Fundamentals of Cryptographic Hashing

Ms. N. Thanu Priya, Dr. V Sheeja Kumari

ISBN: 978-81-984733-7-0

Publisher: International Institute of Organized Research (I2OR)

Fundamentals of Cryptographic Hashing

Author(s): Ms. N. Thanu Priya, Dr. V Sheeja Kumari

Vol. 1 February 2026

ISBN: 978-81-984733-7-0

Published By:

Number 3179, Sector 52, Chandigarh (160036) - India

Printed by
Green ThinkerZ
#530, B-4, Western Towers, Sector 126, Greater Mohali, Punjab (140301) India

Author 1

Acknowledgement

First and foremost, I bow my head in gratitude to my beloved father Mr. M.Nagarupillai, mother Mrs. K. Chithambara Vadivoo for their enduring support, prayers, and unconditional love.

My loving thanks to my niece S.Achira and nephew S.Mahish and to my entire family for their love, understanding and patience.

I express my heartfelt gratitude to my Ph.D. guide HoD. Dr. V. Sheeja Kumari for her expert guidance, encouragement, blessings and invaluable mentorship, because without her any of these work wouldn't have been possible.

With folded hands and a heart full of gratitude, I bow before all the Acharyas & God, for their endless grace, mercy, strength and for being my saviours always.

Ms. Thanu Priya .N

Author 2

Acknowledgement
I want to thank:

My daddy Mr. N.Vaikunda Mani, mummy Mrs. M.Vijaya Kumari, and my kid Johanna P Sheeju for their soulful support and their encouragement throughout my career.

I want to thank EVERYONE who ever said anything positive to me or taught me something. I heard it all, and it meant something.

All the dudes I ever slept with, I appreciate the experiences, but I ain't naming none of you!

I want to thank God most of all, because without God I wouldn't be able to do any of this.

Dr Cja.

Author 1

Ms. N Thanu Priya received the B.Sc. degree in Computer Science from Sri Chandrasekarendra Saraswati Vishwa Maha Vidyalaya (Kancheepuram University), M.Sc degree in Computer Science from Justice Basheer Ahmed Sayeed College for Women (Madras University), Chennai, M.Tech. Degree in Information Technology from Sathyabama University, Chennai and currently a research scholar in Saveetha School of Education, Chennai. She worked as an Assistant Professor in Sri Krishna Engineering College, Padappai, and is currently working as a Teaching Assistant in Sri Sankara Vidyalaya Mat. Hr. Secondary School, Chennai. She has an experience of 14 years in the field of teaching.

Author 2

Dr V Sheeja Kumari, an Indian citizen currently working as a Professor / Department of Computational Intelligence – Institute of AI & ML, SIMATS School of Engineering, SIMATS University in Chennai. Life Member of I2OR, a member of CSTA, a member of IAAC, and a life member of IAENG. Received IRSD International Preeminent Academic Leader Award 2022 from International Institute of Organized Research - I2OR (A Registered MSME with Ministry of MSME, Government of India and Green ThinkerZ. Published books entitled “Internet of things industry 4.0”, “Software Engineering” and “CIA of Cybersecurity” . Published more number of papers in Sci journals, Scopus indexed journals, book chapters and international conferences. Acting as a chief editor for the edited book series “ Advances in Computer Technology and Applications”, Scripown Publications.

TABLE OF CONTENTS

Chapter 1

1.Introduction to Cryptographic Hash Functions

Cryptographic Hash Functions are mathematical algorithms that take any amount of input data (message, file, or password) and convert it into a fixed-length output called a *hash value* (or digest). They are designed to be **one-way**(cannot be reversed), **fast to compute**, and **highly sensitive to input changes**—even a tiny change in input produces a completely different hash. These functions are mainly used for **data integrity checking, password storage, and digital signatures**, ensuring that information has not been altered.

1.1 Understanding Hash Functions

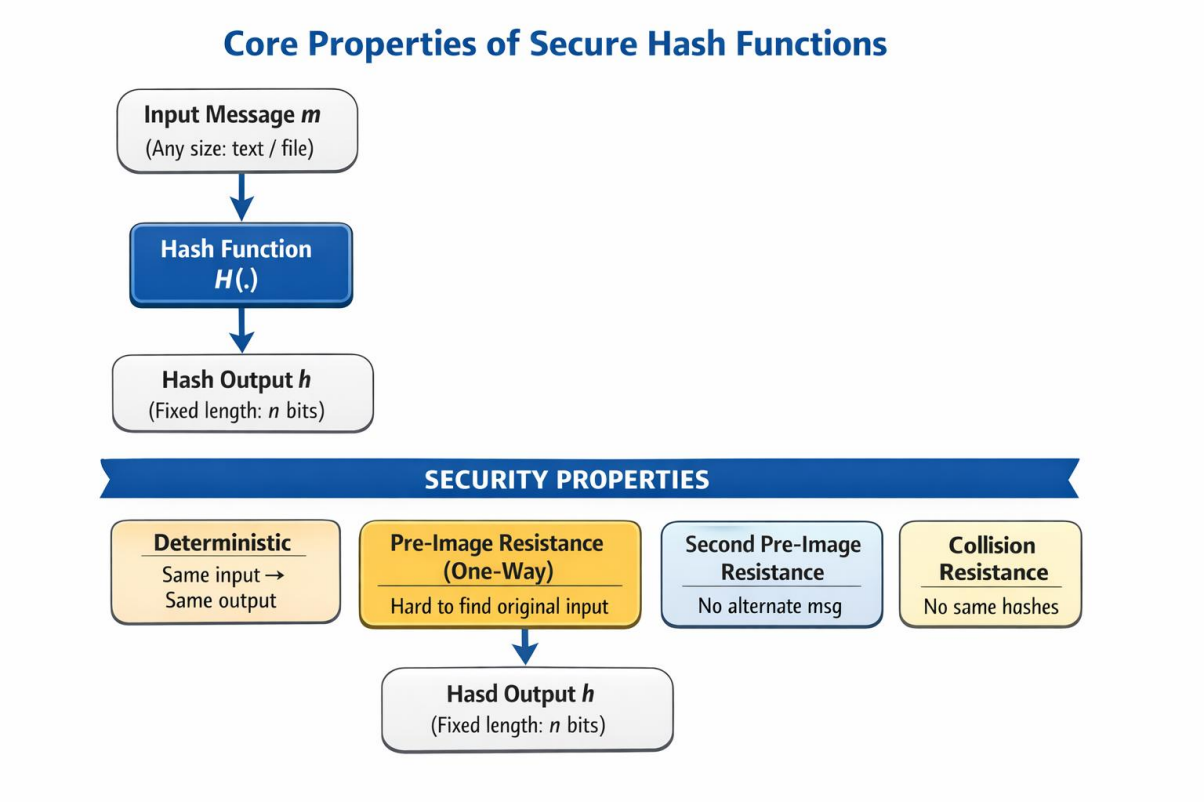
A cryptographic hash function is a mathematical algorithm that transforms an input of any size into a fixed-size output. This output, typically represented as a hexadecimal number, is called the hash value, digest, or simply hash. The fundamental concept is analogous to a digital fingerprint—every piece of data, regardless of its size, produces a unique identifier of consistent length.

Consider the analogy of a blender: you can put in a single strawberry or an entire fruit basket, and you always get a smoothie of the same consistency. However, unlike a blender, hash functions are one-way streets—you cannot reverse the process to retrieve the original ingredients from the smoothie.

Real-World Example: When you download a large software file (like Ubuntu Linux, which is several gigabytes), the website provides an MD5 or SHA-256 checksum. After downloading, you compute the hash of your downloaded file. If it matches the published hash, you know your download is complete and uncorrupted—even though you never compared the actual files byte-by-byte.

1.2 Core Properties of Secure Hash Functions

For a hash function to be useful in security contexts, it must satisfy several critical properties that make it suitable for cryptographic applications:

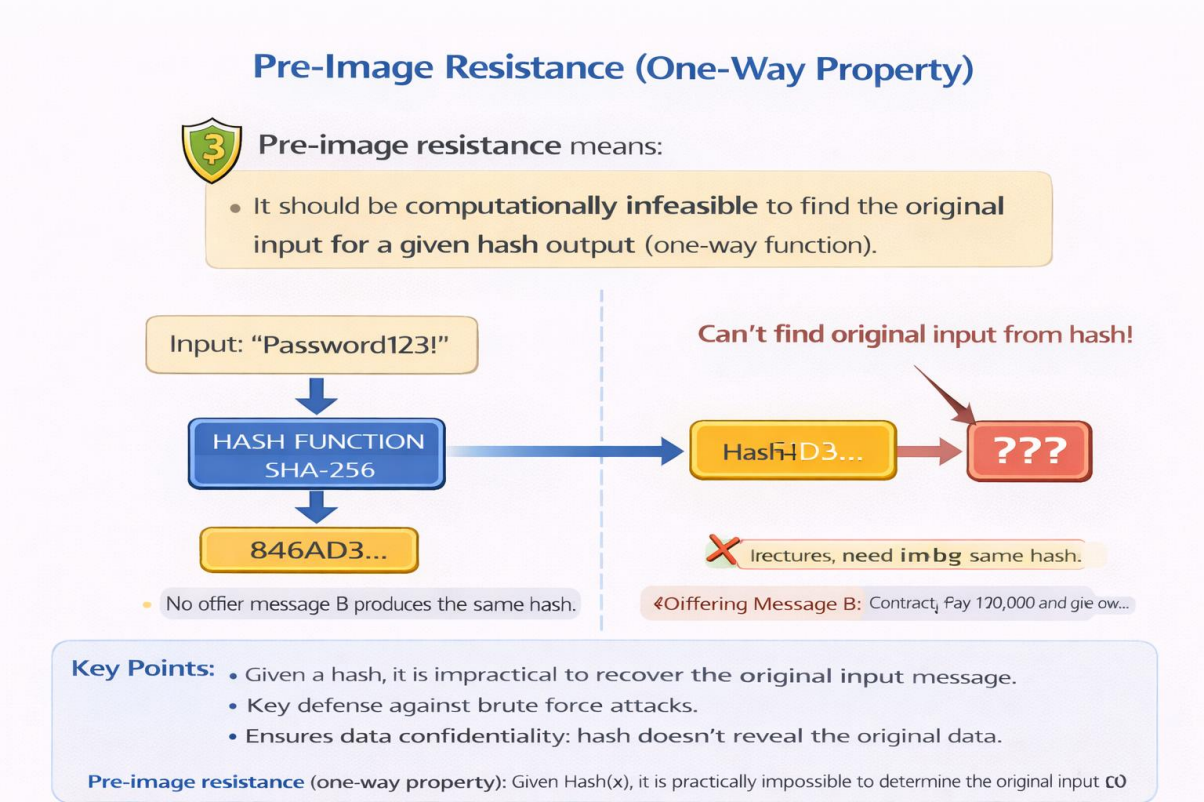


Deterministic Nature

The same input must always produce exactly the same hash output. This property seems obvious but is fundamental—if the same message produced different hashes each time it was processed, the hash function would be useless for verification purposes.

Pre-Image Resistance (One-Way Property)

Given a hash output h, it should be computationally infeasible to find any input message m such that hash(m) = h. This property is what makes hash functions "one-way." An attacker who obtains a hash value should not be able to work backwards to discover the original data. This is crucial for applications like password storage, where the system stores only the hash, not the actual password.

