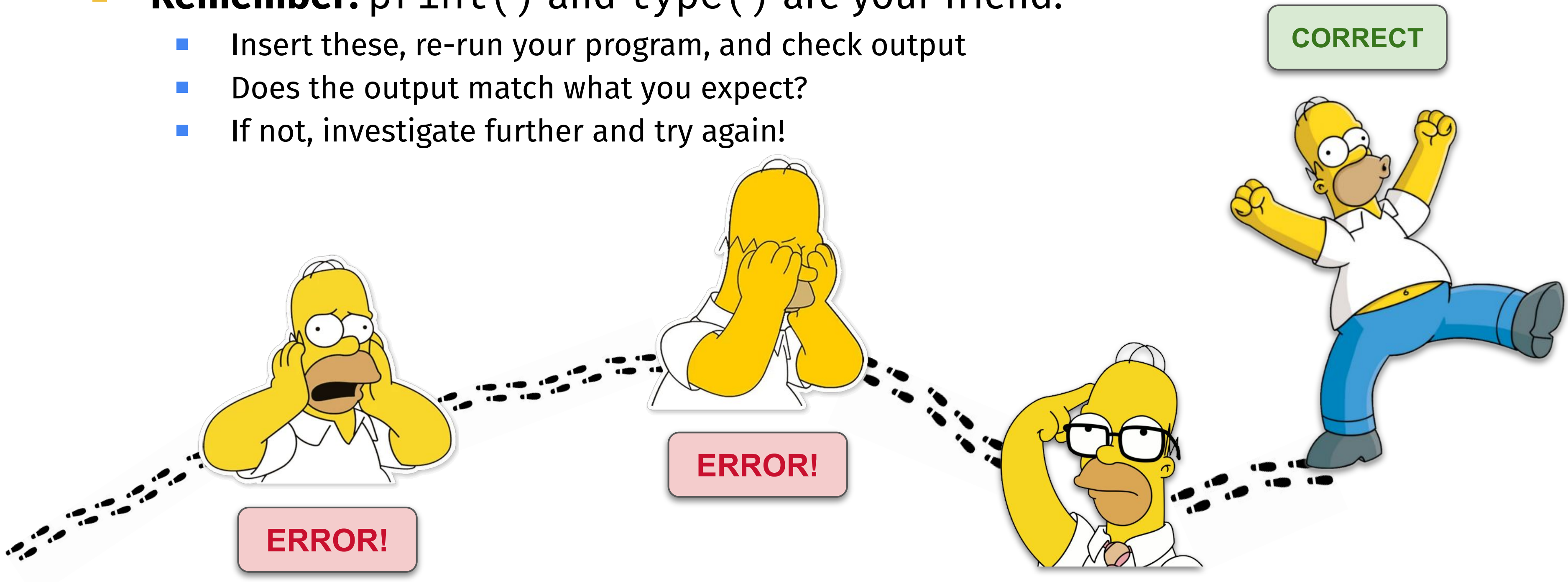
Other Key Concepts

- A few other concepts to review
  - Check these out in the **CS 4440 Wiki**
- Lists
  - Appending
  - Prepending
  - Insert, Remove
- Control Flow
  - Loops
  - If/Else Statements
- Functions



Debugging is a Process

- **Remember:** `print()` and `type()` are your friend!
  - Insert these, re-run your program, and check output
  - Does the output match what you expect?
  - If not, investigate further and try again!



Asking for Help

- **It's perfectly fine to ask for help**
  - That's what we / Piazza are here for!
- Help others help you! **Explain:**
  - What error code are you getting?
  - What do you think it means?
  - What fixes have you tried?
  - What fixes did not work?
- Avoid **"instructor private posts"**
  - We get **a lot** of these near deadlines
  - Impossible to keep up / help everyone!
  - We may un-private your post 🙂

Questions?

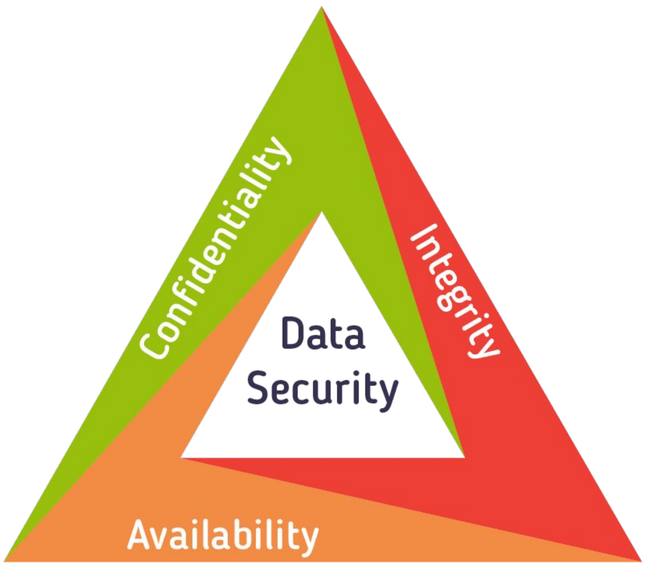
This time on CS 4440…

Message Integrity
Kerckhoffs's Principle
Pseudo-random Functions
Hashes and HMACs

Security Policies

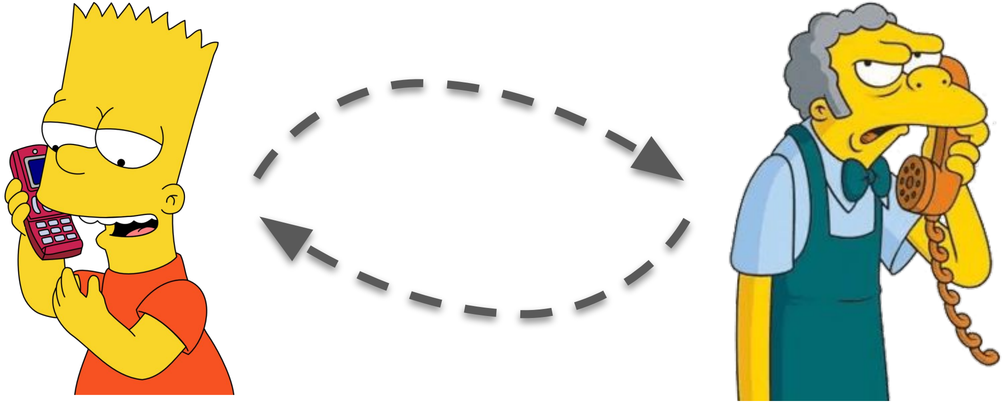
- What assets are we trying to protect?

- What properties are we trying to enforce?
  - Confidentiality
  - **Integrity <— you are here**
  - Availability
  - Privacy
  - Authenticity



Message Integrity

- Two parties want to communicate via an untrusted intermediary or medium



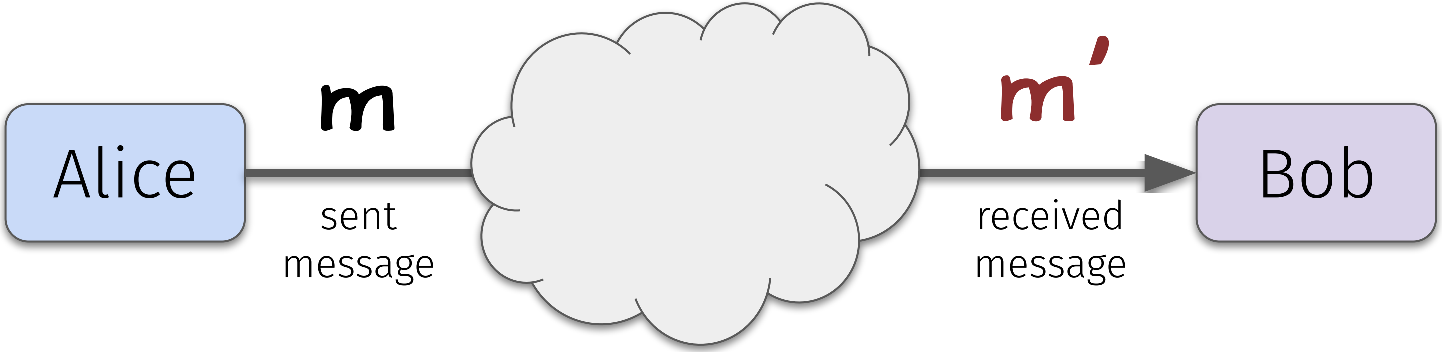
- **Problem:** ensure a message received by one party was sent by the other

Exercise: cheating the final exam

- **Goal:** communicate answers while taking the final exam

Exercise: cheating the final exam

- **Goal:** communicate answers while taking the final exam
- **Countermeasure:** randomized seating



Exercise: cheating the final exam

- **Goal:** communicate answers while taking the final exam
- **Countermeasure:** randomized seating + curved grading

Exercise: cheating the final exam

- Security policy
  - Message integrity

Exercise: cheating the final exam

- Security policy
  - Message integrity

- Threat model
  - Mallory can **see** and **tamper** Alice's messages, and **forge** her own messages
  - Mallory wants to trick Bob into **accepting a message Alice didn't send**

Exercise: cheating the final exam

- Security policy
  - Message integrity

- Threat model
  - Mallory can **see** and **tamper** Alice's messages, and **forge** her own messages
  - Mallory wants to trick Bob into **accepting a message Alice didn't send**

- Risk assessment
  - Very likely Mallory will strategically distort communication between Bob and Alice

Exercise: cheating the final exam

- Security policy
  - Message integrity

- Threat model
  - Mallory can **see** and **tamper** Alice's messages, and **forge** her own messages
  - Mallory wants to trick Bob into **accepting a message Alice didn't send**

- Risk assessment
  - Very likely Mallory will strategically distort communication between Bob and Alice

- Countermeasures
  - **Today's focus**

Message Integrity

Exercise: cheating the final exam

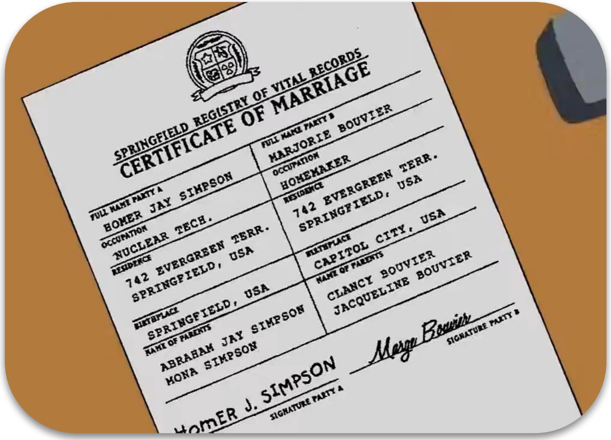
- **Goal:** communicate answers while taking the final exam
- **Countermeasure:** randomized seating + curved grading
- **Threat:** Mallory may **change** the message
- **Counter-countermeasure: ???**

Message Integrity

- **Goal:** communicate answers while taking the final exam
- **Approach:** include a **message-dependent message** with the sent message
  - Let **v** = ***f*** **(m)**

Including a Message-*dependent* Message

- Think of it as a **certificate of authenticity**
  - The output of a particular, pre-chosen **function**
- Unique to the original message
  - If message changed, **certificate will change too**
- Alice **sends this** along with her message
  - Bob recomputes this message-dependent code on the message he thinks came from Alice
  - Bob compares his code to the once he received



Message Integrity

- **Goal:** communicate answers while taking the final exam
- **Approach:** include a **message-dependent message** with the sent message
  - Let **v = *f* (m)**
- Bob accepts message if ***f* (m')** = **v'**

Message Integrity

- **Goal:** communicate answers while taking the final exam
- **Approach:** include a **message-dependent message** with the sent message
  - Let **v** = ***f*** **(m)**
- Bob accepts message if ***f*** **(m')** = **v'**
- If check **fails**, then **???**

$f(m') \mathrel{!=} v'$

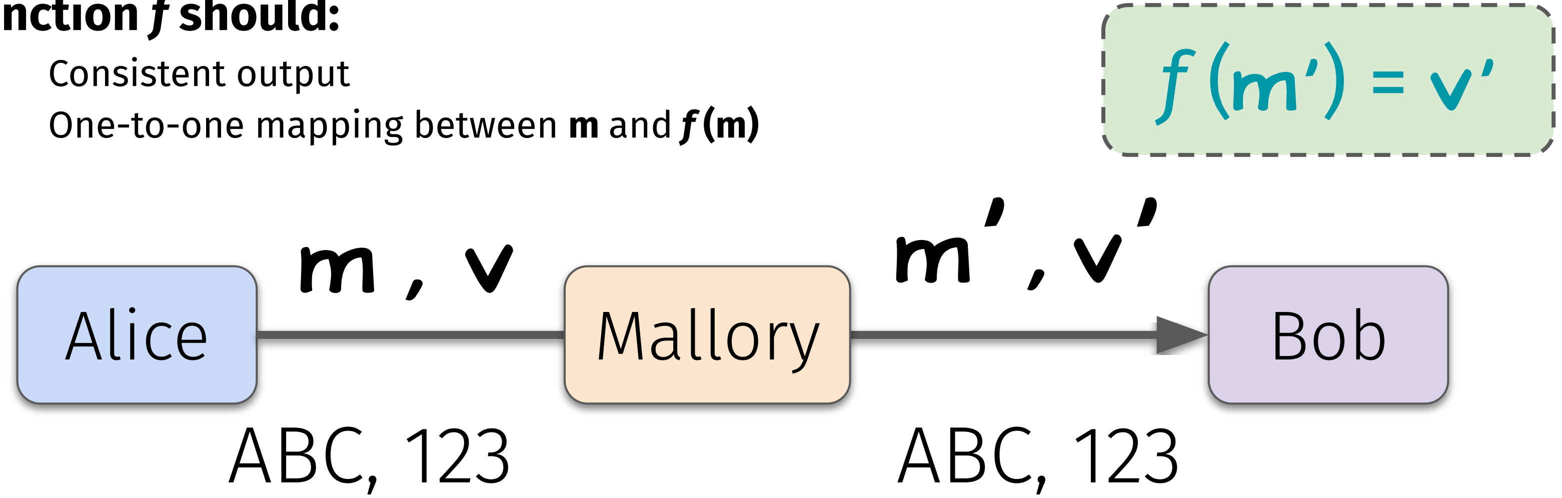
Message Integrity

- **Goal:** communicate answers while taking the final exam
- **Approach:** include a **message-dependent message** with the sent message
  - Let **v** = ***f*** **(m)**
- Bob accepts message if ***f*** **(m')** = **v'**
- If check **fails**, **m'** is **untrusted**

$f(m') \mathrel{!=} v'$

Function Properties

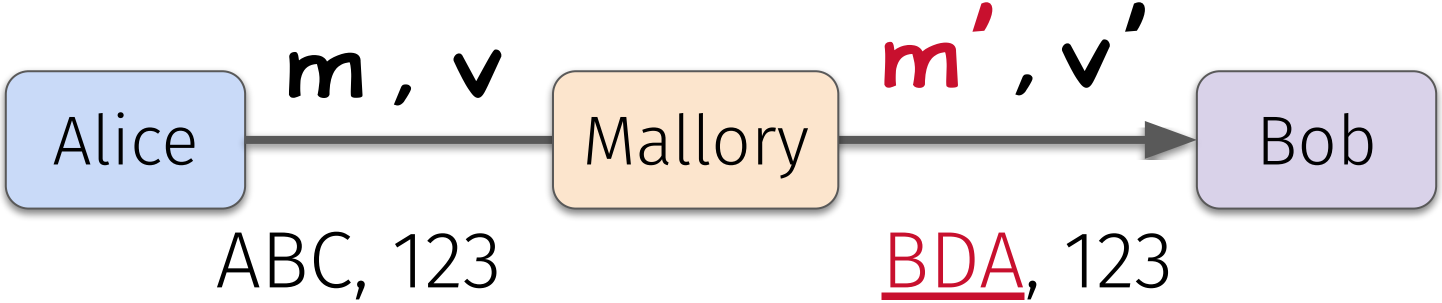
- **Goal:** communicate answers while taking the final exam
- **Approach:** include a **message-dependent message** with the sent message
  - Let **v** = ***f*** **(m)**
- **Function *f* should:**
  - Consistent output
  - One-to-one mapping between **m** and ***f*** **(m)**



Function Properties

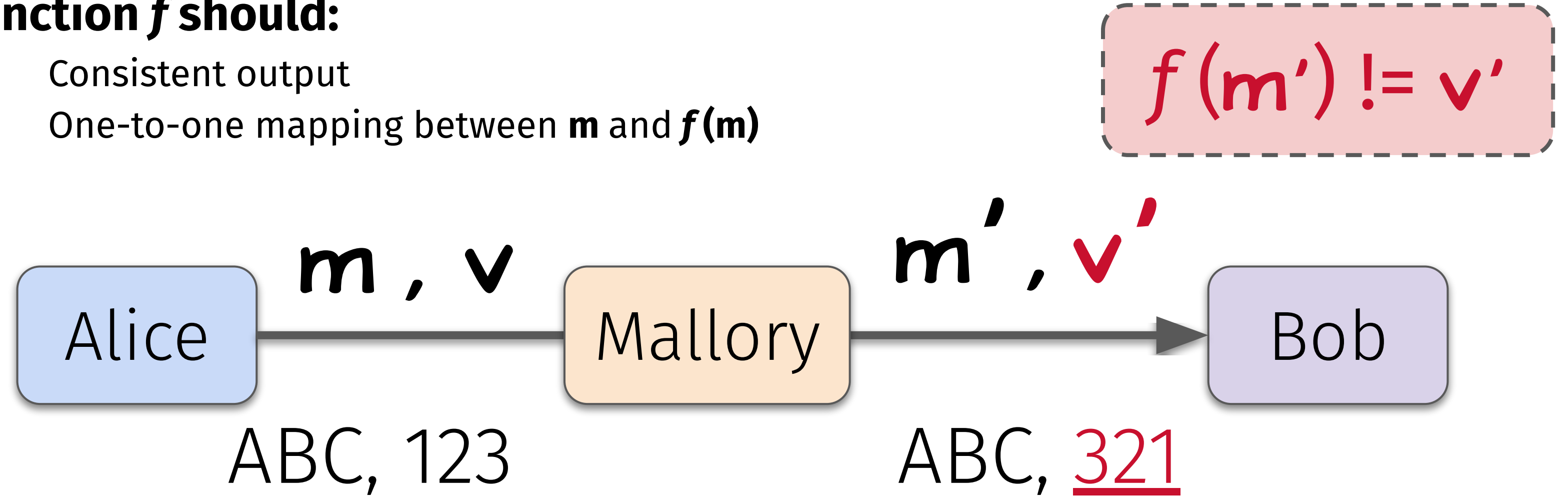
- **Goal:** communicate answers while taking the final exam
- **Approach:** include a **message-dependent message** with the sent message
  - Let **v** = ***f*** **(m)**
- **Function *f* should:**
  - Consistent output
  - One-to-one mapping between **m** and ***f*** **(m)**

f (m') != v'



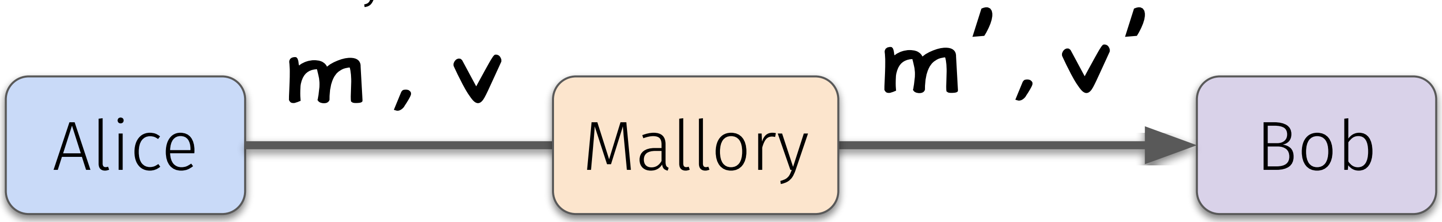
Function Properties

- **Goal:** communicate answers while taking the final exam
- **Approach:** include a **message-dependent message** with the sent message
  - Let **v** = ***f*** **(m)**
- **Function *f* should:**
  - Consistent output
  - One-to-one mapping between **m** and ***f*** **(m)**



Function Properties

- **Goal:** communicate answers while taking the final exam
- **Approach:** include a **message-dependent message** with the sent message
  - Let **v** = ***f*** **(m)**
- **Function *f* should:**
  - Consistent output
  - One-to-one mapping between **m** and ***f*** **(m)**
  - Be known to Mallory?



Is it okay if Mallory fully knows function f?

Yes

0%

No

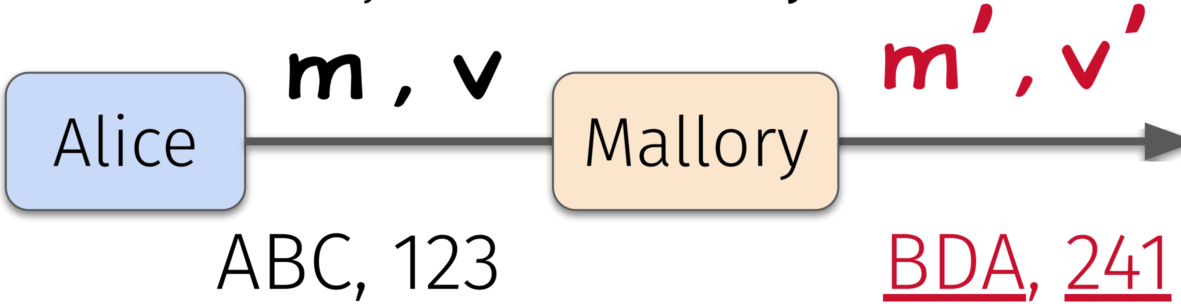
0%

Start the presentation to see live content. For screen share software, share the entire screen. Get help at **pollev.com/app**

Function Properties

- **Goal:** communicate answers while taking the final exam
- **Approach:** include a **message-dependent message** with the sent message
  - Let **v** = ***f*** **(m)**
- **Function *f* should:**
  - Consistent output
  - One-to-one mapping between **m** and **f (m)**
  - ~~Be known to Mallory?~~ **Be unknown to Mallory**

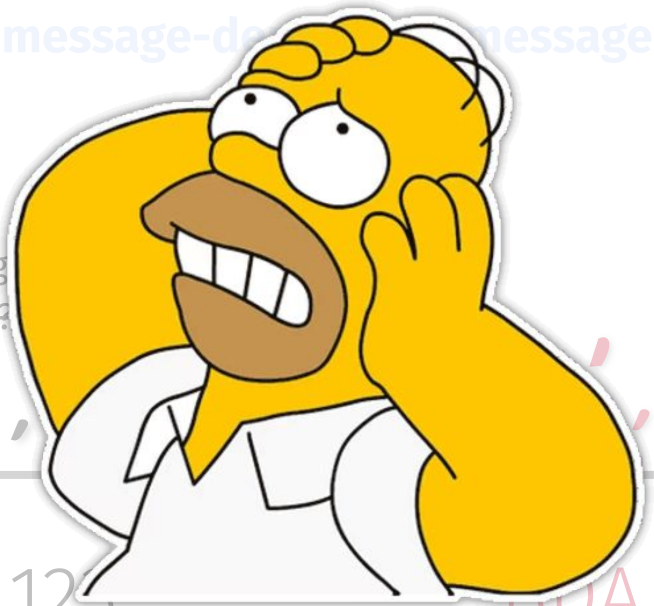
$f(m') = v'$



Function Properties

- **Goal:** communicate answers while taking the final exam
- **Approach:** include a **message-de**… **message** with the sent message
  - Let **v** = *f* **(m)**
- **Function *f* should:**
  - Consistent output
  - One-to-one mapping
  - ~~Be known to Mallory?~~

$f(m') = v'$



m , … , v′

Alice → Bob

ABC, 12… BDA, 241

Questions?

Choosing an Ideal Function for Message-*dependent* Messages

Kerckhoffs's Principles

To be secure, a **cryptosystem** must...

1. Be practically—if not mathematically—indecipherable.
2. **Not require total secrecy**, and not fail if captured.
3. Not require reliance on written notes (keys), and **be modifiable** by the corresponding parties at will.
4. Be applicable to telegraph communications.
5. Be portable and not need many to handle/operate.
6. **Be easy to use**, and not require a long list of rules.

Why Kerckhoffs's principles?

- Quantify probability that adversary (Mallory) **succeeds**

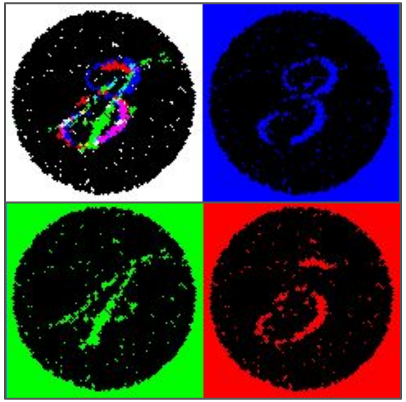
- Different people can use same system, different keys:
  - Alice and Bob use one key
  - Jack and Diane use another
  - Mutually distrusting parties

- Want to easily change key if something goes wrong

Candidate 1: Steganographic Encoding

- **Early form of message secrecy**
  - Messages hidden in ordinary objects
    - Images, paper, video, music, etc.
  - Not plainly visible to the human eye
    - Unless known what to look for

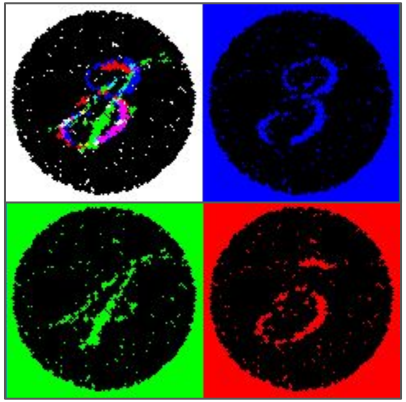
- **Examples:**
  - Different hidden numbers appear when viewed under different lights
  - “Invisible” ink



Candidate 1: Steganographic Encoding

- **Early form of message secrecy**
  - Messages hidden in ordinary objects
    - Images, paper, video, music, etc.
  - Not plainly visible to the human eye
    - Unless known what to look for

- **Examples:**
  - Different hidden numbers appear when viewed under different lights
  - “Invisible” ink

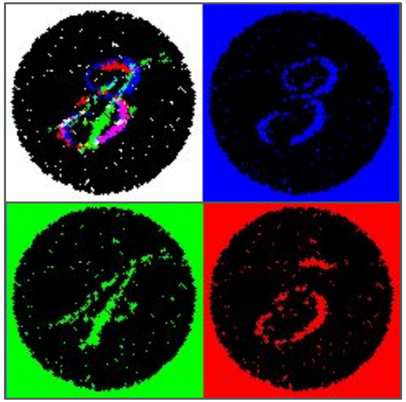


Impractical. **Why?**

Candidate 1: Steganographic Encoding

- **Early form of message secrecy**
  - Messages hidden in ordinary objects
    - Images, paper, video, music, etc.
  - Not plainly visible to the human eye
    - Unless known what to look for

- **Examples:**
  - Different hidden numbers appear when viewed under different lights
  - "Invisible" ink



Impractical. **Why?**

Insecure. **Why?**

Candidate 2: Random Functions

- **Random Functions:**
  - **Input:** Any size up to huge maximum
  - **Output:** Fixed size (e.g., 256 bits)
- Think of it as defined by a **massive lookup table** filled in by coin flips
- Maps inputs independently to any one of possible outputs

0	→ 0011111001010001...
1	→ 1110011010010100...
2	→ 0101010001010000...
...	→ ....