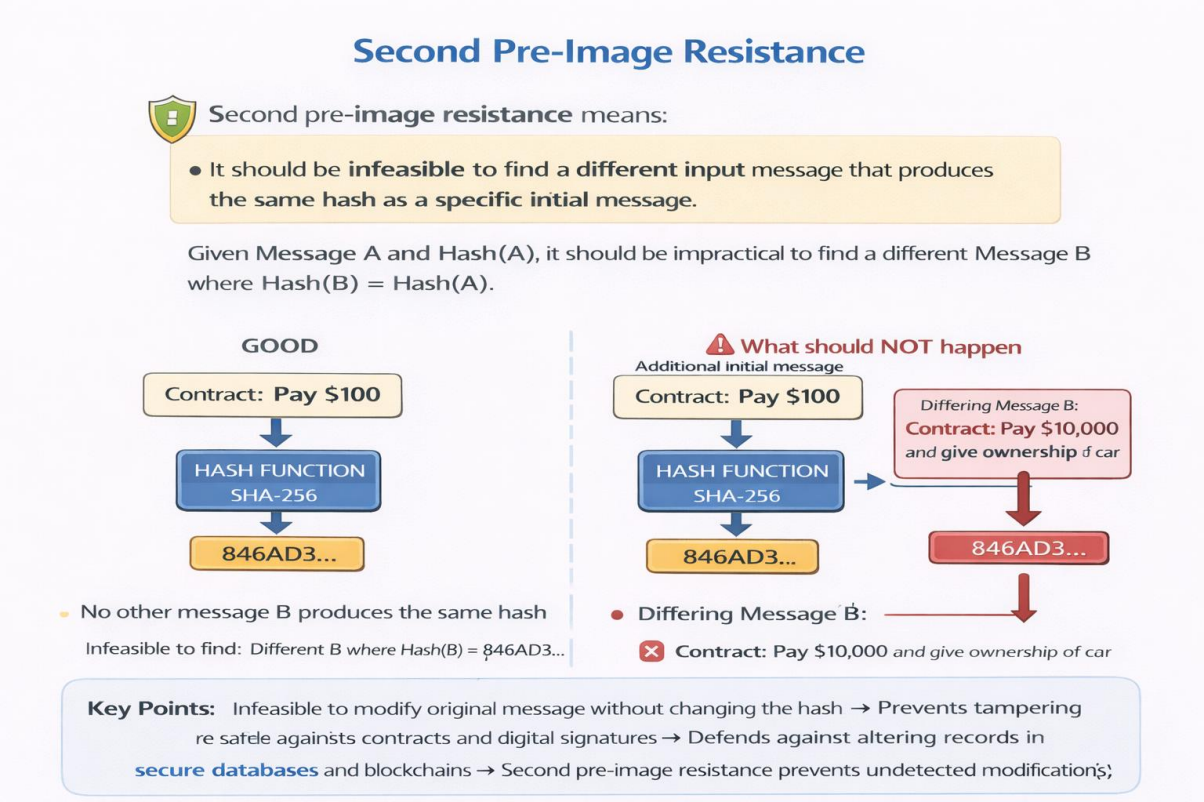


Mathematical Formulation:

For a hash function H and a randomly chosen output y, finding any x such that H(x) = y should require approximately 2^n operations, where n is the output length in bits.

Second Pre-Image Resistance

Given an input message m_1, it should be computationally infeasible to find a different input message m_2 such that hash(m_1) = hash(m_2). This property ensures that if someone has a specific message, they cannot find another message that produces the same hash. This is important for digital signatures—if an attacker has a signed message, they shouldn't be able to create a different message that produces the same signature.

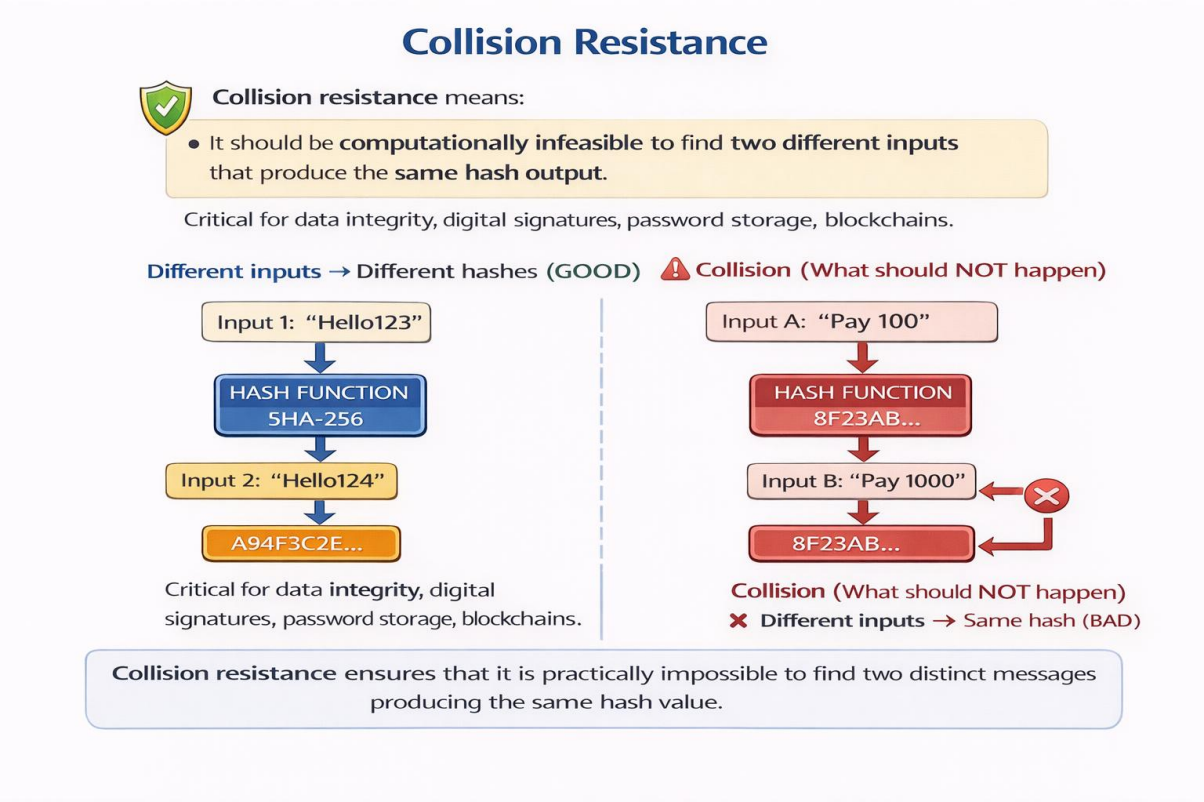


Mathematical Formulation:

Given x, finding $x' \neq x$ such that $H(x') = H(x)$ should require approximately 2^n operations.

Collision Resistance

It should be computationally infeasible to find any two distinct input messages m1 and m2 such that hash(m1) = hash(m2). This is stronger than second pre-image resistance because the attacker can choose both messages arbitrarily. Collision resistance is essential for hash functions used in digital signatures—if an attacker can find two messages with the same hash, they could trick someone into signing one message while intending to claim they signed the other.

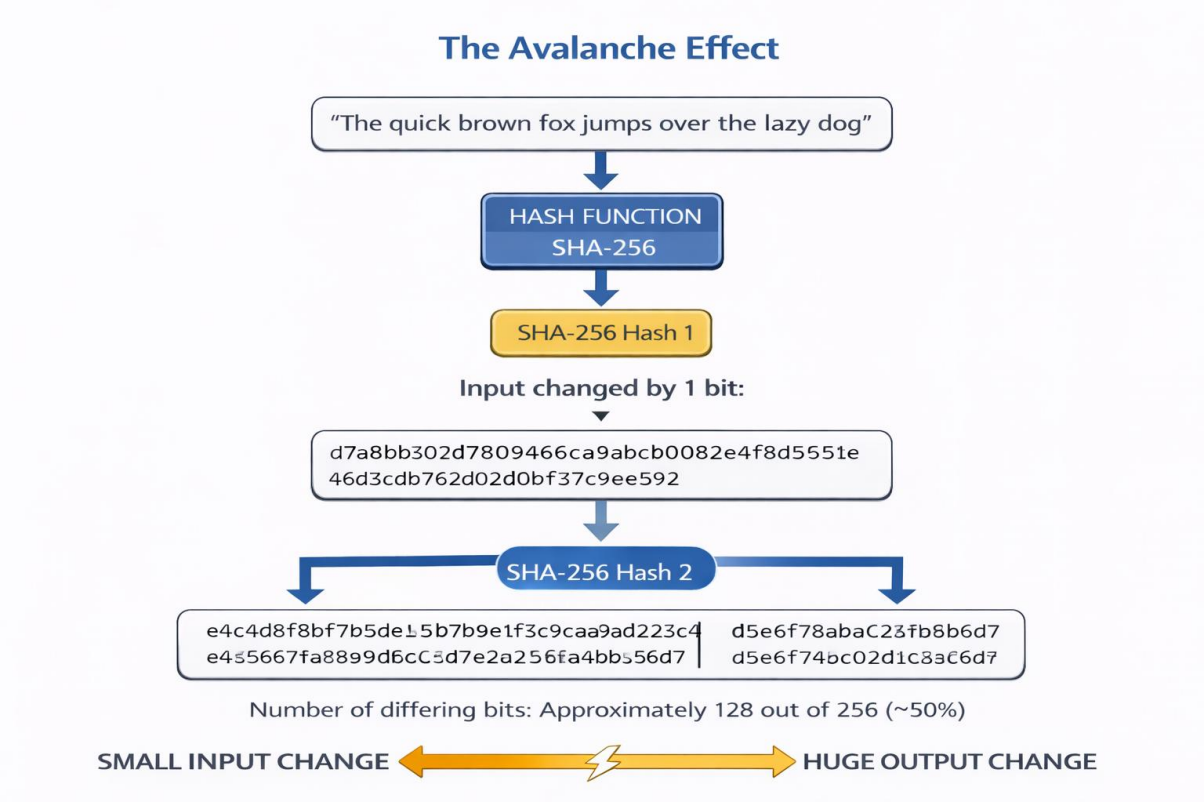


Mathematical Formulation:

Finding any pair (x, y) with $x \neq y$ such that $H(x) = H(y)$ should require approximately $2^{(n/2)}$ operations due to the birthday paradox.

The Avalanche Effect

A small change in the input—changing just one bit—should produce a dramatically different hash output that appears completely uncorrelated to the original hash. Ideally, about half of the output bits should flip for any single-bit change in the input. This property makes it impossible for an attacker to predict how changes to the input will affect the output.



Example of Avalanche Effect:

Input 1: "The quick brown fox jumps over the lazy dog"

SHA-256: d7a8fbb307d7809469ca9abcb0082e4f8d5651e46d3cdb762d02d0bf37c9e592

Input 2: "The quick brown fox jumps over the lazy cog"

SHA-256: e4c4d8f3bf7b5de15b7b9e1f3c9c9a9bd2c3c4d5e6f7a8b9c0d1e2f3a4b5c6d7

Number of differing bits: Approximately 128 out of 256 (50%)

Fixed Output Length

Regardless of whether the input is a single character or a multi-gigabyte file, the output hash is always the same length for a given algorithm. SHA-256 always produces 256 bits, regardless of input size. This property makes hash functions practical for applications like digital signatures, where signing a fixed-size hash is much more efficient than signing the entire message.

1.3 Applications in Modern Computing and Bitcoin

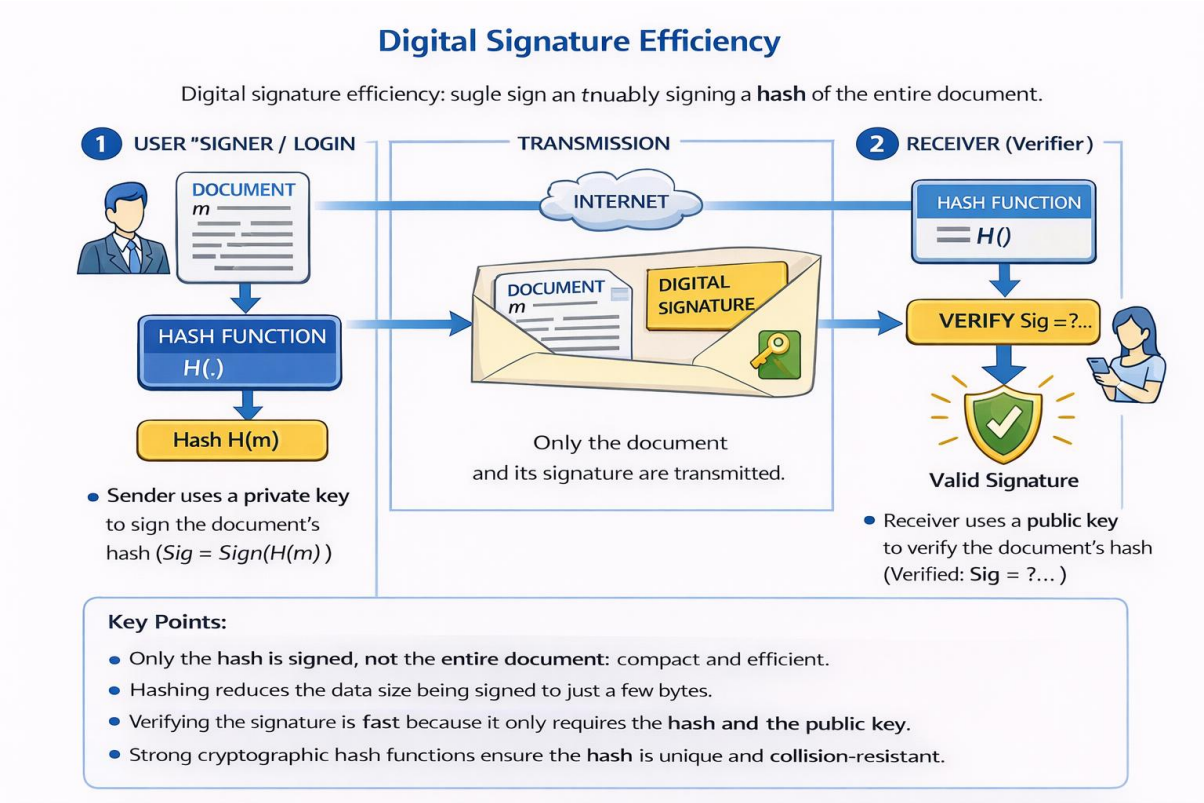
Hash functions are among the most versatile tools in cryptography, finding applications across virtually all areas of computer security:

Data Integrity Verification

When downloading files from the internet, especially software installers or system images, providers often publish hash values alongside the download. After downloading, users can compute the hash of their downloaded file and compare it to the published value. If they match, the file is identical to the original and hasn't been corrupted or tampered with during transmission.