Set of all functions in the universe
(each outputs 256 random bits)

Candidate 2: Random Functions

- **Random Functions:**
  - **Input:** Any size up to huge maximum
  - **Output:** Fixed size (e.g., 256 bits)
- Think of it as defined by a **massive lookup table** filled in by coin flips
- Maps inputs independently to any one of possible outputs

0	→ 0011111001010001…
1	→ 1110011010010100…
2	→ 0101010001010000…
…	→ ………………………………

Provably Secure. **Why?**

Candidate 2: Random Functions

- **Random Functions:**
  - **Input:** Any size up to huge maximum
  - **Output:** Fixed size (e.g., 256 bits)
- Think of it as defined by a **massive lookup table** filled in by coin flips
- Maps inputs independently to any one of possible outputs

0	→	0011111001010001…
1	→	1110011010010100…
2	→	0101010001010000…
…	→	……………………………

Provably Secure. **Why?**

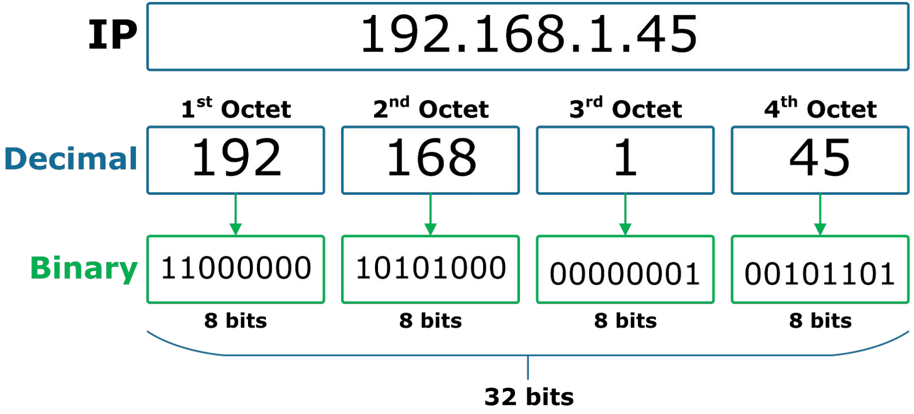
Impractical. **Why?**

Candidate 3: Pseudo-random Function Family (PRF)

- We want a set of functions that are **practical** but **“look” random**
- **“Looks random”** roughly means **two inputs that differ by 1** will very likely produce **two outputs that are far apart** (but no way to know just how far)
- **“Practical”** means efficiently computable
- Also want to **not rely on pre-sharing** all possible input–output pairings

Candidate 3: Pseudo-random Function Family (PRF)

- Can **decimal → binary** encoding be considered a pseudo-random function?



Is decimal-to-binary a PRF?

Yes

0%

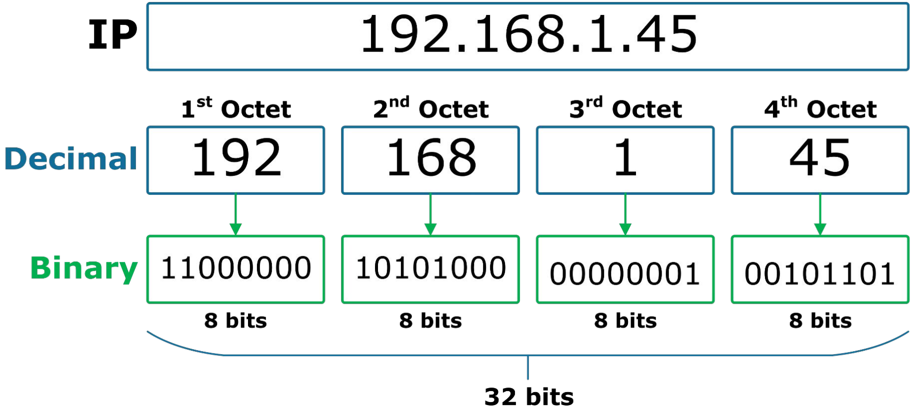
No

0%

Start the presentation to see live content. For screen share software, share the entire screen. Get help at **pollev.com/app**

Candidate 3: Pseudo-random Function Family (PRF)

- Can **decimal → binary** encoding be considered a pseudo-random function?



No! Small changes in **input** ***don't* lead to BIG changes** in the **output**.

Candidate 3: Pseudo-random Function Family (PRF)

- Start with a big **family of functions**
  - **Subset** of our huge random coin-flip table

Candidate 3: Pseudo-random Function Family (PRF)

- Start with a big **family of functions**
  - **Subset** of our huge random coin-flip table

- $\boldsymbol{f}_0, \boldsymbol{f}_1, \boldsymbol{f}_2, \ldots, \boldsymbol{f}_{(2\hat{}N)-1}$ all known to **Mallory**

Candidate 3: Pseudo-random Function Family (PRF)

- Start with a big **family of functions**
  - **Subset** of our huge random coin-flip table

- $f_0, f_1, f_2, \ldots, f_{(2^N)-1}$ all known to **Mallory**

- Use f_k where **k** is a secret value (or **key**)
  - Known **only** to **Alice** and **Bob**
  - **k** is (say) 256 bits, chosen randomly

Candidate 3: Pseudo-random Function Family (PRF)

- Start with a big **family of functions**
  - **Subset** of our huge random coin-flip table

- $f_0, f_1, f_2, \ldots, f_{(2\text{^}N)-1}$ all known to **Mallory**

- Use f_k where **k** is a secret value (or **key**)
  - Known **only** to **Alice** and **Bob**
  - **k** is (say) 256 bits, chosen randomly

How the functions work is not secret...

Candidate 3: Pseudo-random Function Family (PRF)

- Start with a big **family of functions**
  - **Subset** of our huge random coin-flip table
- $f_0, f_1, f_2, \ldots, f_{(2^N)-1}$ all known to **Mallory**
- Use f_k where **k** is a secret value (or **key**)
  - Known **only** to **Alice** and **Bob**
  - **k** is (say) 256 bits, chosen randomly

How the functions work is not secret…

But **which function** is chosen ***is*** secret

Formal Definition of a Secure PRF

- We say f is a **secure PRF** if Mallory can only beat this via random guessing

How the functions work is not secret…

But **which function** is chosen ***is*** secret

Formal Definition of a Secure PRF

- We say f is a **secure PRF** if Mallory can only beat this via random guessing
  - Function f_k is **practically indistinguishable** from a random function (unless **k** known)
- What is Mallory left with?

How the functions work is not secret...

But **which function** is chosen ***is*** secret

Formal Definition of a Secure PRF

- We say ***f*** is a **secure PRF** if Mallory can only beat this via random guessing
  - Function f_k is **practically indistinguishable** from a random function (unless **k** known)
- What is Mallory left with? **Brute Forcing**
  - Mallory would need to enumerate **every possible function** to figure out which is f_k
- How does this guarantee **security**?

How the functions work is not secret…

But **which function** is chosen ***is*** secret

Formal Definition of a Secure PRF

- We say ***f*** is a **secure PRF** if Mallory can only beat this via random guessing
  - Function $\boldsymbol{f_k}$ is **practically indistinguishable** from a random function (unless **k** known)
- What is Mallory left with? **Brute Forcing**
  - Mallory would need to enumerate **every possible function** to figure out which is $\boldsymbol{f_k}$
- How does this guarantee **security**?
  - Idea is that Mallory's **cost of brute-forcing is so high** that it's **computationally infeasible**

How the functions work is not secret…

But **which function** is chosen ***is*** secret

Message Integrity via PRFs

- **Goal:** communicate answers while taking the final exam
- **Approach:** use PRFs
  - Let $\boldsymbol{f}$ be a secure **PRF**
  - In advance, choose random **k** known only to Alice and Bob
  - Let $\mathbf{v} = \boldsymbol{f}_k(\mathbf{m})$
  - Bob checks that $\boldsymbol{f}_k(\mathbf{m^*}) == \mathbf{v^*}$, otherwise **m*** untrusted

Message Integrity for Multiple Messages

- **Goal:** send multiple messages with integrity
- **Problems:** ???

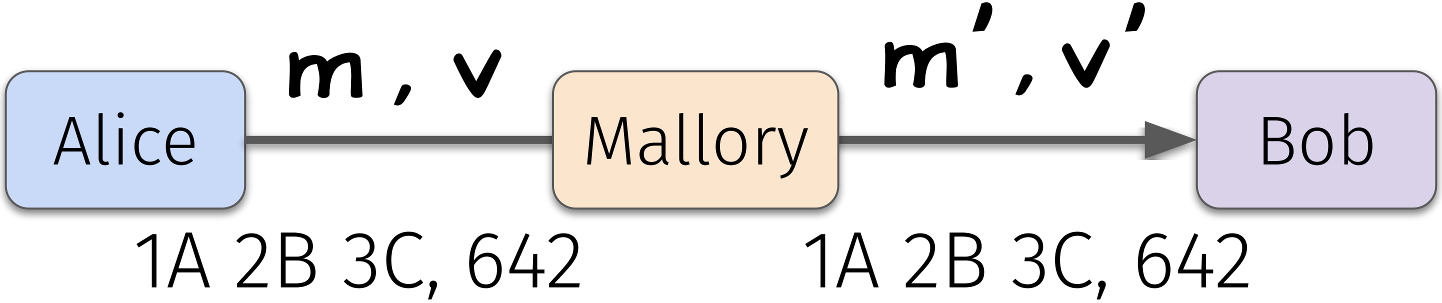
Message Integrity for Multiple Messages

- **Goal:** send multiple messages with integrity
- **Problems:**
  - **Replay attack:** Mallory injects messages from an earlier exam question
  - **Reordering attack:** Mallory answers question 1 after answering question 2

Message Integrity for Multiple Messages

- **Goal:** send multiple messages with integrity
- **Problems:**
  - **Replay attack:** Mallory injects messages from an earlier exam question
  - **Reordering attack:** Mallory answers question 1 after answering question 2

- **Countermeasures:** change **k**, add a sequence number

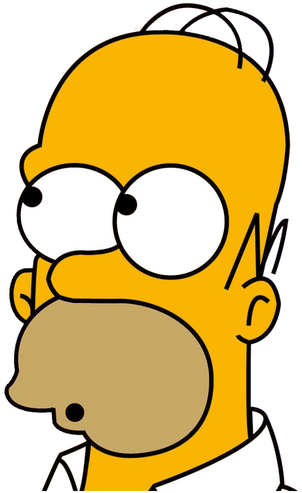


Existing PRFs

- Annoying question:
  - **Do PRFs actually exist?**

- Annoying answer:
  - We don't know for sure…
  - **But we strongly believe they do!**

- Best we can do:
  - Well-studied functions without problems **(yet)**
    - E.g., HMAC-SHA256

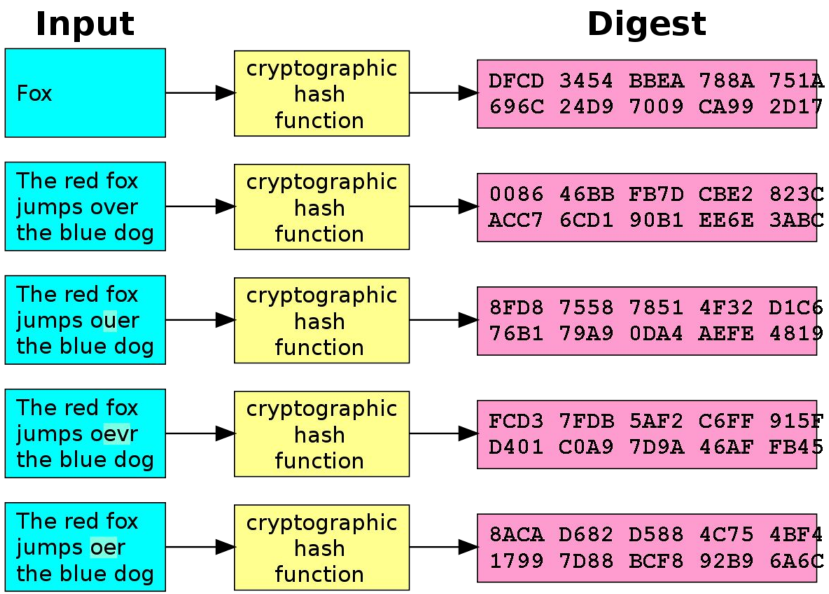


Questions?