Digital Signature Efficiency

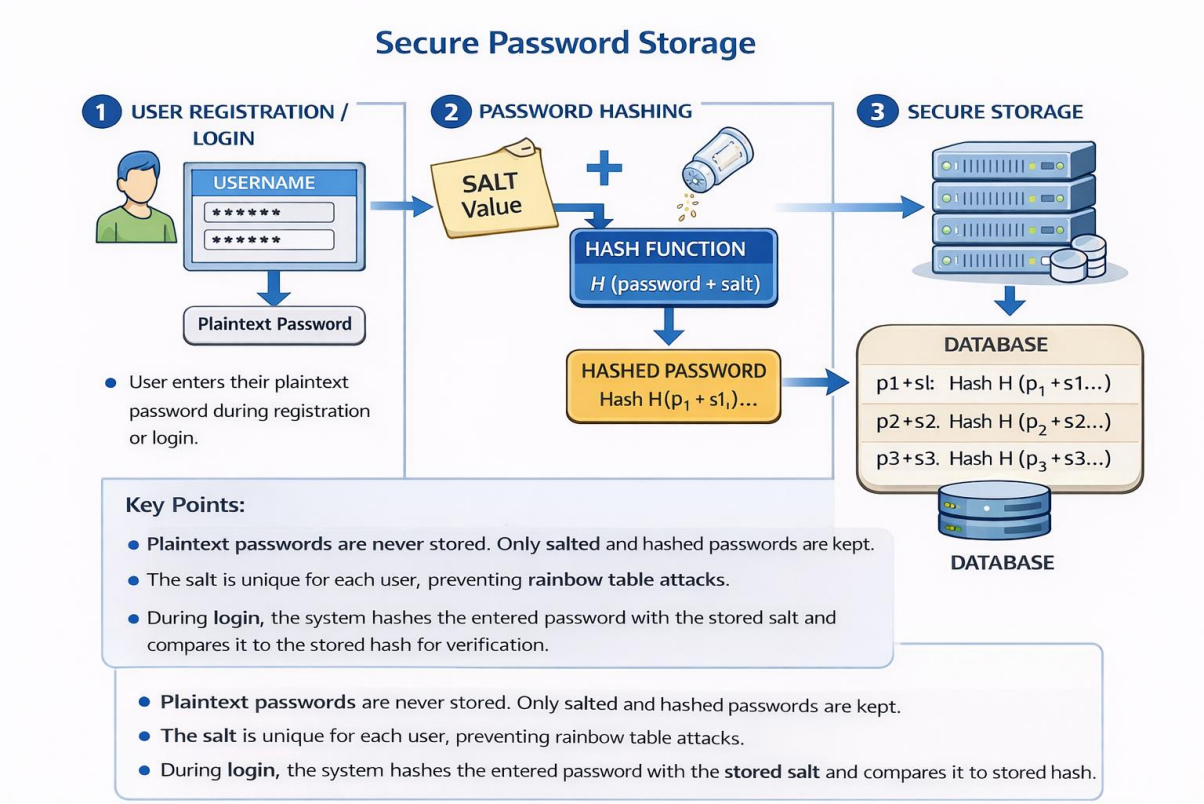
Digital signature algorithms are computationally expensive when operating on large messages. Instead of signing the entire message, systems hash the message first and then sign only the fixed-size hash. This provides the same security properties with vastly improved efficiency.



Password Storage

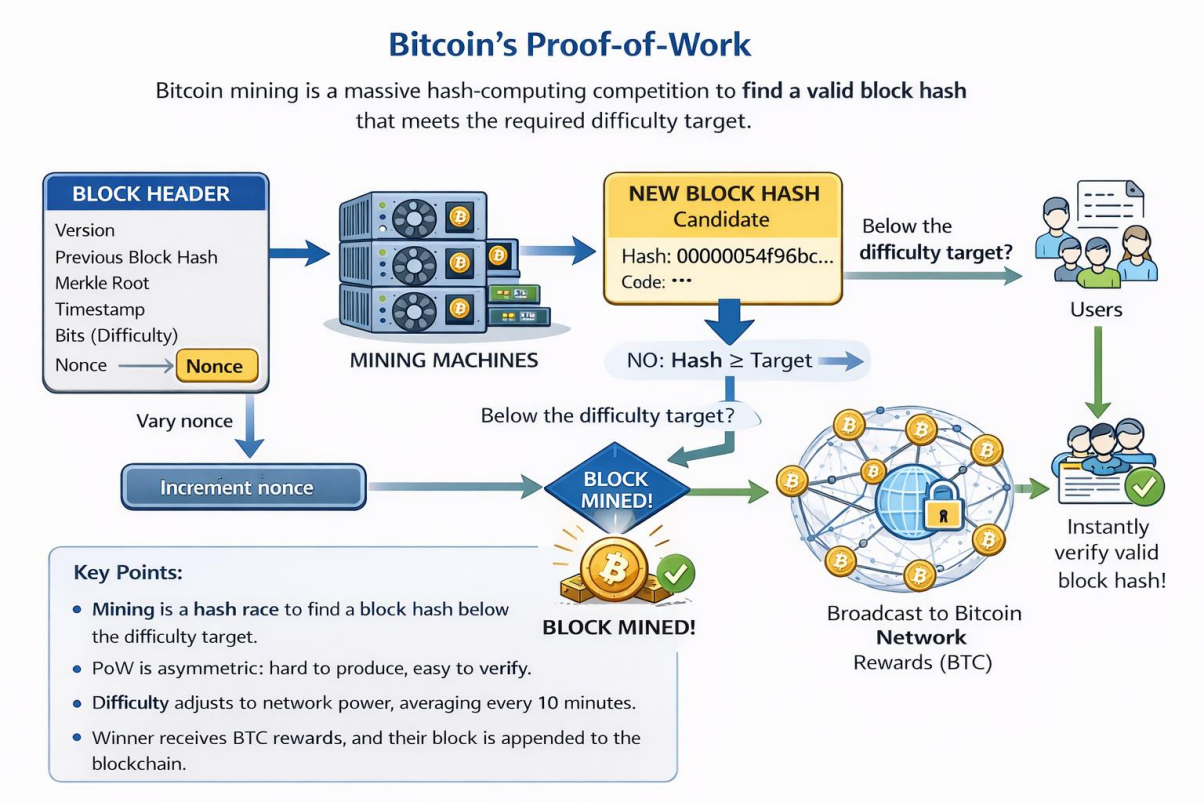
Modern systems never store passwords in plaintext. Instead, when a user creates an account, the system hashes the password and stores only the hash. During login, the system hashes the entered password and compares it to the stored hash. If an attacker breaches the database, they obtain only hashes, not the actual passwords. However, simple hashing is insufficient—modern

systems use specialized password hashing functions with additional security features like salting and key stretching.



Bitcoin's Proof-of-Work

Bitcoin mining is essentially a massive hash-computing competition. Miners repeatedly hash block headers, varying a small field called the nonce, until they find a hash that is below a target threshold. This process requires enormous computational effort but is instantly verifiable by anyone—they simply compute the hash once and check if it meets the requirement. This asymmetric effort (hard to produce, easy to verify) is the foundation of Bitcoin's security.



In **Bitcoin**, Proof-of-Work is the mechanism that decides **who gets to add the next block** to the blockchain and ensures that everyone agrees on the same transaction history.

What miners are actually doing

Each miner builds a **block header** that contains:

- Previous block hash
- Merkle root (summary of all transactions)
- Timestamp
- Difficulty target ("bits")
- A changing number called the **nonce**

The miner then applies a cryptographic hash function to this header.

If the resulting hash is **not small enough** (i.e., not below the difficulty target):

 Change the nonce
 Hash again
 Repeat millions/billions of times per second

This continues until one miner finally finds a hash that starts with enough leading zeros (meaning it is numerically **below the target**).

This is why mining is called a **brute-force guessing game**.

There is **no shortcut** — the only way is trial and error.

Why this is called "Proof-of-Work"

When a miner finds a valid hash, they are effectively proving:

"I spent real computational energy to find this solution."

Other nodes don't redo all that work. They simply:

- Hash the block once
- Check if the result meets the target

Verification takes milliseconds.

So:

✅Producing a block = extremely hard
✅Verifying a block = extremely easy

This is the **asymmetric effort** that secures Bitcoin.

How this provides security

1. Prevents cheating

If someone tries to change even **one transaction** in an old block:

- Merkle root changes
- Block hash changes

- Every block after it becomes invalid

To fix this, the attacker must **redo Proof-of-Work for that block and all following blocks**, faster than the entire global network — practically impossible.

2. Stops double spending

An attacker cannot easily rewrite history to spend the same coins twice, because recreating blocks requires massive computing power.

3. Makes attacks expensive

Any attack needs:

- Huge hardware
- Massive electricity
- Continuous mining power

This economic cost protects the network.

Difficulty adjustment

Bitcoin automatically adjusts mining difficulty about every **2016 blocks (~2 weeks)** so that:

One block is found roughly every **10 minutes**

If more miners join → difficulty increases
If miners leave → difficulty decreases

This keeps block creation stable.

Rewards for miners

The first miner to solve the puzzle:

- Adds the new block to the blockchain
- Receives newly created bitcoins (block reward)
- Collects transaction fees

This reward incentivizes honest participation.

In simple terms

Think of Proof-of-Work like a lottery where:

- Tickets cost electricity
- Everyone guesses numbers
- First correct guess wins
- Everyone else instantly confirms the win

The heavy work keeps Bitcoin decentralized, tamper-resistant, and trustless.

Merkle Trees in Blockchain

Bitcoin doesn't store all transactions individually in the block header. Instead, transactions are organized into a binary tree structure called a Merkle tree. Each leaf node is a transaction hash, and each parent node is the hash of its two children concatenated. The single hash at the root of this tree, the Merkle root, summarizes all transactions in the block. This allows efficient verification that a transaction is included in a block without downloading all transactions.

In **Bitcoin**, a block may contain **thousands of transactions**, but storing every transaction directly in the *block header*would make blocks very large and slow to verify. Instead, Bitcoin uses a **Merkle Tree** to *summarize* all transactions with a **single hash value called the Merkle Root**.

How it works (conceptually)

1. **Each transaction is hashed** (using cryptographic hash functions).
   These hashes become the **leaf nodes** of the tree.
2. **Pairs of hashes are concatenated and hashed again** to form parent nodes.
3. This pairing-and-hashing process continues upward level by level.
4. Finally, only **one hash remains at the top** — this is the **Merkle Root**, and it is stored in the block header.

So instead of storing *every transaction*, the block header stores **just one hash that represents all of them**.

Why this is powerful

Efficient verification (Merkle Proofs)

If you want to check whether a particular transaction is inside a block, you **don't need to download the entire block**.
You only need:

- The transaction hash
- A small set of intermediate hashes (called a *Merkle proof*)
- The Merkle root from the block header

With these, you can recompute the path to the root and verify inclusion.
This is especially important for **lightweight wallets (SPV nodes)**.

 Complexity drops from **O(n)** (checking all transactions) to **O(log n)**.

Strong data integrity

If *even one bit* of any transaction changes:

- Its hash changes
- Its parent hash changes
- Every level above changes
- The **Merkle Root changes**

Since the Merkle Root is inside the block header (which itself is hashed and chained to previous blocks), **tampering becomes immediately detectable**.