Obsolete PRFs: Hash Functions

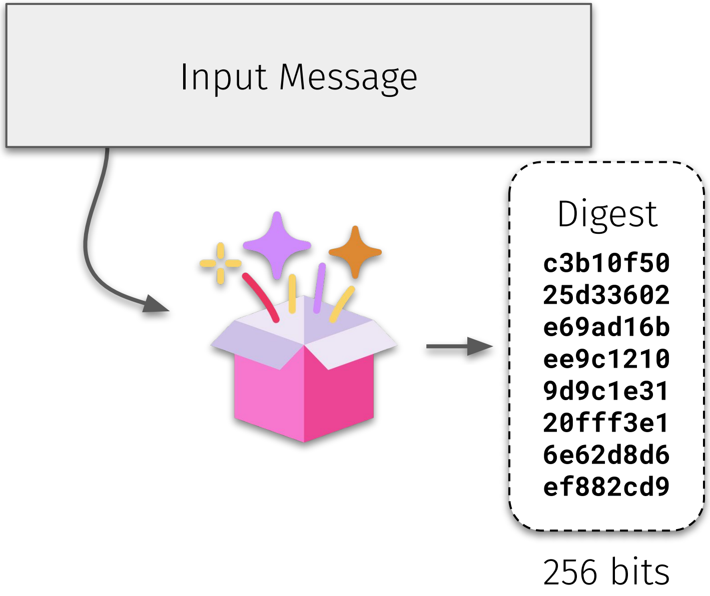
Cryptographic Hash Functions

- Based on idea of **compression**
- **Input:** arbitrary length data
- **Output:** fixed-size digest (*n* bits)
- **No key** and **fixed function**
- **Examples:** SHA-256, SHA-512, SHA-3



The SHA-256 Cryptographic Hash

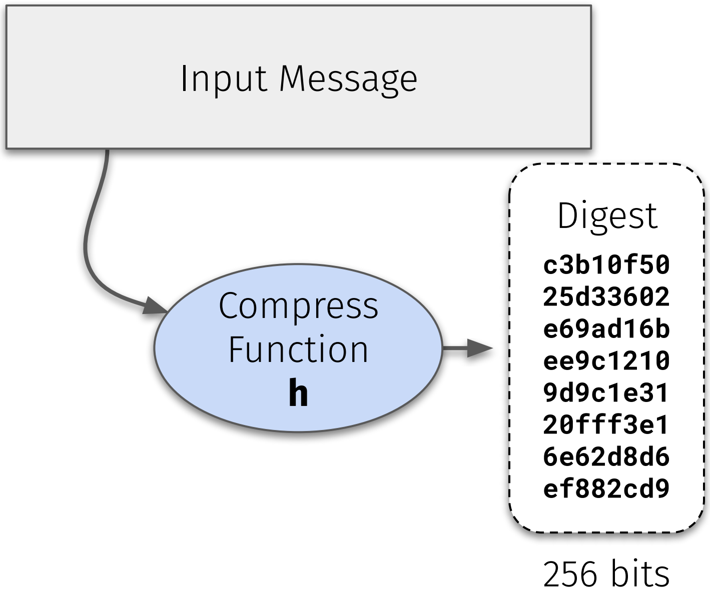
- **Input:** arbitrary-length data
- **Output:** 256-bit hash digest



The SHA-256 Cryptographic Hash

- **Input:** arbitrary-length data
- **Output:** 256-bit hash digest

- Internal compression function ***h***

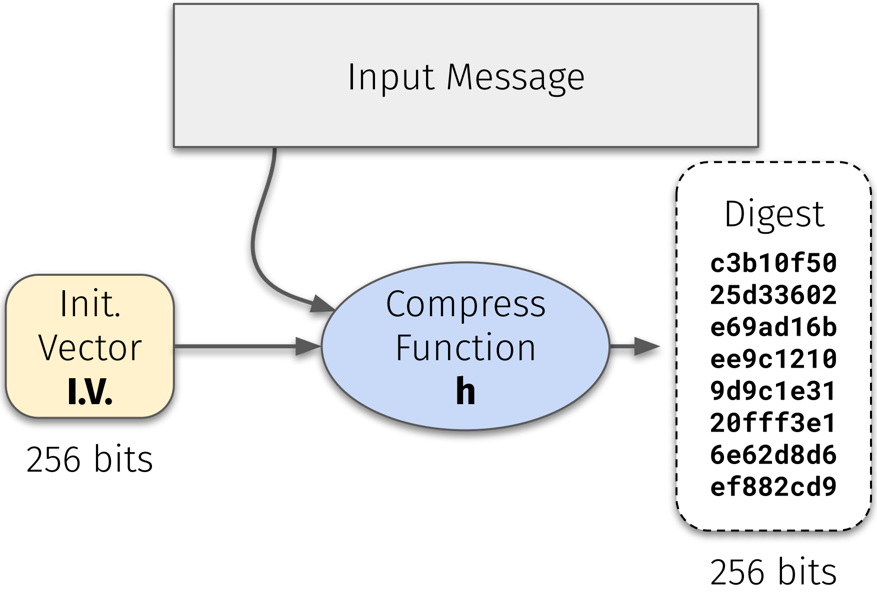


The SHA-256 Cryptographic Hash

- **Input:** arbitrary-length data
- **Output:** 256-bit hash digest

- Internal compression function ***h***

- **Inputs to *h*:**
  - **256-bit** initialization vector
    - Public—known to Mallory!

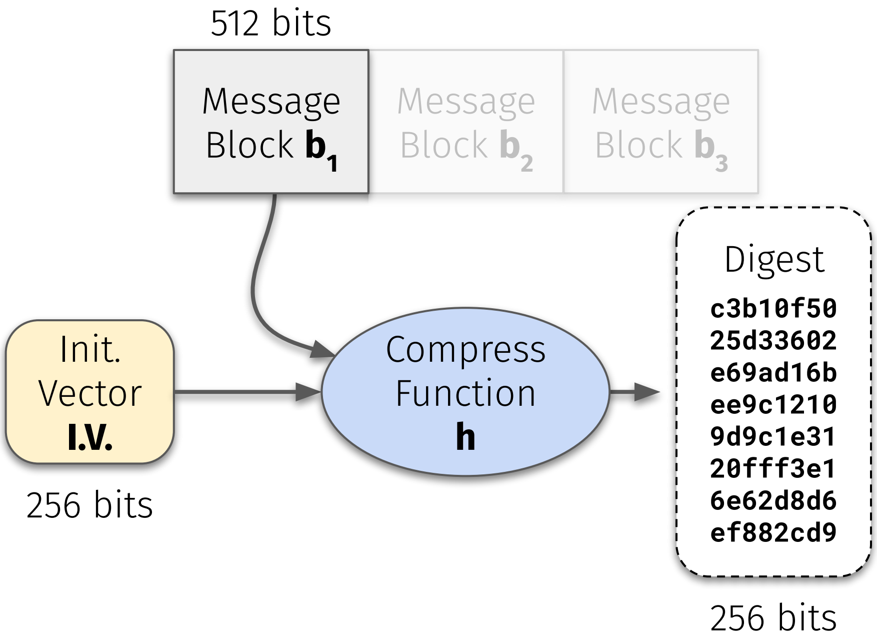


The SHA-256 Cryptographic Hash

- **Input:** arbitrary-length data
- **Output:** 256-bit hash digest

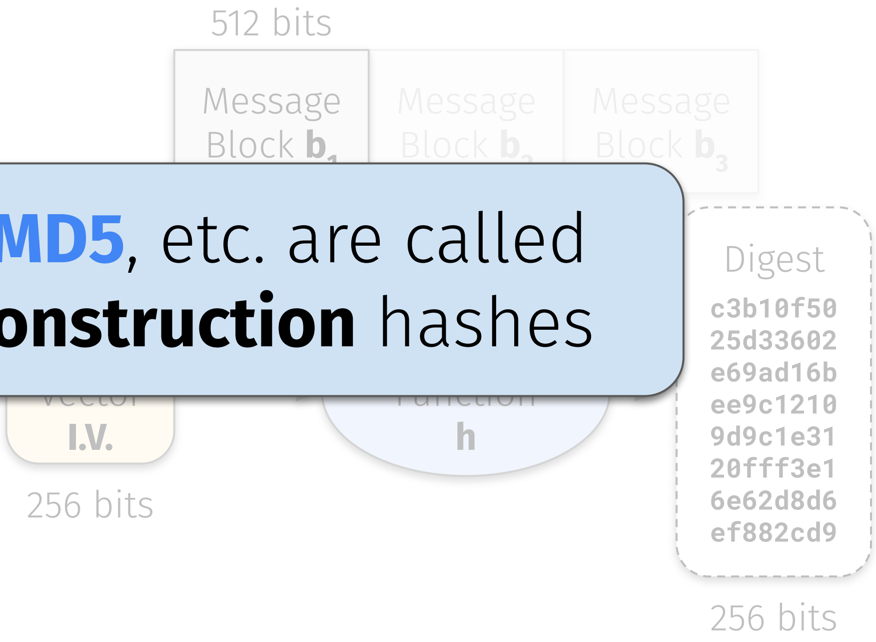
- Internal compression function ***h***

- **Inputs to *h*:**
  - **256-bit** initialization vector
    - Public—known to Mallory!
  - **512-bit** input message block
    - Input split up into 512-bit blocks



The SHA-256 Cryptographic Hash

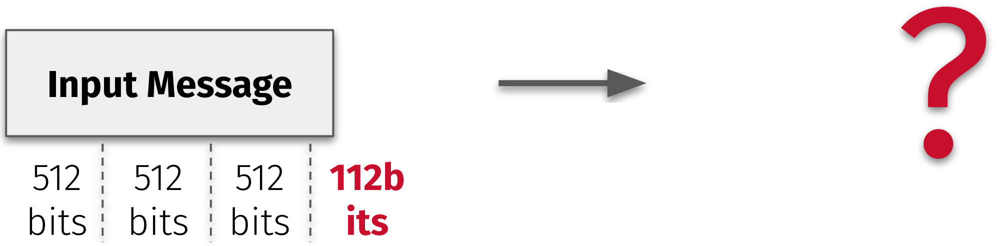
- **Input:** arbitrary-length data
- **Output:** 256-bit hash digest
- Inter
- **Input**
  - **256-bit** initialization vector
    - Public—known to Mallory!
  - **512-bit** input message block
    - Input split up into 512-bit blocks



SHA-256, **SHA-512**, **MD5**, etc. are called **Merkle–Damgård Construction** hashes

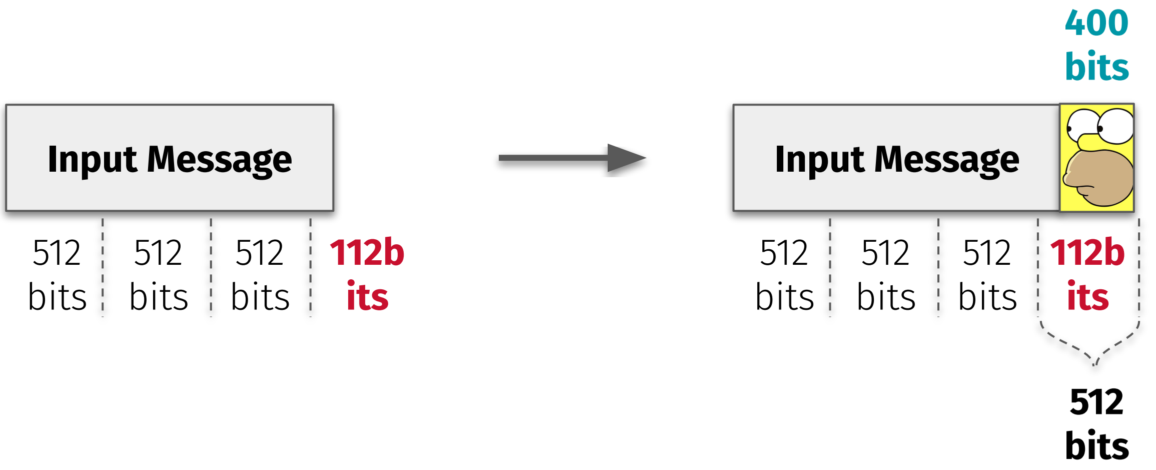
Merkle–Damgård Hash Construction

1. **Pad** input message (using a fixed, public algorithm) to a **multiple of 512 bits**



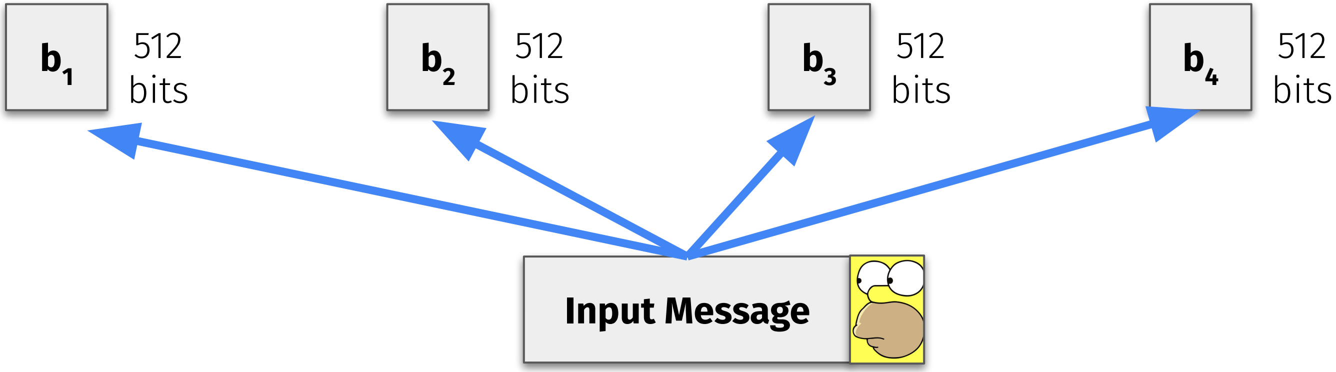
Merkle–Damgård Hash Construction

1. **Pad** input message (using a fixed, public algorithm) to a **multiple of 512 bits**



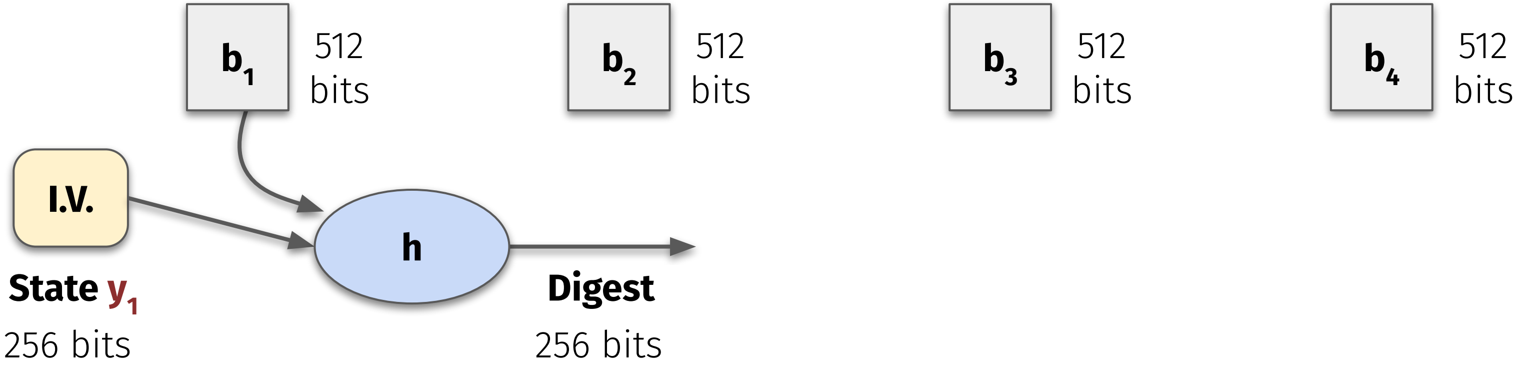
Merkle–Damgård Hash Construction

1. **Pad** input message (using a fixed, public algorithm) to a **multiple of 512 bits**
2. Split input message into **512-bit blocks: $b_1, b_2, ..., b_n$**



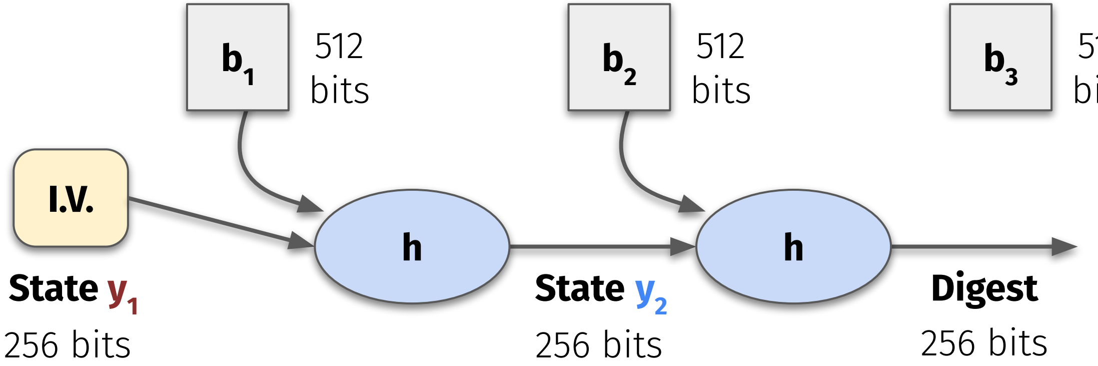
Merkle–Damgård Hash Construction

1. **Pad** input message (using a fixed, public algorithm) to a **multiple of 512 bits**
2. Split input message into **512-bit blocks:** $\mathbf{b_1, b_2, ..., b_n}$
3. Initial state $\mathbf{y_1}$ is an **Initialization Vector** (not a key!)



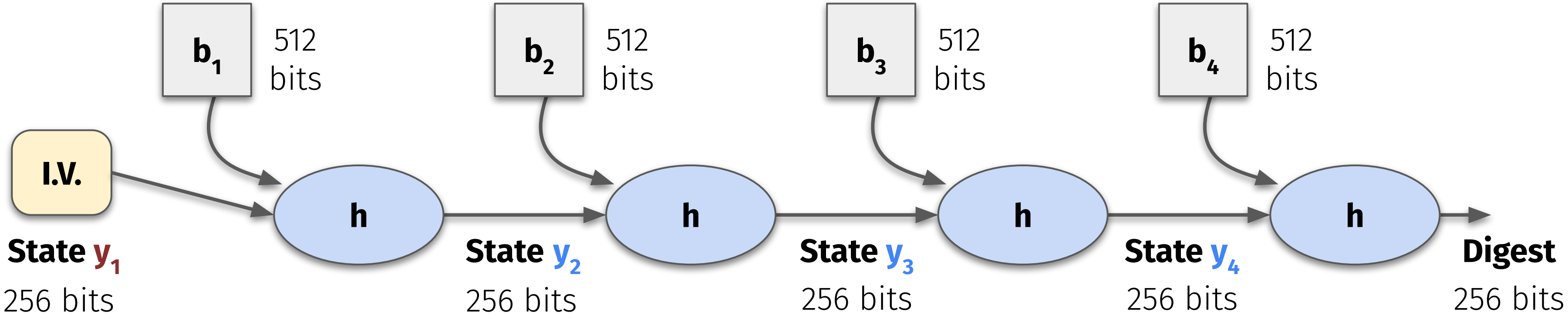
Merkle–Damgård Hash Construction

1. **Pad** input message (using a fixed, public algorithm) to a **multiple of 512 bits**
2. Split input message into **512-bit blocks:** $b_1, b_2, ..., b_n$
3. Initial state y_1 is an **Initialization Vector** (not a key!)
4. Rest of block digests calculated as: $y_n = h(y_{n-1}, b_{n-1})$



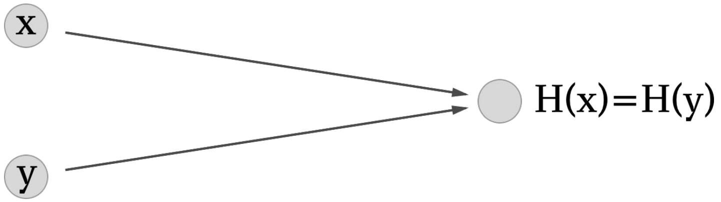
Merkle–Damgård Hash Construction

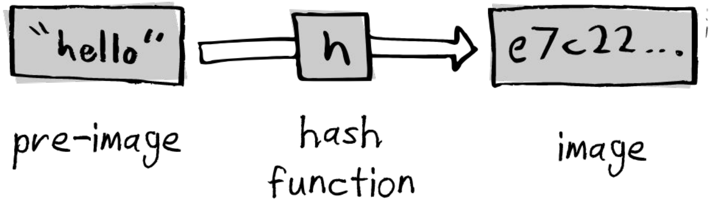
1. **Pad** input message (using a fixed, public algorithm) to a **multiple of 512 bits**
2. Split input message into **512-bit blocks:** $b_1, b_2, \ldots, b_n$
3. Initial state y_1 is an **Initialization Vector** (not a key!)
4. Rest of block digests calculated as: $y_n = h(y_{n-1}, b_{n-1})$



Properties of Cryptographic Hash Functions

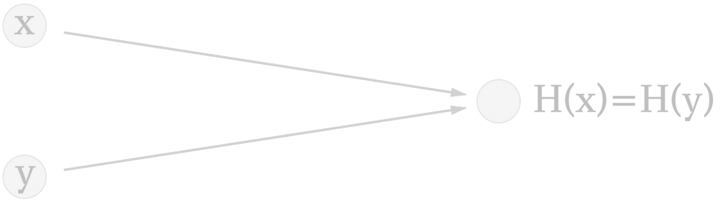
- Collision resistance:
  - Can't find any $\mathbf{m_1} \mathbf{!=} \mathbf{m_2}$ such that $\boldsymbol{h}\,(\mathbf{m_1}) = \boldsymbol{h}\,(\mathbf{m_2})$
- Second pre-image resistance:
  - Given $\mathbf{m_1}$, can't find $\mathbf{m_2} \mathbf{!=} \mathbf{m_1}$ such that $\boldsymbol{h}\,(\mathbf{m_1}) = \boldsymbol{h}\,(\mathbf{m_2})$
- Pre-image resistance:
  - Given ***h* (m)**, can't find **m**
- "Can't find" = **infeasible** to compute

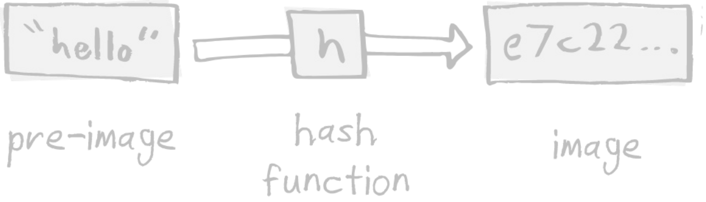




Properties of Cryptographic Hash Functions

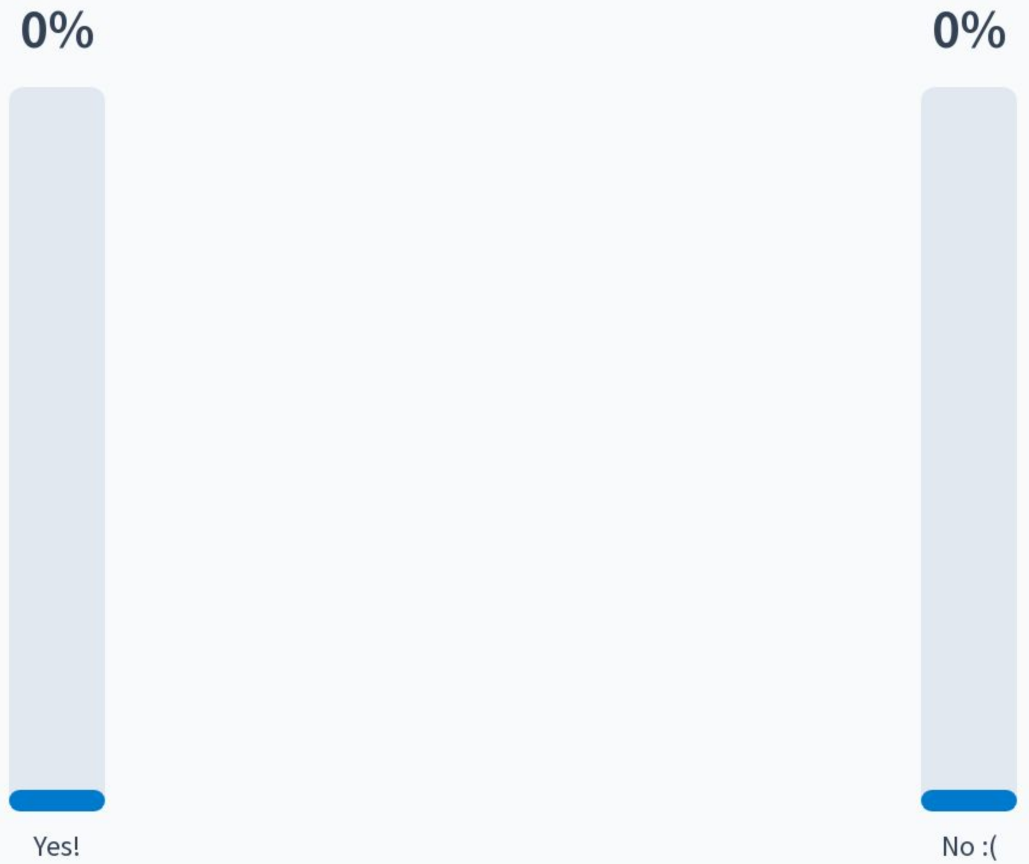
- Collision resistance:
  - Can't find any $\mathbf{m_1} \mathrel{!}= \mathbf{m_2}$ such that $h(\mathbf{m_1}) = h(\mathbf{m_2})$
- Second pre-image resistance:
  - Given $\mathbf{m_1}$, can't find $\mathbf{m_2} \mathrel{!}= \mathbf{m_1}$ such that $h(\mathbf{m_1}) = h(\mathbf{m_2})$
- Pre-image resistance:
  - Given $h(\mathbf{m})$, can't find $\mathbf{m}$
- "Can't find" = **infeasible** to compute





Are "secure" hashes secure **forever**?