Scalability

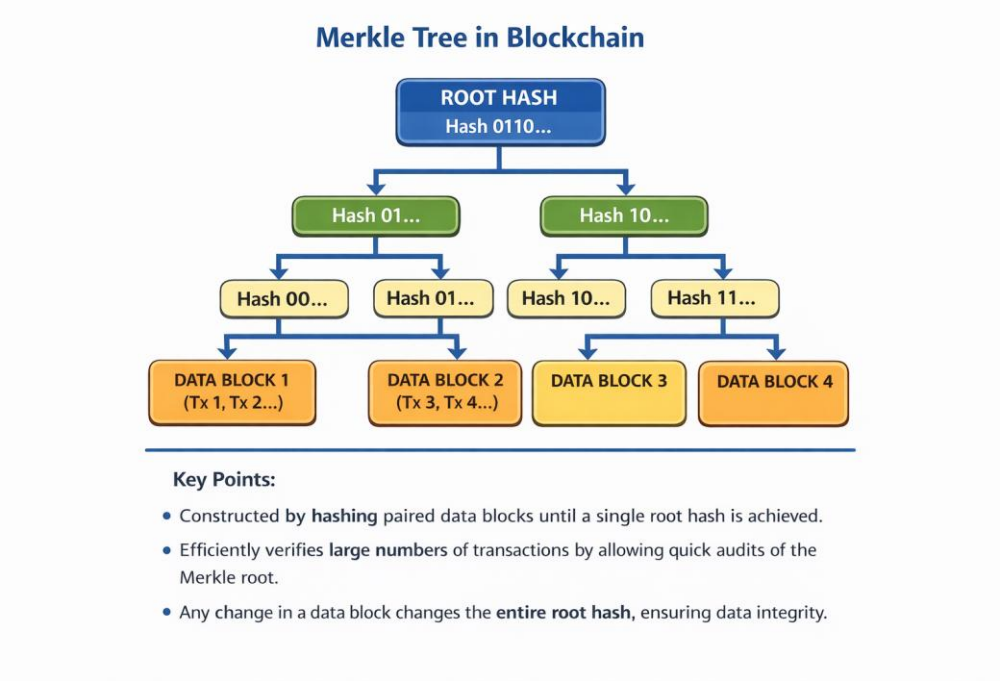
Merkle Trees allow Bitcoin to scale to large numbers of transactions while keeping:

- Block headers small
- Verification fast
- Storage efficient

Without this structure, blockchain systems would be far heavier and slower.

In simple words

 The Merkle Tree acts like a **cryptographic fingerprint of all transactions in a block**.
 The Merkle Root is a **compact summary** proving exactly *what* transactions belong to that block.
 It enables **fast checking, strong security, and efficient storage** — all critical for decentralized blockchains.



Practice Problems Set 1: Hash Function Fundamentals

Problem 1.1: Pre-Image Resistance Calculation

Suppose a hash function produces a 128-bit output. An attacker wants to find a pre-image for a given hash value. If the attacker can compute 1 million hashes per second, how long will it take on average to find a pre-image? Express your answer in years.

Solution :

For a 128-bit hash function, the output space size is 2^{128} possible hash values.

For pre-image resistance, the attacker needs to try inputs until finding one that hashes to the target value. With a ideal hash function, each trial has probability $1/2^{128}$ of success.

Average number of trials needed = 2^{128}

$2^{128} = 3.4 \times 10^{38}$ trials

At 1 million (10^6) trials per second:

Time in seconds = $(3.4 \times 10^{38}) / (10^6) = 3.4 \times 10^{32}$ seconds

Convert to years:

Seconds in a year = $365 \times 24 \times 3600 = 31{,}536{,}000 \approx 3.15 \times 10^7$

Time in years = $(3.4 \times 10^{32}) / (3.15 \times 10^7) = 1.08 \times 10^{25}$ years

Answer

Approximately 1.08×10^{25} years, which is vastly longer than the age of the universe (about 1.38×10^{10} years). This demonstrates why pre-image attacks are infeasible for well-designed hash functions.

Problem 1.2: Collision Resistance and Birthday Paradox

A hash function produces 64-bit outputs. Using the birthday paradox approximation, how many hash computations are needed to find a collision with probability greater than 50%? Compare this to the number needed for a pre-image attack.

Solution

Step 1: Understanding the birthday paradox

The birthday paradox states that with N possible outputs, you need approximately $\sqrt{(2N)}$ trials to have a 50% chance of finding a collision.

For $N = 2^{64}$ possible outputs:

Number of trials $\approx \sqrt{(2 \times 2^{64})} = \sqrt{(2^{65})} = 2^{\wedge}(65/2) = 2^{32.5}$

$2^{32.5} = 2^{32} \times \sqrt{2} = 4{,}294{,}967{,}296 \times 1.414 \approx 6.07 \times 10^9$ trials

Step 2: More precise calculation

The exact formula for probability of collision after k trials is:

$P(\text{collision}) \approx 1 - e^{\wedge}(-k^2/2N)$

Setting P = 0.5:

$0.5 = 1 - e^{(-k^2/2N)}$

$e^{(-k^2/2N)} = 0.5$

$-k^2/2N = \ln(0.5) = -0.693$

$k^2 = 2N \times 0.693 = 1.386N$

$k = \sqrt{(1.386N)} = \sqrt{(1.386 \times 2^{64})} = \sqrt{(1.386)} \times 2^{32} = 1.177 \times 2^{32}$

$k = 1.177 \times 4{,}294{,}967{,}296 \approx 5.06 \times 10^9$ trials

Step 3: Comparison with pre-image attack

For pre-image attack (finding a specific hash): Need $2^{64} \approx 1.84 \times 10^{19}$ trials

Ratio = $(1.84 \times 10^{19}) / (5.06 \times 10^9) \approx 3.64 \times 10^9$

Answer

- Collision attack: $\approx 5.06 \times 10^9$ trials

- Pre-image attack: $\approx 1.84 \times 10^{19}$ trials

- Collision attacks are about 3.6 billion times easier than pre-image attacks!

This is why hash functions need output lengths twice as large as the desired security level—to account for the birthday paradox advantage in collision attacks.

Problem 1.3: Avalanche Effect Verification

Given the following two inputs that differ by one bit:

Input A: 0x00000000

Input B: 0x00000001

Assume a simple hash function that computes $H(x) = x \oplus (x << 16) \oplus (x >> 16)$ (where $\oplus$ is XOR, << is left shift, >> is right shift, all on 32-bit values). Compute both hashes and determine what percentage of bits changed. Does this function demonstrate a good avalanche effect?

Solution

Step 1: Compute H(0x00000000)

Let x = 0x00000000 (32 bits all zeros)

x << 16 = 0x00000000 shifted left 16 bits = 0x00000000

x >> 16 = 0x00000000 shifted right 16 bits = 0x00000000

H(x) = 0x00000000 ⊕ 0x00000000 ⊕ 0x00000000 = 0x00000000

So H(0x00000000) = 0x00000000

In binary: 00000000 00000000 00000000 00000000 (32 zeros)

Step 2: Compute H(0x00000001)

x = 0x00000001 = binary: 00000000 00000000 00000000 00000001

x << 16 = 0x00010000 = binary: 00000000 00000001 00000000 00000000

x >> 16 = 0x00000000 = binary: 00000000 00000000 00000000 00000000

Now XOR all three:

x: 00000000 00000000 00000000 00000001

x << 16: 00000000 00000001 00000000 00000000

x >> 16: 00000000 00000000 00000000 00000000

H(x) = x⊕(x<<16)⊕(x>>16) = 00000000 00000001 00000000 00000001

So H(0x00000001) = 0x00010001

Step 3: Compare bits

H(0x00000000): 00000000 00000000 00000000 00000000

H(0x00000001): 00000000 00000001 00000000 00000001

Count bits that differ:

- Byte 1 (bits 31-24): 00000000 vs 00000000 → 0 bits differ

- Byte 2 (bits 23-16): 00000000 vs 00000001 → 1 bit differs (the least significant bit of this byte)

- Byte 3 (bits 15-8): 00000000 vs 00000000 → 0 bits differ

- Byte 4 (bits 7-0): 00000000 vs 00000001 → 1 bit differs (the least significant bit)

Total bits differing = 2 out of 32 bits = 6.25%

Step 4: Evaluate avalanche effect

A good avalanche effect should change approximately 50% of bits (16 out of 32). This hash function changed only 2 bits (6.25%), which is far too low.

Answer: The hash function changes only 6.25% of bits for a single-bit input change, demonstrating poor avalanche effect. An attacker could potentially predict how changes to input affect output, making this function unsuitable for cryptographic use.

Problem 1.4: Password Storage Security

A website stores passwords as simple MD5 hashes without salting. An attacker obtains the password database containing 10,000 hashes. The attacker has access to a rainbow table that covers all possible 8-character lowercase passwords (a-z) and requires 1 second per lookup. How long will it take to recover all passwords? What would be the impact of adding a random 32-bit salt to each password?

Solution

Step 1: Understand the rainbow table without salting

Without salting, all users with the same password have the same hash.

Number of possible 8-character lowercase passwords:

Each character: 26 possibilities

Total combinations = $26^8 = 208,827,064,576 \approx 2.09 \times 10^{11}$ passwords

A rainbow table precomputes hashes for all these passwords. With 1 second per lookup (this is unrealistically slow—real rainbow tables use parallel lookups in milliseconds, but we'll use the given numbers):

Time to look up one hash = 1 second

For 10,000 hashes = 10,000 seconds = 2.78 hours

However, many users will have the same password, so after finding one instance of "password123", all users with that password are recovered instantly.

Step 2: With 32-bit salt

Each password now has $2^{32} = 4{,}294{,}967{,}296$ possible salt values.

Number of hash computations needed for a full rainbow table:

$= 26^8 \times 2^{32} = 2.09 \times 10^{11} \times 4.29 \times 10^9 = 8.97 \times 10^{20}$ hashes

This is astronomically large—impossible to precompute.

Step 3: Time to crack salted passwords

Without a precomputed table, the attacker must compute hashes on the fly for each password guess, for each salt.

For each of the 10,000 users:

- Try all 26^8 passwords $= 2.09 \times 10^{11}$ hashes per user

- At 1 million hashes per second (typical for MD5 on modern hardware):

  Time per user $= 2.09 \times 10^{11} / 10^6 = 209{,}000$ seconds ≈ 2.42 days

For all 10,000 users sequentially = 24,200 days ≈ 66.3 years

Step 4: Parallel cracking

With 100 GPUs each doing 10^9 hashes/second:

Time $= (10{,}000 \times 2.09 \times 10^{11}) / (100 \times 10^9) = 2.09 \times 10^{15} / 10^{11} = 20{,}900$ seconds ≈ 5.8 hours

Answer:

- Without salt: 2.78 hours to recover all passwords

- With 32-bit salt: 5.8 hours with 100 GPUs, but requires actually computing hashes rather than just looking up precomputed values

- The salt forces the attacker to do computational work for each user individually, rather than using precomputed tables. This is why salting is essential for password storage.

Problem 1.5: Bitcoin Mining Difficulty

Bitcoin's difficulty is set so that a valid block hash must be less than a target value. If the target requires the hash to have the first 72 bits as zeros, and a miner has a hash rate of 100 TH/s (10^{14} hashes per second), how long will it take on average to find a valid block? Assume SHA-256 produces uniformly distributed outputs.

Solution

Step 1: Understanding the requirement

SHA-256 produces 256-bit outputs. Requiring the first 72 bits to be zero means the hash must be in the range:

[0, 2^(256-72) - 1] = [0, 2^{184} - 1]

Step 2: Probability of success per hash

Total possible hash values = 2^{256}

Valid hash values = 2^{184}

Probability per hash = 2^{184} / 2^{256} = 1 / 2^{72}

$2^{72} = 4.72 \times 10^{21}$

So probability = 1 / (4.72×10^{21})

Step 3: Expected number of hashes

Expected hashes needed = $2^{72} = 4.72 \times 10^{21}$ hashes

Step 4: Time calculation

Hash rate = 100 TH/s = 10^{14} hashes/second

Time in seconds = (4.72×10^{21}) / (10^{14}) = 4.72×10^{7} seconds

Convert to days:

Seconds in a day = 86,400

Time in days = (4.72×10^{7}) / 86,400 = 546 days

Step 5: Convert to more meaningful units

546 days = 1.5 years

Answer: At 100 TH/s, it would take approximately 1.5 years on average to find a valid block with 72 leading zero bits. This is why mining pools combine the hashing power of many miners—with 100,000 TH/s (100 PH/s), the time drops to about 0.55 days (13 hours).

Problem 1.6: Hash Length and Security

Compare the security of a 160-bit hash function versus a 256-bit hash function against:

a) Pre-image attacks

b) Collision attacks

Express the security difference in terms of the number of operations required.

Solution

Step 1: Pre-image attacks

For pre-image resistance, security is approximately 2^n operations.

For 160-bit hash: 2^{160} operations

For 256-bit hash: 2^{256} operations

Ratio = $2^{256} / 2^{160} = 2^{96}$

$2^{96} \approx 7.9 \times 10^{28}$

So the 256-bit hash is about 7.9×10^{28} times harder to break with a pre-image attack.

Step 2: Collision attacksFor collision resistance, due to birthday paradox, security is approximately 2^(n/2) operations.

For 160-bit hash: 2^{80} operations

For 256-bit hash: 2^{128} operations

Ratio = $2^{128} / 2^{80} = 2^{48}$

$2^{48} \approx 2.81 \times 10^{14}$

So the 256-bit hash is about 2.81×10^{14} times harder to break with a collision attack.

Step 3: Practical interpretation

To put these numbers in perspective:

$2^{80} \approx 1.2 \times 10^{24}$ operations

- With 1 billion hashes per second: 1.2×10^{15} seconds ≈ 38 million years

$2^{128} \approx 3.4 \times 10^{38}$ operations

- With 1 billion hashes per second: 3.4×10^{29} seconds ≈ 1.08×10^{22} years

Answer:

a) Pre-image: 256-bit is $2^{96} \approx 7.9 \times 10^{28}$ times stronger

b) Collision: 256-bit is $2^{48} \approx 2.81 \times 10^{14}$ times stronger

The difference is dramatic—this is why modern systems use at least 256-bit hash functions, as 160-bit (like SHA-1) is now considered too weak for collision resistance.

Problem 1.7: Real-World Application Analysis

A software distribution website provides SHA-1 hashes for all downloads. A security researcher discovers that SHA-1 is broken and practical collisions can be generated. Explain the specific attack scenarios that become possible and why this is dangerous even if the website uses HTTPS.

Solution

Step 1: Understanding the SHA-1 collision vulnerability

SHA-1 produces 160-bit hashes. Practical collision attacks (like the SHAttered attack) can find two different files with the same SHA-1 hash using about 2^{63} operations, which is feasible with modern computing clusters.

Step 2: Attack Scenario 1 - Malicious Software Substitution

1. Attacker creates two files:

- File A: Legitimate software (e.g., a PDF reader installer)

- File B: Malicious software (contains malware, backdoors, etc.)

2. Using collision attack techniques, attacker modifies both files (adding invisible padding, comments, etc.) until they have the same SHA-1 hash.

3. Attacker submits File A to the software vendor for signing. The vendor verifies the hash matches their records and signs File A.

4. The signature on File A is valid. Since File B has the same SHA-1 hash, the signature also verifies for File B!

5. Attacker distributes File B to victims. Their systems verify the signature (which matches the vendor's signature) and trust that it's legitimate software.

Step 3: Attack Scenario 2 - Man-in-the-Middle with HTTPS

Even with HTTPS, this attack works because:

1. HTTPS ensures the file hasn't been modified in transit from the official server.

2. However, if the attacker can get the victim to download from a different source (torrent, mirror site, USB drive), HTTPS doesn't protect that.

3. The victim verifies the SHA-1 hash against the official website's published hash (which is correct for both files).

4. The victim thinks: "The hash matches, and the official website says this hash is for the legitimate software, so this must be legitimate."

Step 4: Attack Scenario 3 - Git Repository Compromise

Git uses SHA-1 to identify commits. A collision attack could:

1. Create two different commits with the same hash

2. Convince a developer to review and approve one commit (seems harmless)

3. The repository actually contains the malicious commit

4. Since Git uses the hash as the identifier, the repository appears consistent

Step 5: Why it's dangerous despite HTTPS

HTTPS only protects the communication channel between client and server. It doesn't protect against:

- Collisions in the hash function itself

- Compromised mirrors or alternative download sources

- Time-of-check to time-of-use attacks (the hash is verified at download time, but the file might be swapped later)

- Social engineering where users are directed to download from untrusted sources

Step 6: Mitigation

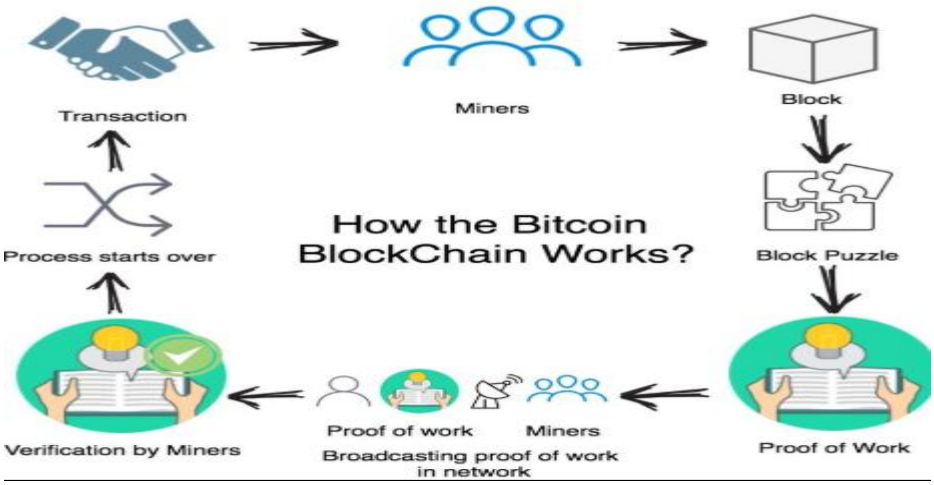
- Stop using SHA-1 for security-critical applications

- Use SHA-256 or SHA-3 instead

- Implement Certificate Transparency for code signing

- Use multiple hash functions simultaneously

Answer: SHA-1 collisions enable attackers to create two different files with identical hashes, allowing malicious software to inherit the valid signature of legitimate software. HTTPS doesn't prevent this because the attack exploits the hash function itself, not the communication channel. This is why SHA-1 has been deprecated for all security applications.



Chapter 2

Hash Algorithms in Depth

2.1 MD5 (Message-Digest Algorithm 5)

MD5 was developed by Ronald Rivest, one of the inventors of the RSA algorithm, in 1991. It was designed as an improvement over its predecessor, MD4, which had shown signs of cryptographic weakness. MD5 quickly became the most widely used hash function in the world, finding applications in file integrity checking, software distribution, and early digital signature schemes.

Technical Architecture

MD5 produces a 128-bit hash value, typically rendered as a 32-character hexadecimal number. The algorithm processes input messages in 512-bit blocks through a series of operations:

The process begins with padding the input message so that its length is congruent to 448 modulo 512. This means that after padding, the message is 64 bits short of being a multiple of 512 bits long. These remaining 64 bits are used to append a 64-bit representation of the original message length, creating a final message that is an exact multiple of 512 bits.

The algorithm maintains internal state in four 32-bit registers, initialized to specific constants. These registers, typically labeled A, B, C, and D, are updated as each 512-bit block is processed. The processing of each block involves four rounds, each containing 16 operations. Each operation performs a nonlinear function on three of the registers, adds the result to the fourth register, along with a message word and a constant, and then rotates the result.

The Four Rounds and Their Functions

Each round uses a different nonlinear function:

The first round uses the function $F(X,Y,Z) = (X \wedge Y) \vee (\neg X \wedge Z)$. This function selects between Y and Z based on the value of X.

The second round uses $G(X,Y,Z) = (X \wedge Z) \vee (Y \wedge \neg Z)$. This is similar to F but with different variable roles.

The third round uses $H(X,Y,Z) = X \oplus Y \oplus Z$, a simple XOR operation.

The fourth round uses $I(X,Y,Z) = Y \oplus (X \vee \neg Z)$, a more complex combination.

Security Analysis and Vulnerabilities

Despite its widespread adoption, MD5's security began to erode as cryptanalytic techniques advanced. In 1996, Dobbertin announced a collision for the compression function of MD5, though not for the full hash function. This was an early warning sign that went largely unheeded.

The critical breakthrough came in 2004, when Chinese researcher Wang Xiaoyun and her team demonstrated practical collisions for the full MD5 algorithm. They showed that they could create two different messages that produced the same MD5 hash, breaking the collision resistance property entirely.

This vulnerability has real-world implications. Attackers have demonstrated the ability to create two different executable programs with the same MD5 hash—one benign and one malicious. They could trick a software vendor into signing the benign version with a digital signature, then substitute the malicious version that would verify with the same signature.

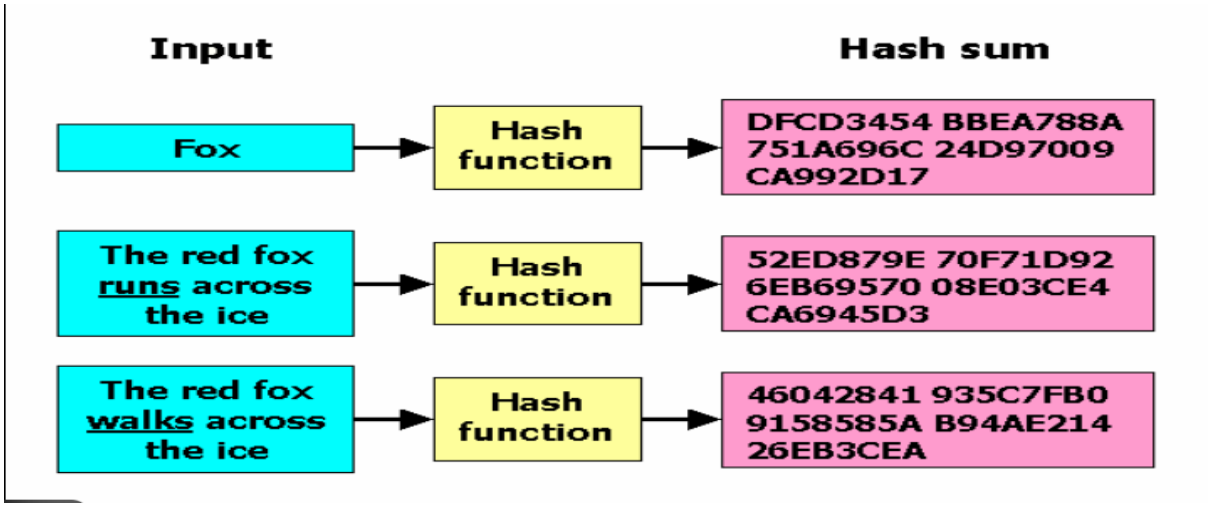
In 2008, researchers used MD5 collisions to create a rogue Certificate Authority certificate, demonstrating that MD5 was completely broken for security applications.

Current Status and Legacy Use

MD5 is now considered cryptographically broken and should never be used for security-sensitive applications. However, it still appears in some non-security contexts:

Some file-sharing systems use MD5 for detecting file corruption during transmission, where the threat model is random errors rather than malicious tampering. Version control systems like Git use MD5 for identifying objects, though they rely on the assumption that collisions in this context are extremely unlikely rather than cryptographically impossible.

The lesson of MD5 serves as a cautionary tale in cryptography—algorithms that appear secure may have hidden weaknesses, and cryptographic standards must evolve as attacks improve.



2.2 The Secure Hash Algorithm (SHA) Family

The **Secure Hash Algorithm (SHA) family** is a collection of cryptographic hash functions designed to ensure data integrity, authentication, and security in modern digital systems. These algorithms transform an input of arbitrary size—such as a file, password, or message—into a fixed-length output called a *hash value* or *message digest*. A key property of SHA algorithms is determinism: the same input always produces the same hash, while even a tiny change in input results in a completely different output. This behavior makes SHA extremely valuable for detecting data tampering, verifying authenticity, and supporting secure communications.

The SHA family was originally developed by the National Security Agency (NSA) and later standardized and published by the National Institute of Standards and Technology (NIST). Over time, multiple versions have been released, including SHA-0, SHA-1, SHA-2, and SHA-3, each improving upon the security and design of its predecessors. Earlier versions such as SHA-1 are now considered cryptographically weak due to discovered collision vulnerabilities, leading to widespread migration toward stronger variants like SHA-256 and SHA-3 in modern systems.

At its core, SHA operates by processing data in fixed-size blocks through a series of mathematical operations involving bitwise logic, modular addition, and compression functions. The result is a compact fingerprint of the original data. One of the most important characteristics of SHA is its *one-way nature*: while it is easy to compute a hash from input data, it is computationally infeasible to reconstruct the original data from the hash. Additionally,

SHA algorithms are designed to resist collisions (two different inputs producing the same hash), preimage attacks, and second-preimage attacks—properties that are fundamental to cryptographic security.

The SHA family plays a critical role across many areas of computing and cybersecurity. It is widely used in password storage, digital signatures, SSL/TLS certificates, blockchain technologies, software integrity verification, and secure file transfer. In password systems, for example, SHA ensures that actual passwords are never stored directly; instead, only their hashes are kept. In digital signatures and certificates, SHA guarantees that documents or messages have not been altered during transmission.

In summary, the Secure Hash Algorithm family forms a cornerstone of modern cryptography. By providing reliable mechanisms for data verification and integrity assurance, SHA enables trust in online transactions, cloud services, embedded systems, and communication networks. As computational power increases and attack methods evolve, newer SHA standards continue to be developed and adopted, ensuring that cryptographic hashing remains robust against emerging threats.

2.2.1 SHA-1

SHA-1 was designed as an improvement over SHA-0, which had an undisclosed flaw corrected shortly after its release. It produces a 160-bit hash value and was widely deployed in security protocols including SSL/TLS, PGP, SSH, and IPsec.

The algorithm structure closely resembles MD5 but with a larger hash size and a more conservative design. It processes messages in 512-bit blocks and maintains internal state in five 32-bit registers (A, B, C, D, E) instead of MD5's four registers.

For many years, SHA-1 was considered secure, though cryptanalysts expressed concern about its similarity to MD5. In 2005, the same team that broke MD5 announced the first theoretical attacks against SHA-1, reducing its security margin significantly.