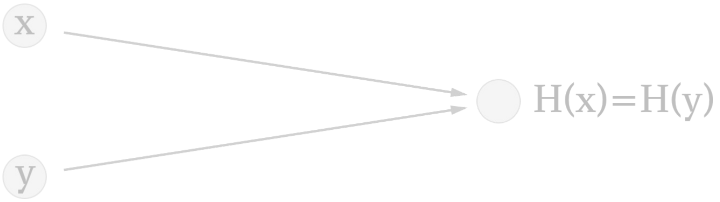
Are "secure" hash functions secure forever?



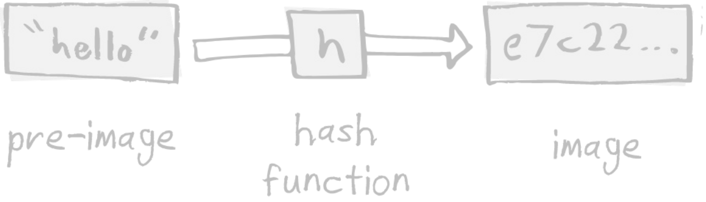
Start the presentation to see live content. For screen share software, share the entire screen. Get help at **pollev.com/app**

Properties of Cryptographic Hash Functions

- Collision resistance:
  - Can't find any $\mathbf{m_1} \mathrel{!}= \mathbf{m_2}$ such that $h(\mathbf{m_1}) = h(\mathbf{m_2})$



- Second pre-image resistance:
  - Given $\mathbf{m_1}$, can't find $\mathbf{m_2} \mathrel{!}= \mathbf{m_1}$ such that $h(\mathbf{m_1}) = h(\mathbf{m_2})$



- Pre-image resistance:
  - Given $h(\mathbf{m})$, can't find $\mathbf{m}$
- "Can't find" = **infeasible** t

Are "secure" hashes secure **forever**? **No!**

Questions?

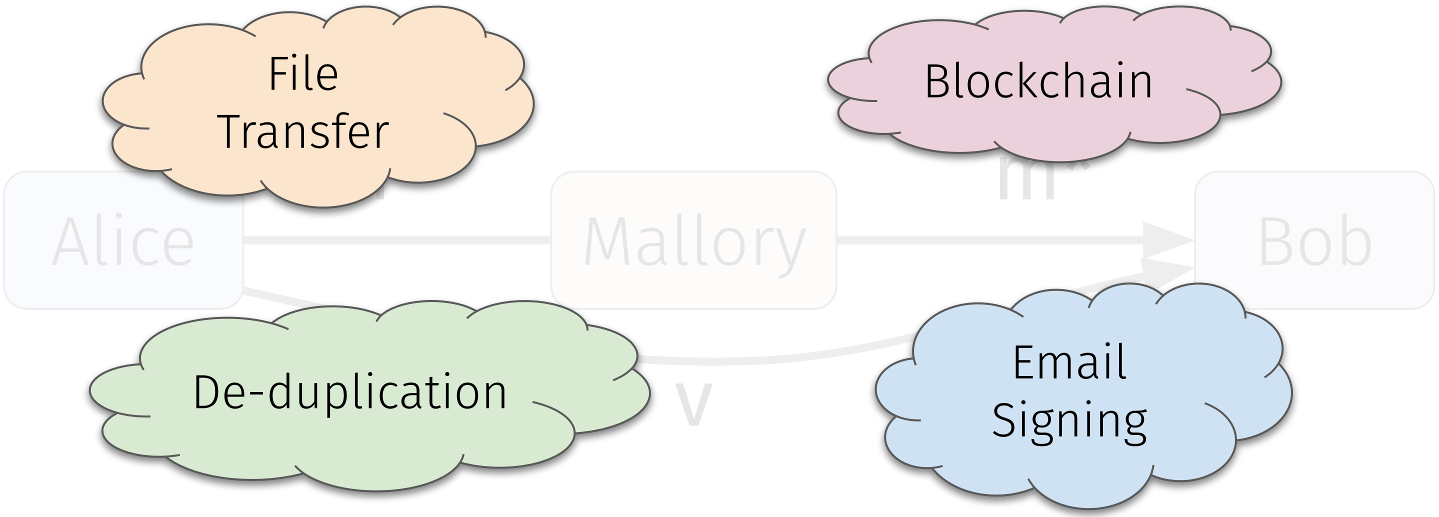
Attacks on Hash Functions

What are some everyday uses of hashes?

- What are some everyday uses of hashes?

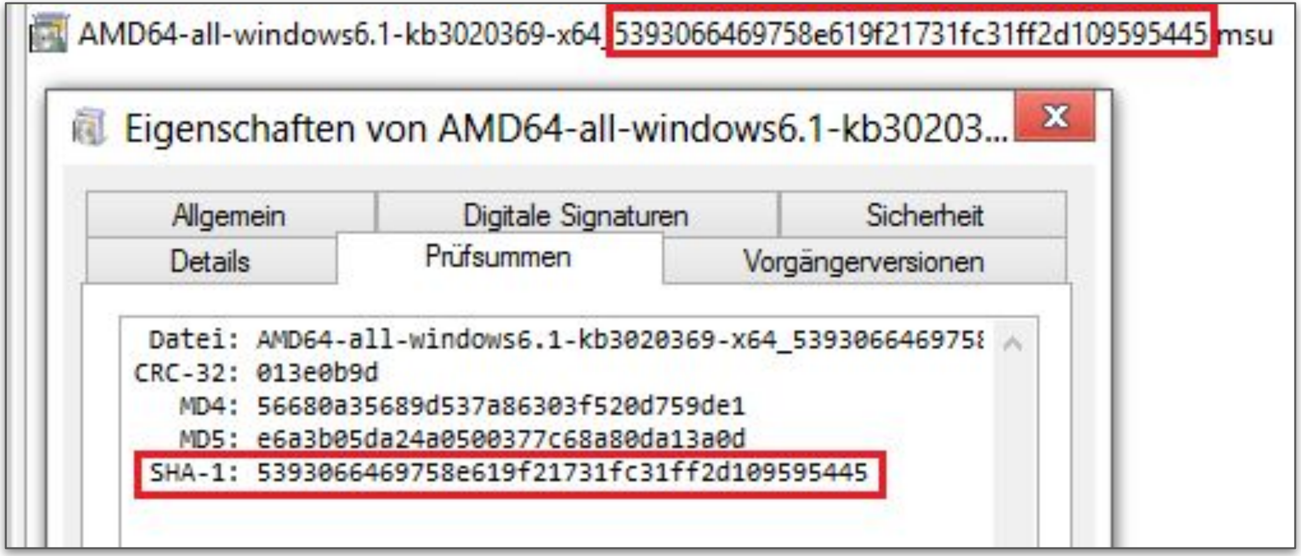
What are some everyday uses of hashes?

- What are some everyday uses of hashes?



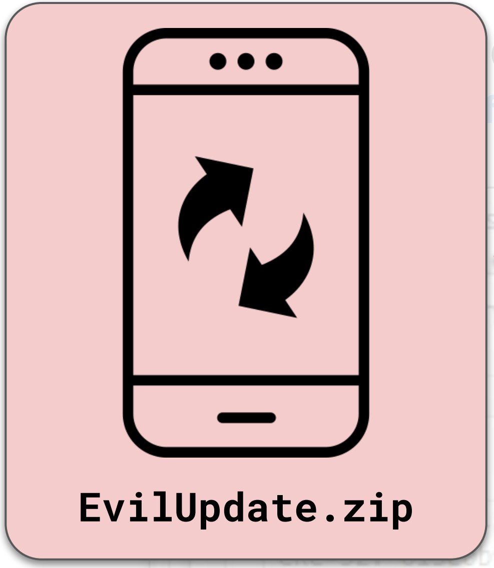
Problem: Collision Attacks

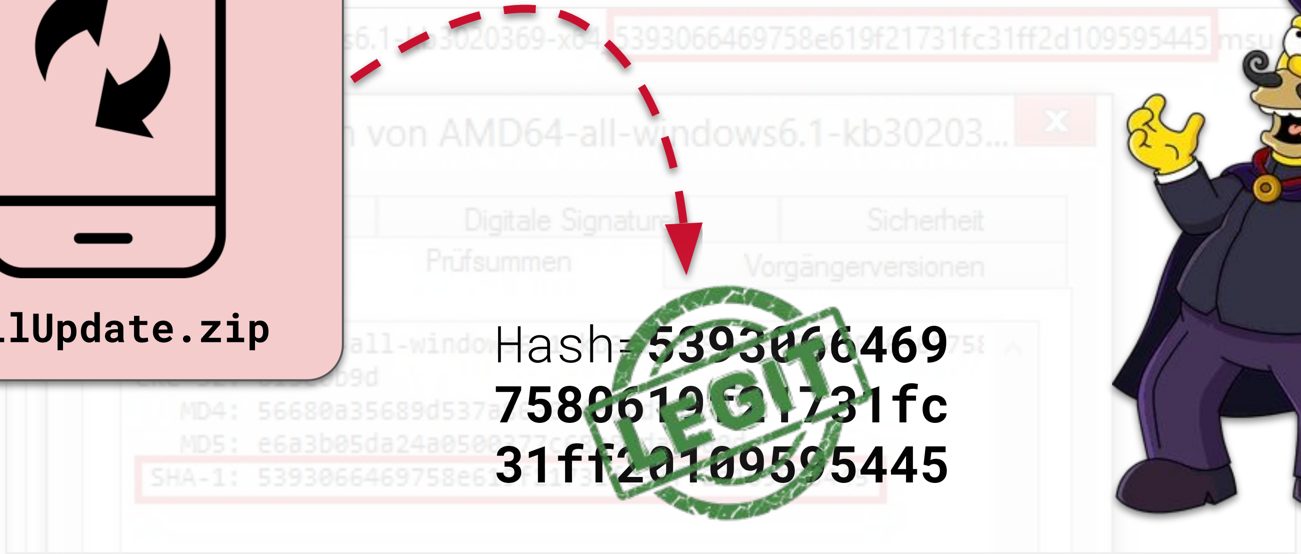
- Suppose the Crabapple yPhone prompts you to install a **software update**…
  - How do you know **the file you downloaded** is the file **Crabapple wanted you to download**?



Problem: Collision Attacks

- ...e yPhone prompts you to install a **software update...**
  - ...file you downloaded is the file Crabapple wanted you to download?





Problem: Collision Attacks

Flame's MD5 collision is the most worrisome security discovery of 2012

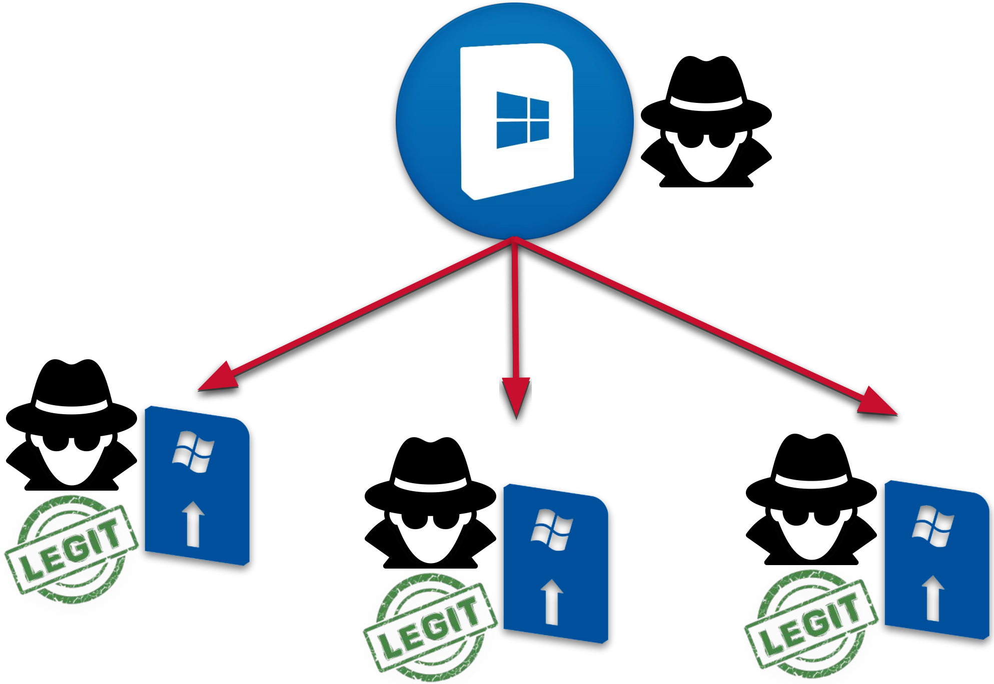
Richard Stiennon Former Contributor

Industry analyst. Author.