The final blow came in 2017 when Google and CWI Amsterdam announced the SHAttered attack—the first practical collision against SHA-1. They demonstrated that it was possible to create two different PDF files with the same SHA-1 hash, requiring approximately 6500 CPU years and 110 GPU years of computation. While this was computationally expensive, it proved that SHA-1 collisions were feasible with sufficient resources.

Following this demonstration, major technology companies and browser vendors began deprecating SHA-1 certificates, and NIST officially withdrew SHA-1 from the Digital Signature Standard.

2.2.2 The SHA-2 Family

SHA-2 represents a significant advancement over its predecessors, offering stronger security through larger hash sizes and a more complex structure. The family includes several variants that differ in output length:

SHA-224 produces a 224-bit hash, truncated from SHA-256. It's designed for applications where a smaller hash size is beneficial while maintaining security equivalent to 112-bit symmetric keys.

SHA-256 produces a 256-bit hash and has become the workhorse of modern cryptography. It's used in Bitcoin's proof-of-work, TLS certificates, code signing, and countless other applications. The "256" refers to the output length in bits, corresponding to 32 bytes or 64 hexadecimal characters.

SHA-384 produces a 384-bit hash, truncated from SHA-512. It offers security equivalent to 192-bit symmetric keys.

SHA-512 produces a 512-bit hash, offering the strongest security in the SHA-2 family. It's particularly efficient on 64-bit processors, which can process its 64-bit word operations natively.

Internal Structure

SHA-2 follows the **Merkle–Damgård construction**, which means the input message is first padded and divided into fixed-size blocks, and each block is processed sequentially through a *compression function*. The output of one block becomes the input state for the next block, creating a chaining effect until the entire message is consumed. This structure ensures that messages of any length can be hashed securely, while producing a fixed-size digest. SHA-224 and SHA-256 operate on **512-bit message blocks using 32-bit words**, whereas SHA-384 and SHA-512 use **1024-bit blocks with 64-bit words**, allowing higher variants to achieve stronger security and better resistance against brute-force attacks.

At the heart of SHA-2 is its **compression function**, which updates an internal state consisting of eight working variables. Each round applies a carefully designed mix of **bitwise logical operations (AND, OR, XOR)**, **bit rotations**, and **modular addition**. These operations are computationally efficient on modern processors while providing strong cryptographic properties. Rotations ensure that bits from different positions influence each other, while modular addition introduces non-linearity. Together, these operations create the *avalanche effect*, where even a one-bit change in the input produces a dramatically different output hash.

SHA-2 employs **six logical functions** to achieve *confusion* (making relationships between input and output complex) and *diffusion* (spreading input changes across many output bits). Two primary Boolean functions—often called **Choose (Ch)**and **Majority (Maj)**—control how bits from different working variables are selected or combined during each round. Additionally, four rotation-based functions (commonly denoted as $\Sigma 0$, $\Sigma 1$, $\sigma 0$, and $\sigma 1$) mix bits across word boundaries using combinations of right rotations and shifts. These functions ensure that every bit of the message gradually influences many parts of the internal state, strengthening resistance against cryptanalytic attacks.

Another critical component is the **message schedule**. Instead of directly using only the original 16 words from each message block, SHA-2 expands them into **64 words for SHA-256** (or **80 words for SHA-512**). This expansion is achieved through recursive formulas involving rotations, shifts, and additions. Each newly generated word depends on several earlier words, meaning information from the original message is repeatedly reintroduced during later rounds. This extensive expansion dramatically increases mixing, ensuring that early message bits continue to affect computations far into the algorithm, improving diffusion and preventing localized weaknesses.

During processing, each round updates the eight working variables using constants derived from mathematical properties of prime numbers, along with the scheduled message words and logical functions. Over 64 rounds (SHA-256) or 80 rounds (SHA-512), the internal state becomes highly scrambled. After all rounds are complete, the updated values are added back into the original hash state, and this becomes the input for the next message block. Once the final block is processed, the concatenation of these state variables forms the final hash digest.

In essence, SHA-2's strength comes from the **tight integration of Boolean logic, rotations, modular arithmetic, and message expansion**. The Merkle–Damgård framework provides

scalability, while the improved compression design ensures strong avalanche behavior, collision resistance, and preimage resistance. These architectural choices make SHA-2 suitable for critical applications such as digital signatures, password hashing, blockchain systems, and secure communications.

Security Status

As of 2024, **SHA-2 remains cryptographically secure**, with no known practical attacks against its full-round implementations such as SHA-256 and SHA-512. While researchers have published *theoretical cryptanalytic results*—including reduced-round collision or differential attacks—these apply only to simplified or truncated versions of the algorithm and **do not threaten the complete standard designs**. Such studies are valuable because they help evaluate the *security margin* (the gap between current attack capabilities and the algorithm's full strength), but they have not translated into real-world compromises of SHA-2.

Collision resistance and pre-image resistance remain intact for SHA-2. A collision attack would require computational effort far beyond current technological limits (approximately 2^{128} operations for SHA-256), making it infeasible even for nation-state adversaries. Similarly, pre-image attacks—attempts to reconstruct original input data from a hash—are computationally impractical due to the algorithm's strong avalanche properties and nonlinear structure. The extensive round functions, message expansion, and modular arithmetic used in SHA-2 ensure that attackers cannot exploit structural weaknesses in any meaningful way.

Because of this continued robustness, SHA-2 is still formally recommended by the National Institute of Standards and Technology for most cryptographic purposes, including digital signatures, TLS certificates, password hashing (when combined with salting and key stretching), blockchain systems, and integrity verification. Although SHA-3 exists as a newer alternative with a completely different internal design, SHA-2 remains the dominant standard in deployed systems due to its proven reliability, widespread hardware acceleration, and extensive real-world testing over more than two decades.

Importantly, SHA-2 also benefits from conservative engineering choices: its large internal state, long digest sizes, and carefully constructed Boolean functions provide strong resistance against emerging attack strategies. Even advances in hardware, GPU computing, and distributed processing have not significantly weakened SHA-2's practical security. While

quantum computing poses theoretical long-term concerns for many cryptographic primitives, SHA-2 can be adapted simply by increasing hash lengths, preserving its usefulness in post-quantum transition strategies.

In summary, SHA-2 continues to represent a **high-confidence cryptographic primitive**. Despite minor reductions in theoretical security margins through academic analysis, the full algorithm remains resistant to collision and pre-image attacks. For this reason, SHA-2 is still considered a **current best-practice standard** for secure hashing in modern cryptographic applications, balancing strong security guarantees with excellent performance and broad ecosystem support.

2.2.3 SHA-3

SHA-3 represents a **major architectural shift** from traditional hash function designs used in earlier SHA families. Unlike SHA-1 and SHA-2, which are based on the Merkle–Damgård construction, SHA-3 adopts an entirely different framework known as the **sponge construction**. This change was intentional: SHA-3 was not created to replace SHA-2 immediately, but to serve as a *cryptographically independent alternative* in case structural weaknesses were ever discovered in SHA-2. To achieve this diversity, the National Institute of Standards and Technology launched an open, global competition in 2007 that ran until 2012, inviting researchers worldwide to submit candidate hash algorithms.

The competition emphasized transparency, peer review, and rigorous cryptanalysis. Over five years, dozens of proposals were examined by the international cryptographic community, with multiple rounds of evaluation focusing on security strength, performance, simplicity, and resistance to emerging attack models. This process mirrored earlier open competitions in spirit, ensuring that SHA-3 would benefit from extensive public scrutiny before standardization. Importantly, SHA-3 was designed to complement SHA-2 rather than replace it, providing *algorithmic diversity*—a critical principle in cryptography that reduces systemic risk.

The winning design, **Keccak**, introduced a radically different internal structure centered on *permutations instead of compression functions*. Rather than processing message blocks sequentially with chaining variables, Keccak absorbs input data into a large internal state and later "squeezes" out the hash output. This sponge-based approach naturally supports variable-length outputs and enables additional cryptographic primitives such as extendable-output

functions (XOFs). It also offers strong resistance against length-extension attacks, a known limitation of Merkle–Damgård–based hashes.

Keccak was created by a team of cryptographers led by Guido Bertoni, Joan Daemen, Michaël Peeters, and Gilles Van Assche. Their design philosophy emphasized mathematical clarity, compact implementation, and strong diffusion properties. The algorithm relies on simple bitwise operations applied over a large 1600-bit internal state, arranged in a 5×5 matrix of lanes, which are repeatedly transformed through carefully structured permutation rounds. This design provides excellent avalanche behavior while remaining efficient in both software and hardware.

Another significant advantage of SHA-3 is its flexibility. Beyond standard fixed-length hashes (such as SHA3-256 and SHA3-512), it supports extendable-output variants that allow arbitrary-length digests, making it suitable for advanced cryptographic protocols. Furthermore, SHA-3's internal structure is fundamentally different from SHA-2, meaning that any future attack on SHA-2 is unlikely to apply directly to SHA-3—a key reason it is viewed as a strategic safeguard for long-term cryptographic resilience.

In summary, SHA-3 is not merely a newer hash function but a **conceptual evolution in hash design**. Born from a global competition and grounded in a novel sponge architecture, it provides a robust, independent security foundation for modern and future cryptographic systems. While SHA-2 remains widely trusted and deployed, SHA-3 stands as a powerful alternative, ensuring continuity of secure hashing even in the face of unforeseen cryptanalytic breakthroughs.

The Sponge Construction

Unlike traditional hash functions such as MD5, SHA-1, and SHA-2, which rely on the Merkle–Damgård construction, SHA-3 is built on a fundamentally different paradigm known as the **sponge construction**. In this model, data is processed through a large internal state rather than a sequence of chained compression functions. The sponge framework operates in two distinct phases: *absorption* and *squeezing*. During absorption, the input message is XORed into part of the internal state in fixed-size blocks, with a permutation applied after each block. Once all input data has been absorbed, the algorithm transitions to the squeezing phase, where output

bits are extracted from the state, again interleaved with permutations, until the desired hash length is produced.

A defining feature of the sponge construction is its separation of the internal state into two regions: the **rate**, which directly interacts with input and output, and the **capacity**, which remains hidden and provides security. The size of the capacity determines resistance to collisions and pre-image attacks, while the rate controls performance. Because a large portion of the state is never directly exposed, attackers have far less information to exploit, significantly strengthening cryptographic resilience. This architectural choice allows SHA-3 to achieve strong diffusion, ensuring that every absorbed bit rapidly influences the entire internal state.

One major advantage of this design is its **natural resistance to length-extension attacks**. In Merkle–Damgård hashes, attackers can append data to a hashed message and predict the resulting hash without knowing the original input—an issue that affects algorithms like MD5 and SHA-1. SHA-3 avoids this vulnerability entirely because the internal state after hashing is not directly reusable for further message absorption. The squeezing phase is strictly separated from absorption, preventing adversaries from extending a hash without full knowledge of the original state.

Another powerful benefit is **output-length flexibility**. The sponge construction allows SHA-3 to generate hashes of arbitrary length using the same core permutation. This capability supports both fixed-length digests (such as SHA3-256 and SHA3-512) and extendable-output functions (XOFs), making SHA-3 highly adaptable for modern cryptographic applications including key derivation, random number generation, and domain-specific security protocols.

Additionally, SHA-3 offers what is known as **graceful or smooth security degradation**. If future cryptanalysis were to weaken the internal permutation, the security of the hash does not collapse suddenly. Instead, resistance gradually decreases in proportion to the attack strength. This contrasts with many Merkle–Damgård–based designs, where a structural flaw can lead to catastrophic failure. The sponge model therefore provides a more robust long-term security profile.

In essence, SHA-3's sponge construction represents a shift toward *state-centric cryptographic design*. By absorbing data into a large internal state and squeezing output afterward, SHA-3 achieves superior resistance to structural attacks, flexible output generation, and improved fault

tolerance. These properties make SHA-3 not just an alternative to SHA-2, but a forward-looking foundation for secure hashing in evolving threat environments.

Internal State

SHA-3 (based on the Keccak design) maintains a **large 1600-bit internal state**, which is organized as a **5×5 grid of 64-bit lanes (words)**. This two-dimensional arrangement is not arbitrary—it allows the algorithm to mix bits both horizontally and vertically, creating strong diffusion across the entire state. Each lane interacts with multiple neighbors during processing, ensuring that even a single input bit rapidly influences many distant parts of the state. This structure is one of the key reasons SHA-3 achieves excellent avalanche behavior and resistance to cryptanalytic attacks.