Jun 14, 2012, 06:45am EDT

This article is more than 10 years old.

In 2009, while I was researching *Surviving Cyberwar*, I attended the COSAC security conference outside of Dublin for the first time. During an open session I posed this question to the attendees: “Can you think of any cyber weapons we may see in the near future?” There were few responses during the open session but that evening at dinner one of the attendees leaned towards me and said “I have one for you, Microsoft update.” What he was implying was that if an attacker could get between Microsoft's massive update service and an intended target any machine could be compromised.



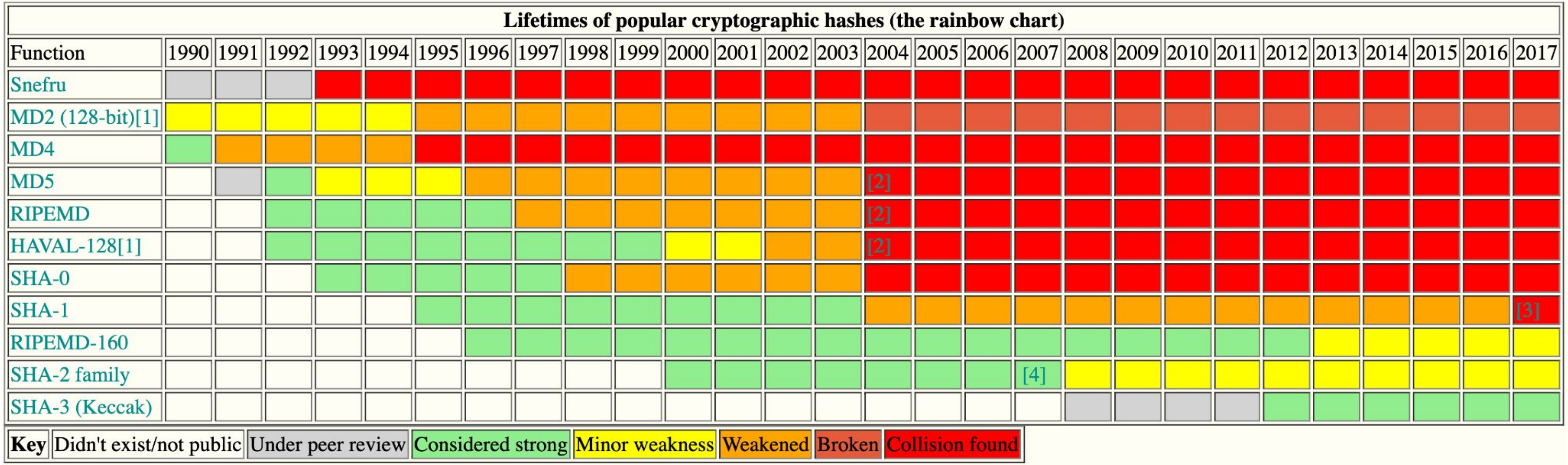
Defeated Hash Functions

- **MD5**
  - Once ubiquitous
  - Broken in 2004
  - Now easy to find collisions
    - You will in Project 1 😁
  - Exploited to attack real systems

- **SHA-1**
  - All major web browser vendors ceased acceptance of SHA-1 SSL certificates in 2017
  - February 2017: CWI Amsterdam and Google announced a collision attack against SHA-1
    - Created two dissimilar PDF files with same SHA-1 hash
  - April 2019: Leurent and Peyrin created an attack capable of finding chosen-prefix collisions in approximately 268 SHA-1 evaluations, requiring only $100,000 of cloud processing

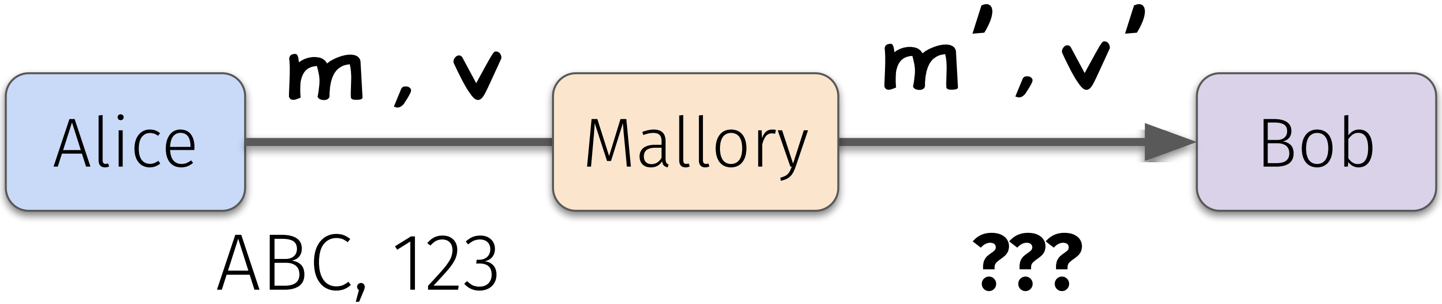
Defeated Hash Functions

- Hashes proven to be **insecure**—do not use cryptographically!
  - valerieaurora.org/hash.html



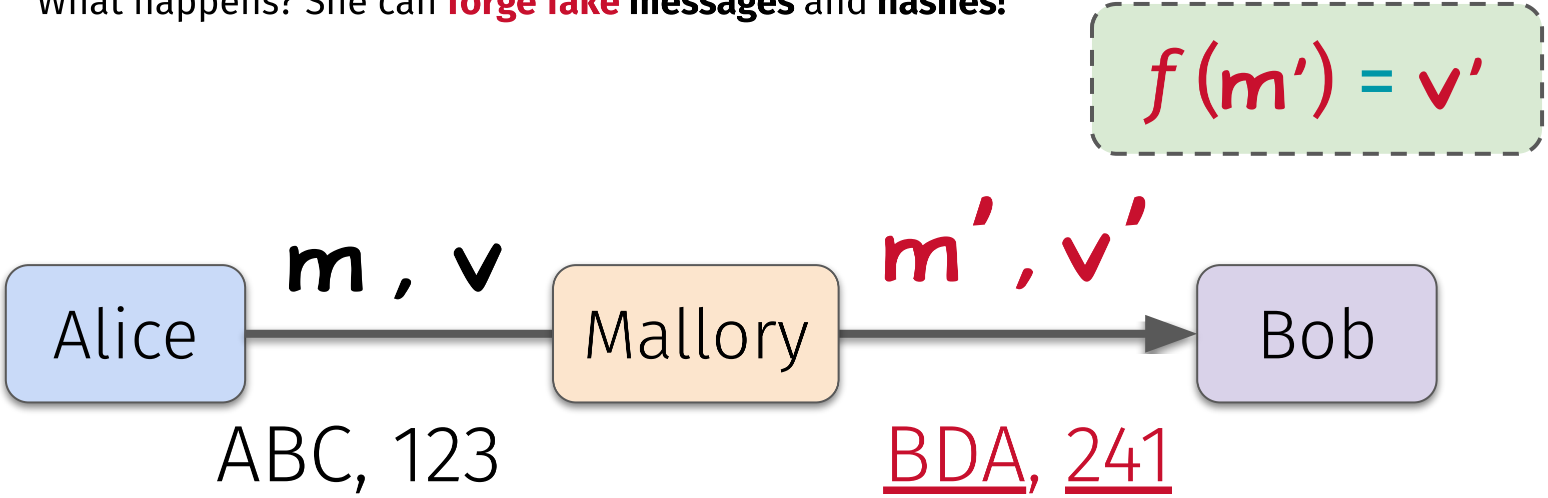
Recap: Mallory-known Function

- We talked about the case where Mallory **knows the internals** of function ***f***
  - What happens?



Recap: Mallory-known Function

- We talked about the case where Mallory **knows the internals** of function ***f***
  - What happens? She can **forge fake messages** and **hashes!**



Recap: Mallory-known Function

- We talked about the case where Mallory **knows the internals** of function f
  - What happens? She can **forge fake messages** and **hashes!**

$f(m') = y'$

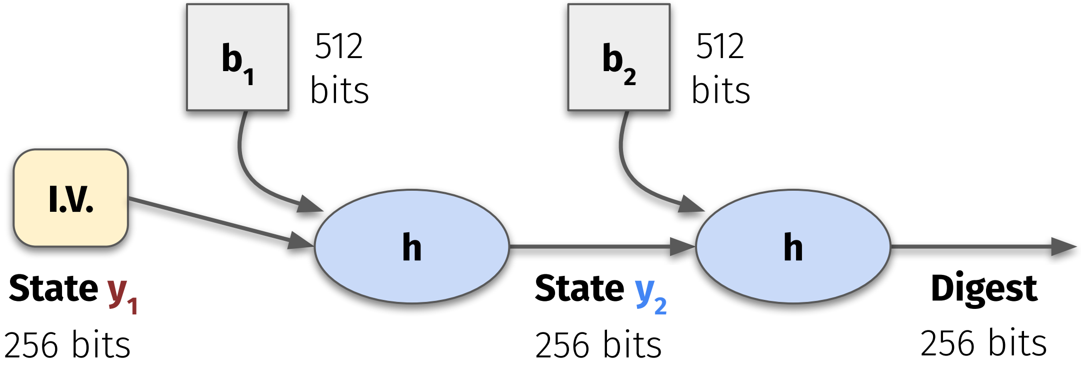
ABC, 123

BDA, 241

If our function is a **Merkle–Damgård Hash**, what **control could Mallory** have over the **final digest**?

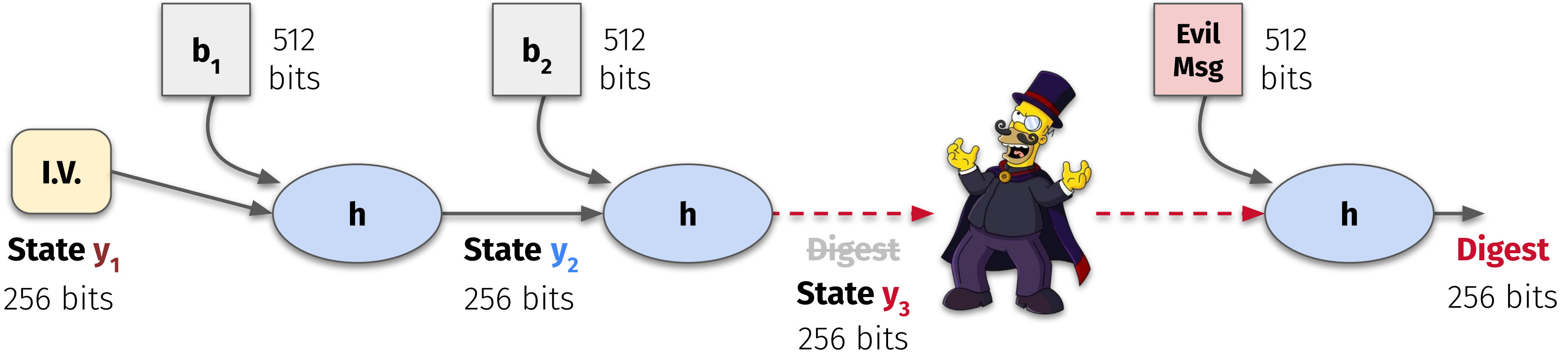
Problem: Length Extension Attacks

- Merkle–Damgård construction: digest is formed from **the last chaining value**



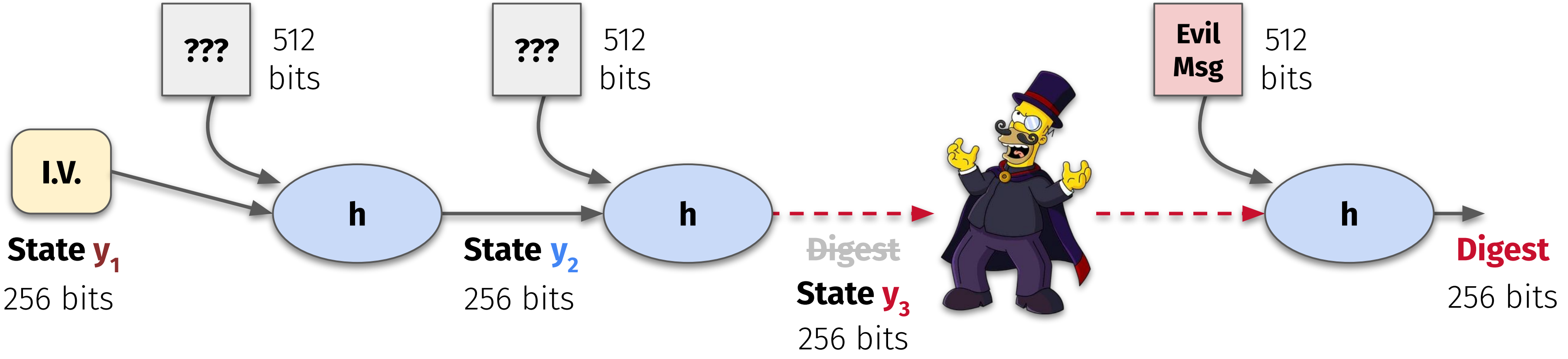
Problem: Length Extension Attacks

- Merkle–Damgård construction: digest is formed from **the last chaining value**
- Nothing stopping Mallory from **continuing** the hash chain…