During hashing, this 1600-bit state is repeatedly transformed through a sequence of permutation rounds. Each round consists of **five carefully designed step mappings**—commonly referred to as Theta, Rho, Pi, Chi, and Iota. Together, these steps provide complementary cryptographic properties. Theta spreads parity information across columns (global diffusion), Rho and Pi rotate and rearrange bits (spatial dispersion), Chi introduces nonlinearity using Boolean logic, and Iota injects round constants to prevent symmetry. The combined effect is that patterns in the input are quickly destroyed, producing highly unpredictable internal states after only a few rounds.

A major advantage of this design is the **very large state size** compared to earlier hash functions. Traditional algorithms like SHA-2 operate with much smaller internal states, whereas SHA-3's 1600 bits provide a substantial **security margin**. This means that even if future research discovers partial weaknesses in the permutation, attackers would still face enormous computational barriers to mounting practical collision or pre-image attacks. In cryptographic terms, SHA-3 is engineered for *graceful degradation*: security decreases slowly rather than collapsing suddenly.

The large internal state also enables SHA-3 to separate its memory into *rate* (used for absorbing input and squeezing output) and *capacity* (kept hidden for security). A significant portion of the 1600 bits is reserved purely for protection, making it extremely difficult for attackers to infer internal behavior from observed hashes. This architectural choice strengthens resistance against advanced attack techniques and contributes to SHA-3's long-term robustness.

From an implementation perspective, the 5×5 lane structure maps very naturally to parallel hardware architectures. Because many operations occur independently across lanes, SHA-3 achieves **high throughput in hardware implementations**, such as FPGAs and ASICs, while remaining relatively compact. This makes it attractive for embedded systems, secure processors, and high-speed cryptographic accelerators. Despite its strong security properties, SHA-3 remains efficient due to its reliance on simple bitwise operations and rotations.

Standardized by the National Institute of Standards and Technology, SHA-3 was designed not merely as another hash function, but as a future-proof alternative to SHA-2. Its large internal state, permutation-based rounds, and hardware-friendly structure collectively provide both **cryptographic strength and practical performance**, ensuring resilience against evolving attack models for decades to come.

Variants

SHA-3 includes four hash function variants:

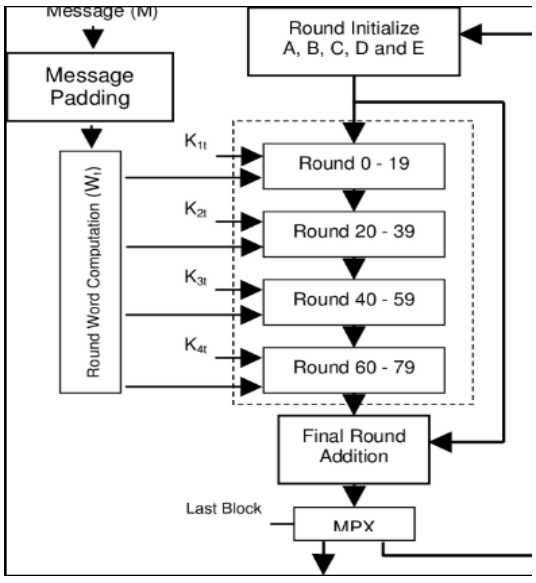
- SHA3-224: 224-bit output, security equivalent to 112-bit symmetric keys

- SHA3-256: 256-bit output, replacing SHA-256 in new applications

- SHA3-384: 384-bit output

- SHA3-512: 512-bit output

Additionally, SHA-3 includes two extendable-output functions (XOFs), SHAKE128 and SHAKE256, which can produce output of any desired length.



Practice Problems Set 2: Hash Algorithms in Depth

Problem 2.1: MD5 Collision Attack Analysis

In 2004, Wang Xiaoyun demonstrated MD5 collisions with complexity 2^{39} operations. If a modern computer can perform 10^9 MD5 operations per second, how long would it take to find an MD5 collision using this attack? Compare this to brute force collision search (which would require about 2^{64} operations due to birthday paradox).

Solution

Step 1: Time for Wang's attack

Attack complexity = 2^{39} operations

$2^{39} = 549,755,813,888 \approx 5.50 \times 10^{11}$ operations

At 10^9 operations per second:

Time = $5.50 \times 10^{11} / 10^9 = 550$ seconds ≈ 9.17 minutes

Step 2: Time for brute force collision search

Brute force collision search requires approximately 2^(n/2) = 2^{64} operations for 128-bit MD5

$2^{64} = 1.84 \times 10^{19}$ operations

At 10^9 operations per second:

Time = $1.84 \times 10^{19} / 10^9 = 1.84 \times 10^{10}$ seconds

Convert to years:

Seconds per year = 3.15×10^7

Time in years = $(1.84 \times 10^{10}) / (3.15 \times 10^7) = 584$ years

Step 3: Comparison

Wang's attack: 9.17 minutes

Brute force: 584 years

Ratio = (584 years) / (9.17 minutes) = 584 × 365 × 24 × 60 / 9.17

= 584 × 525,600 / 9.17

= 306,950,400 / 9.17 ≈ 33.5 million times faster

Answer: Wang's attack finds MD5 collisions in about 9 minutes, while brute force would take 584 years—a speedup of about 33.5 million times. This demonstrates why MD5 is completely broken for security applications.

Problem 2.2: SHA-1 SHAttered Attack

The SHAttered attack against SHA-1 required approximately 2^{63} operations. If Google used a cluster of GPUs that could perform 2^{50} operations per day, how many days did the attack take? How many GPU-years was this?

Solution

Step 1: Understand the numbers

Attack complexity = 2^{63} operations

Processing power = 2^{50} operations per day

Step 2: Calculate days needed

Days = 2^{63} / 2^{50} = 2^{13} days

2^{13} = 8,192 days

Step 3: Convert to years

Years = 8,192 / 365 = 22.44 years

Step 4: GPU-years

Since the cluster performs 2^{50} operations per day, and they ran for 8,192 days:

Total GPU-days = 8,192

GPU-years = 8,192 / 365 = 22.44 GPU-years

Step 5: Verify with actual SHAttered announcement

The actual SHAttered attack used approximately:

- 6,500 CPU-years

- 110 GPU-years

Our simplified calculation gives 22.44 GPU-years, which is lower because we assumed a much faster per-device rate (2^{50} operations per day ≈ 11.6 million operations per second, which is quite fast for a single GPU in 2017). The actual attack used many GPUs over time.

Answer: At 2^{50} operations per day, the attack would take 8,192 days (22.44 years) or 22.44 GPU-years. The actual SHAttered attack achieved this in a shorter time by using parallel processing across many GPUs simultaneously.

Problem 2.3: SHA-256 Security Margin

SHA-256 is considered to have 128-bit security against collision attacks. Explain what this means numerically. If computing power doubles every 18 months (Moore's Law), how many years until a collision attack becomes feasible (assuming 2^{80} operations is considered feasible)?

Solution

Step 1: Understanding 128-bit security

128-bit security means the best known attack requires approximately 2^{128} operations.

$2^{128} = 3.4 \times 10^{38}$ operations

To put this in perspective:

- If every atom in the universe (≈ 10^{80} atoms) were a computer performing 10^{12} operations per second, it would still take billions of years.

Step 2: Feasibility threshold

We'll consider 2^{80} operations as the boundary of feasibility.

$2^{80} = 1.2 \times 10^{24}$ operations

With current technology (say 10^{15} operations per second globally), this would take:

$1.2 \times 10^{24} / 10^{15} = 1.2 \times 10^{9}$ seconds ≈ 38 years

So 2^{80} is at the edge of feasibility with massive parallelization.

Step 3: Calculate the gap

Current security: 2^{128} operations

Feasibility threshold: 2^{80} operations

Gap factor = $2^{128} / 2^{80} = 2^{48} = 2.81 \times 10^{14}$ times harder

Step 4: Moore's Law calculation

If computing power doubles every 18 months (1.5 years), the number of doublings needed:

$2^k = 2^{48}$

k = 48 doublings

Time = k × 1.5 years = 48 × 1.5 = 72 years

Step 5: Important caveats

- Moore's Law is slowing down; doubling every 18 months may not continue

- Quantum computers could potentially break this faster using Grover's algorithm (which would reduce security to 2^{64} for collision attacks)

- This assumes attack complexity remains constant; cryptanalysis might find better attacks

Answer: At current Moore's Law rates, it would take about 72 years for computing power to make SHA-256 collisions feasible. However, this is a rough estimate—realistically, SHA-256 is considered secure for the foreseeable future.

Problem 2.4: SHA-3 Sponge Construction

In SHA-3's sponge construction, the internal state is 1600 bits. For SHA3-256, the rate r = 1088 bits and capacity c = 512 bits. Explain what rate and capacity mean. If an attacker wants to find a pre-image, how many operations are theoretically required? How does this relate to the capacity?

Solution

Step 1: Understanding rate and capacity in sponge construction

In a sponge construction:

- Total state size = rate (r) + capacity (c) = 1600 bits

- Rate (r): Number of bits absorbed from input in each block

- Capacity (c): The hidden part of the state that provides security

For SHA3-256:

- r = 1088 bits (absorbed in each block)

- c = 512 bits (hidden, not directly output)

- Output length = 256 bits (truncated from the state)

Step 2: Security implications of capacity

The security of a sponge function against pre-image and collision attacks is determined by the capacity:

- Collision resistance: ~2^(c/2) operations

- Pre-image resistance: ~2^(c/2) for first pre-image, ~2^(c/2) for second pre-image

- For SHA3-256 with c = 512:

  - Collision resistance: 2^(512/2) = 2^{256} operations

  - Pre-image resistance: also 2^{256} operations

Step 3: Compare with SHA-256

SHA-256 (Merkle-Damgård) provides:

- Collision resistance: 2^{128} operations (due to birthday bound on 256-bit output)

- Pre-image resistance: 2^{256} operations

SHA3-256 provides:

- Collision resistance: 2^{128} operations (due to 256-bit output truncation, despite 2^{256} theoretical from capacity)

- Pre-image resistance: 2^{256} operations

Step 4: Why capacity matters

The capacity ensures that even if an attacker can control the rate (input) bits, they cannot control the capacity bits. This prevents various attacks:

- Length extension attacks become impossible because attacker doesn't know capacity state

- The security proof shows that distinguishing the sponge from random requires at least 2^(c/2) operations

Answer: For SHA3-256 with c=512, pre-image attacks theoretically require 2^{256} operations, matching SHA-256. The capacity provides a security margin even if the output is truncated. This is why SHA-3 with c=512 offers 256-bit pre-image resistance despite having a 256-bit output.

Problem 2.5: Algorithm Transition Planning

A company currently uses SHA-1 for code signing. They plan to migrate to SHA-256. If they have 10,000 signed files that need to remain verifiable for 10 years, and they issue new signatures every month, what are the risks during the transition period? Propose a migration strategy.

Solution

Step 1: Identify risks during transition

Risk 1: Dual acceptance period

If verification systems accept both SHA-1 and SHA-256 signatures during transition:

- Attackers could create SHA-1 collisions for new files

- Users might accept SHA-1 signatures on malicious files thinking "it's still accepted"

Risk 2: Backward compatibility

Old systems that only understand SHA-1 won't verify SHA-256 signatures

- Must maintain SHA-1 signatures for legacy systems until they're upgraded

Risk 3: Certificate chain issues

Code signing certificates might support both algorithms or need replacement

- Dual-key certificates (one for SHA-1, one for SHA-256) might be needed

Risk 4: SHA-1 weakening over time

As SHA-1 attacks improve, the risk increases throughout the 10-year period

Step 2: Quantitative risk assessment

SHA-1 collision attacks in 2017 cost ≈ $100,000 (110 GPU-years)

By 2024, cost has dropped significantly

In 10 years, SHA-1 collisions might be trivial

Probability of attack over 10 years: Very high if SHA-1 signatures remain accepted

Step 3: Migration strategy

Phase 1: Preparation (Month 1-3)

- Inventory all systems that verify signatures

- Test SHA-256 verification on all platforms

- Prepare dual-signing infrastructure

Phase 2: Dual signing (Month 4-24)

- Sign all new files with both SHA-1 and SHA-256

- Include both signatures in file metadata or separate signature files

- Update documentation to prefer SHA-256 verification

Phase 3: Verification update (Month 6-30)

- Update all verification systems to accept SHA-256

- Configure them to prefer SHA-256 over SHA-1

- Keep SHA-1 acceptance for backward compatibility

Phase 4: SHA-1 deprecation (Month 24-36)

- Stop issuing SHA-1 signatures for new files

- Update verification systems to warn on SHA-1 only

- Begin planning for SHA-1 removal

Phase 5: SHA-1 removal (Month 36-48)

- After confirming all systems updated, remove SHA-1 acceptance

- Old files with only SHA-1 signatures become unverifiable (need re-signing)

Step 4: Handling existing 10,000 files

Option A: Re-sign all with SHA-256 (requires access to original signing keys)

- Best security, but operational overhead

Option B: Provide dual signatures alongside originals

- Distribute .sig256 files alongside original signed files

- Verification systems check for SHA-256 signature first, fall back to SHA-1

Option C: Timestamp-based acceptance

- Accept SHA-1 signatures only for files timestamped before deprecation date

- Requires trusted timestamp service

Step 5: Recommendation

For maximum security:

1. Re-sign critical files with SHA-256 immediately

2. Implement dual signing for all new files

3. Set a firm deprecation date for SHA-1 (e.g., 2 years out)

4. Communicate clearly to all users about the timeline

5. Monitor for any SHA-1 collision attempts during transition

Answer: The main risks are attackers exploiting SHA-1 collisions during the transition window and legacy systems losing verification capability. A phased approach over 2-4 years, with dual signing and clear deprecation timelines, minimizes these risks while ensuring backward compatibility.