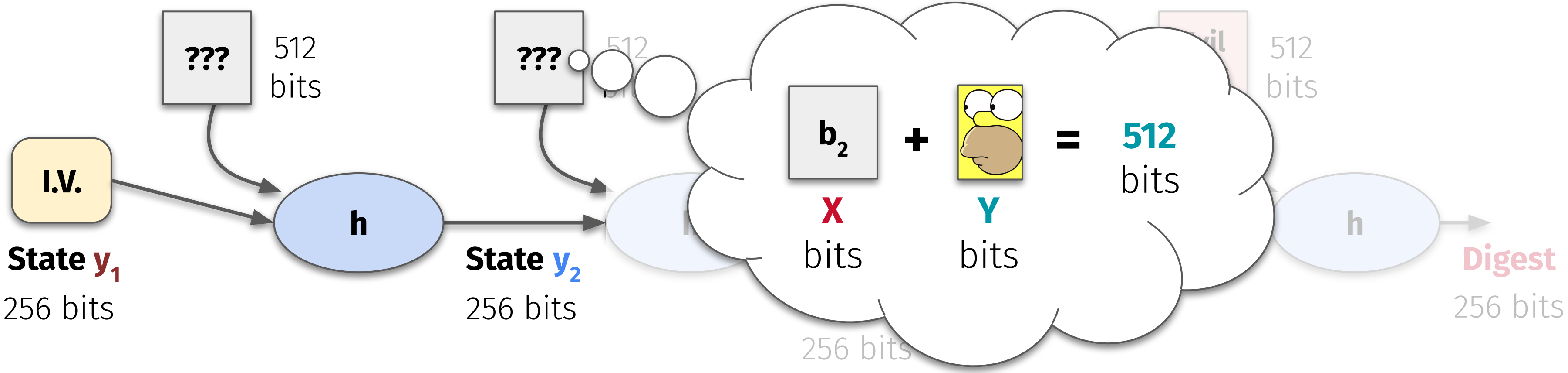
Problem: Length Extension Attacks

- Merkle–Damgård construction: digest is formed from **the last chaining value**
- Nothing stopping Mallory from **continuing** the hash chain…
  - Mallory **doesn't need** to know the **previous blocks' plaintext**



Problem: Length Extension Attacks

- Merkle–Damgård construction: digest is formed from **the last chaining value**
- Nothing stopping Mallory from **continuing** the hash chain…
  - Mallory **doesn't need** to know the **previous blocks' plaintext**
  - But she does know that the **last block was padded** to 512 bits



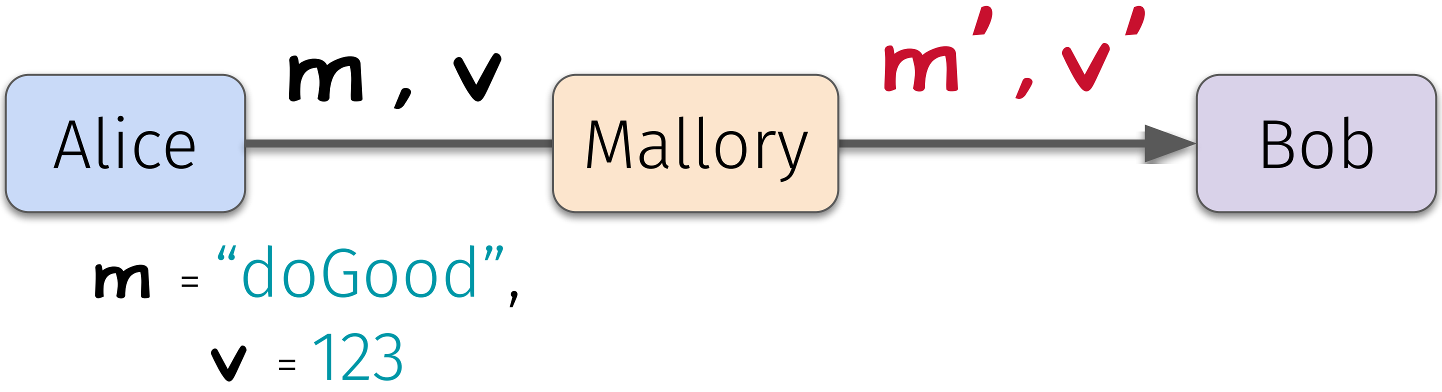
Problem: Length Extension Attacks

- What if Mallory figures out the **length** of the input message?

Problem: Length Extension Attacks

- What if Mallory figures out the **length** of the input message?
  - She can then calculate the **final block's padding**!

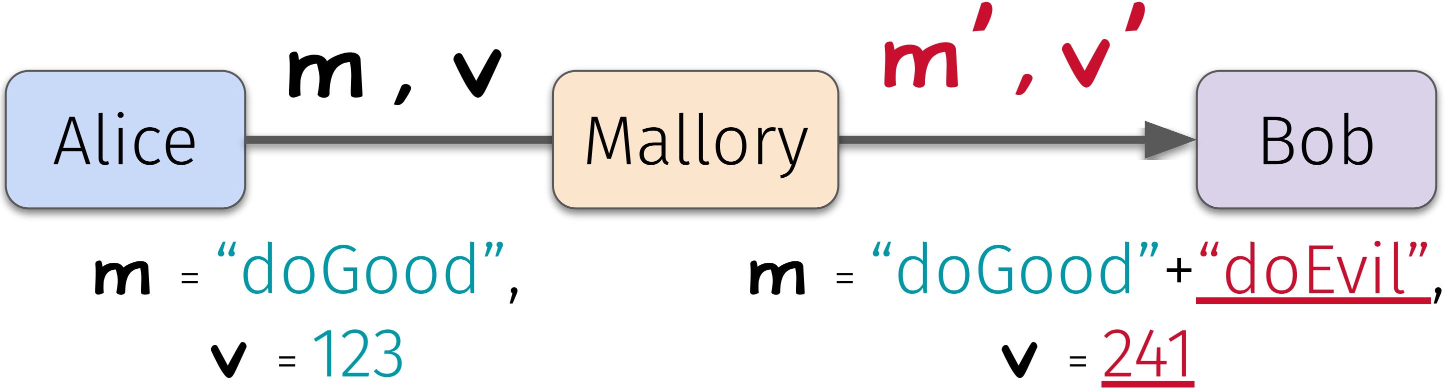
- Suppose our system validates users' command strings via their hashes…



Problem: Length Extension Attacks

- What if Mallory figures out the **length** of the input message?
  - She can then calculate the **final block's padding**!

- Suppose our system validates users' command strings via their hashes…
  - Mallory can **inject her own commands**—just by knowing the original message **length**!



Problem: Length Extension Attacks

- What if Mallory figures out the **length** of the input message?
  - She can then calculate the **final block's padding**!
- Suppose our system vali… strings via their hashes…
  - Mallory can **inject her ow**… the original message **length**!

Final outcome:

$f(m') = v'$

m , … , v'

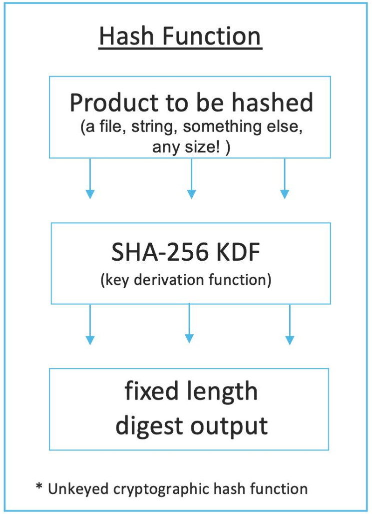
Alice → Mallory → Bob

m = "doGood",
v = 123

m = "doGood"+"doEvil",
v = 241

Solution: Use a MAC Instead

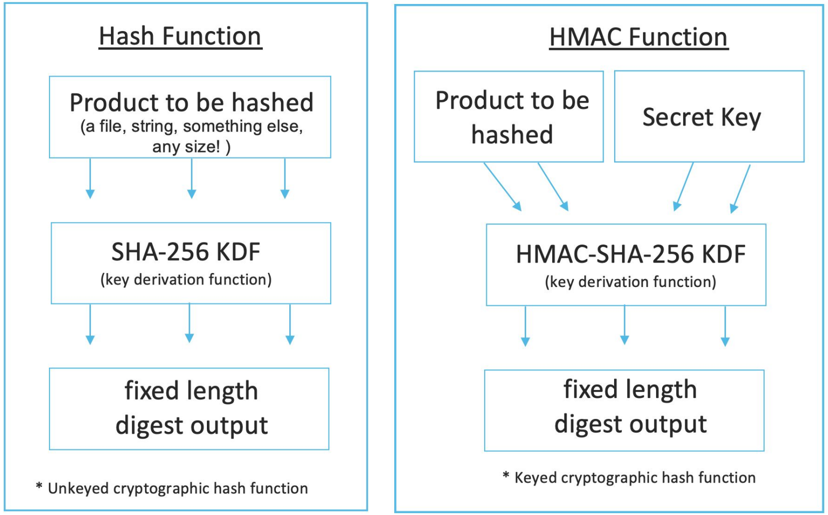
- Cryptographic Hash Function
  - e.g., SHA256
  - **Not a strong PRF**
    - Length-extension attacks



Solution: Use a MAC Instead

- Cryptographic Hash Function
  - e.g., SHA256
  - **Not a strong PRF**
    - Length-extension attacks

- Message Authentication Code (MAC)
  - Think of as **synonymous with PRF**
    - Widely believed to be PRFs
  - e.g., HMAC-SHA256
    - HMAC = **keyed-hash MAC**
    - Currently recommended



The HMAC-SHA256 Function

- $HMAC_k\ (m)$ =

$$\text{SHA256}\ (\ (\mathbf{k} \oplus \mathbf{pad}_{outer})\ ||\ \text{SHA256}\ ((\mathbf{k} \oplus \mathbf{pad}_{inner})\ ||\ \mathbf{m})\ )$$

- Here, **k** = secret key

The HMAC-SHA256 Function

- $HMAC_k(m) =$

XOR

$$SHA256\,(\,(\mathbf{k} \oplus \mathbf{pad}_{outer}) \,||\, SHA256\,((\mathbf{k} \oplus \mathbf{pad}_{inner}) \,||\, \mathbf{m})\,)$$

concatenate

- Here, **k** = secret key

The HMAC-SHA256 Function

- $HMAC_k$ (m) =

$$SHA256\ (\ (\mathbf{k} \oplus \mathbf{pad}_{outer})\ ||\ SHA256\ ((\mathbf{k} \oplus \mathbf{pad}_{inner})\ ||\ \mathbf{m})\ )$$

0x5c5c5c5c… (outer) **0x36363636…** (inner)

- Here, **k** = secret key; **padding** = 0x5c and 0x36 repeated **64 times**

The HMAC-SHA256 Function

- $HMAC_k(m)$ =

$$SHA256\left(\left(\mathbf{k} \oplus \mathbf{pad}_{outer}\right) \,||\, SHA256\left(\left(\mathbf{k} \oplus \mathbf{pad}_{inner}\right) \,||\, \mathbf{m}\right) \right)$$

0x5c5c5c5c… (pad outer)

0x36363636… (pad inner)

- Here, **k** = secret key; **padding** = 0x5c and 0x36 repeated **64 times**
- **Nested** construction rather than chained like Merkle–Damgård
  - Goodbye length extension and forgery!

Questions?

Project Tips

Project Tips

- Projects are challenging—**you're performing real-world attacks!**
  - Build off of lecture concepts
  - Make sure you understand the lectures
  - Prepare you to defend **in the real world**

Project Tips

- Projects are challenging—**you're performing real-world attacks!**
  - Build off of lecture concepts
  - Make sure you understand the lectures
  - Prepare you to defend **in the real world**

- **Suggested strategy: get high-level idea down, then start implementing**
  1. Go through assignment and start sketching-out your approach
  2. **Come to Office Hours and ask if you're on the right track!**
  3. Then start building your program

Project Tips

- Projects are challenging—**you're performing real-world attacks!**
  - Build off of lecture concepts
  - Make sure you understand the lectures
  - Prepare you to defend **in the real world**

- **Suggested strategy: get high-level idea down, then start implementing**
  1. Go through assignment and start sketching-out your approach
  2. **Come to Office Hours and ask if you're on the right track!**
  3. Then start building your program

- Don't get discouraged—**we are here to help!**
  - Most issues are cleared up in a few minutes of white-boarding

Next time on CS 4440...

Confidentiality, Substitution Ciphers