Problem 2.6: Performance Comparison

Compare the performance of SHA-256 and SHA-3-256 on a 64-bit processor. SHA-256 uses 32-bit operations, while SHA-3 uses 64-bit operations. If SHA-256 processes 64 bytes per cycle and SHA-3 processes 48 bytes per cycle, which is faster for a 1 GB file? What about on a 32-bit embedded system?

Solution

Step 1: Performance on 64-bit processor

SHA-256: 64 bytes/cycle

SHA-3-256: 48 bytes/cycle

For 1 GB = 1,073,741,824 bytes:

SHA-256 cycles needed = 1,073,741,824 / 64 = 16,777,216 cycles

SHA-3 cycles needed = 1,073,741,824 / 48 = 22,369,621 cycles

Ratio = 22,369,621 / 16,777,216 = 1.33

SHA-256 is about 33% faster on 64-bit processors.

Step 2: Why the difference

SHA-256's 32-bit operations are less efficient on 64-bit processors (they use only part of the ALU width)

SHA-3's 64-bit operations match the processor's native word size

Despite this, SHA-256's simpler round function and better optimization give it the edge

Step 3: Performance on 32-bit embedded system

On 32-bit processors, SHA-3's 64-bit operations must be emulated using multiple 32-bit operations, significantly slowing it down.SHA-256: Still 32-bit operations, efficient

SHA-3: Each 64-bit operation requires at least 2-3 32-bit operations

Assume SHA-3 effective throughput drops to 20 bytes/cycle on 32-bit

SHA-256 cycles = 16,777,216 (same as before)

SHA-3 cycles = 1,073,741,824 / 20 = 53,687,091 cycles

Ratio = 53,687,091 / 16,777,216 = 3.2

SHA-256 is about 3.2 times faster on 32-bit systems.

Step 4: Additional factors

- Power consumption: More cycles = more power, critical for battery-powered devices

- Memory usage: SHA-3's larger state (1600 bits vs 512 bits) uses more memory

- Hardware acceleration: Some processors have SHA-256 hardware instructions, making it even faster

Step 5: Practical implications

- For servers (64-bit): SHA-256 is faster, so it remains preferred

- For IoT/embedded (32-bit): SHA-256 is significantly faster

- For new designs: Some choose SHA-3 for future-proofing despite performance penalty

Answer: On 64-bit systems, SHA-256 is about 33% faster than SHA-3 for a 1GB file. On 32-bit embedded systems, SHA-256 is about 3.2 times faster. This performance advantage, plus hardware acceleration in modern CPUs, explains why SHA-256 remains more widely used than SHA-3 despite SHA-3's newer design.

Chapter 3

The Merkle-Damgård Construction

3.1 Understanding the Construction

The **Merkle–Damgård construction** is a foundational framework used to build cryptographic hash functions that can securely process messages of **arbitrary length** while producing a **fixed-length output**. It was independently developed by Ralph Merkle and Ivan Damgård. Rather than hashing an entire message at once, this construction breaks the message into fixed-size blocks and processes them sequentially using a compression function and a chaining mechanism.

The process begins by padding the message so its length becomes an exact multiple of the block size. The padded message is then divided into blocks
$M_1, M_2, \dots, M_n$.
An initial value H_0 (called the Initialization Vector) is defined by the algorithm. Each block is processed iteratively using a compression function f:

$H_i = f(H_{i-1}, M_i)$ for i = 1 to n

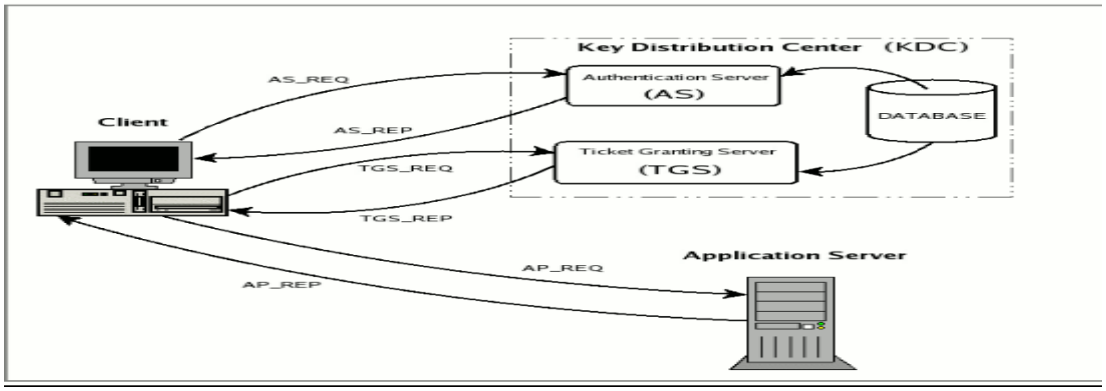
Here, H_i is the intermediate chaining value after processing block M_i. Each new state depends on both the previous state and the current message block, effectively "chaining" all blocks together. After the final block is processed, the last chaining value H_n becomes the hash output. This structure allows very long messages to be handled efficiently while preserving security properties.

A key theoretical result of the Merkle–Damgård construction is that **if the compression function is collision-resistant, then the full hash function is also collision-resistant**. This makes the design modular: cryptographers can focus on building a strong compression function and then safely extend it to arbitrary-length inputs. This principle has been used in many classic hash algorithms, including MD5, SHA-1, and SHA-2.

However, Merkle–Damgård also introduces certain structural characteristics. Because the internal chaining value after the final block is directly exposed as the hash, these designs are

vulnerable to **length-extension attacks**: an attacker who knows Hash(M) can compute Hash(M ‖ X) for some extension X without knowing M. This is not a flaw in the compression function itself, but a consequence of the overall construction. As a result, modern systems often use protective techniques (such as HMAC) or alternative designs like sponge constructions to avoid this issue.

Despite these limitations, Merkle–Damgård remains historically and practically important. It established the standard "iterate-a-compression-function" approach to hashing and influenced decades of cryptographic design. Its simplicity, efficiency, and provable properties made it the dominant paradigm until newer constructions (such as SHA-3's sponge model) were introduced to address its structural weaknesses.



The Core Concept

The core idea behind the Merkle–Damgård construction is to transform a **fixed-input cryptographic compression function** into a **full hash function capable of handling messages of arbitrary length**. Instead of trying to hash an entire message at once, the construction processes the message **incrementally**, block by block, while maintaining a running internal state called the *chaining value*. This design makes hashing scalable, efficient, and mathematically analyzable.

At a conceptual level, the message is first padded and split into equal-sized blocks:

$M_1, M_2, \ldots, M_n$

The algorithm starts with a predefined initial value H_0 (Initialization Vector). Each message block is then combined with the current chaining value using a compression function f:

$H_i = f(H_{i-1}, M_i)$

This equation captures the heart of the construction. Every new state H_i depends on **all previous blocks**, because H_{i-1} already contains the accumulated influence of M_1 through M_{i-1}. After the final block M_n is processed, the last chaining value H_n becomes the hash output. In effect, the entire message is "folded" into a single fixed-length digest through repeated application of the same compression function.

The brilliance of this approach lies in its **iterative chaining**. Each block modifies the internal state, and that modified state becomes the input for the next block. This creates strong *diffusion*: even a one-bit change early in the message propagates through every subsequent round, producing a completely different final hash. This mechanism is responsible for the avalanche effect seen in cryptographic hash functions.

Another important aspect of the core concept is its **security inheritance property**. The construction guarantees that if the underlying compression function is collision-resistant, then the overall hash function is also collision-resistant. This theoretical result, developed independently by Ralph Merkle and Ivan Damgård, allowed designers to focus on building a strong compression primitive and safely extend it to arbitrary-length inputs.

However, this same structure also exposes a fundamental characteristic: the final chaining value is directly revealed as the hash output. Because of this, Merkle–Damgård hashes are vulnerable to *length-extension attacks*, where an attacker can extend a message and compute a valid hash without knowing the original input. This does not break the compression function itself, but arises from the chaining-based design.

In essence, the core concept of Merkle–Damgård is **recursive state updating**: each message block updates a shared internal state, and the final state represents the cryptographic fingerprint of the entire message. This simple yet powerful idea shaped classic hash functions such as MD5, SHA-1, and SHA-2, and dominated hash design for decades before newer paradigms like sponge constructions (used in SHA-3) were introduced.

3.2 Detailed Processing Steps

Padding

Before any hashing computation begins, the input message must be prepared so that its total length fits exactly into the fixed-size blocks expected by the compression function. Since messages can be of arbitrary length, a **padding scheme** is required to make the message length a precise multiple of the block size (for example, 512 or 1024 bits, depending on the algorithm). This preprocessing step is not just a formatting convenience—it is a **critical security component** of the hash function design.

The standard padding method used in Merkle–Damgård–based hash functions follows three structured steps. First, a single **'1' bit** is appended to the end of the original message. This guarantees that the padded message is always different from the original, even if the original already aligned with the block boundary. Next, a sequence of **'0' bits** is added until the message length reaches a specific position within the final block (typically leaving room for the length field). Finally, a **fixed-size binary representation of the original message length** is appended. This length field is usually 64 bits (or 128 bits in larger variants) and records the size of the unpadded message.

Including the original message length is essential because it makes the padding **reversible and unambiguous**. Given a padded message, the hash function can always determine where the real message ended and where padding began. More importantly, this step prevents certain structural attacks by ensuring that two different messages—even if one is an extension of the other—cannot result in the same padded input. Without the length field, attackers could exploit ambiguity in padding to engineer collisions or manipulate internal states.

This padding strategy also guarantees **domain separation**: every possible input maps to a unique padded representation. Even if two messages differ by only a single bit, their padded forms will differ significantly, especially once the length encoding is added. This uniqueness is a foundational requirement for collision resistance, because hash functions rely on the assumption that distinct inputs lead to distinct internal processing paths.

From a cryptographic perspective, MD padding supports the avalanche effect and preserves the integrity of the compression-chain mechanism. Each padded block influences the internal state, and the final length field ensures that the hash reflects not just the content of the message, but also its exact size. This prevents attackers from appending extra data to an already-hashed

message while preserving the original digest—a class of weaknesses that padding is specifically designed to avoid or mitigate.

In summary, MD padding is far more than a technical detail. By appending a '1' bit, filling with zeros, and encoding the original message length, the scheme guarantees **unique block alignment, reversibility, and resistance to ambiguous representations**. These properties are fundamental to maintaining collision resistance and overall hash security, making padding a cornerstone of modern cryptographic hash function design.

Initialization Vector

In Merkle–Damgård–based hash functions (such as SHA-1 and SHA-2), the hashing process does not start from an empty or random state. Instead, it begins with an **Initialization Vector (IV)**—a predefined set of fixed constants that forms the *initial internal state* of the algorithm. This IV acts as the **starting chaining value** for the first message block and is explicitly specified in the hash standard. Every message, regardless of its content or length, begins processing from this same IV.

The purpose of using a fixed IV is to ensure **consistency and determinism**: the same input message will always produce the same hash output on any compliant implementation. Without a standardized IV, different systems could produce different hashes for identical data, breaking interoperability. These constants are not chosen arbitrarily; they are carefully derived (often from fractional parts of mathematical constants or primes) to avoid hidden structure or bias. This transparent selection process helps prevent suspicion of intentional weaknesses or "backdoors."

Once hashing begins, the IV is combined with the first padded message block through the compression function, producing a new internal state. This output then becomes the chaining value for the next block, and so on, until all blocks are processed. In effect, the IV seeds the entire computation, and every subsequent transformation depends indirectly on it. Although the IV itself is public and identical for all users, security does not rely on keeping it secret—the strength comes from the nonlinear compression rounds and the overall hash structure.

Using a fixed IV also simplifies security analysis. Because the starting state is known and constant, cryptographers can rigorously evaluate resistance to collisions and pre-image attacks

under well-defined conditions. This predictability is essential for formal proofs and for standardization across platforms. Organizations such as the National Institute of Standards and Technology define these IV values as part of the official algorithm specification to ensure global compatibility.

It is important to distinguish this concept from IVs used in encryption modes (which are often random or unique per message). In hash functions, the IV is **not meant to provide randomness**; instead, it establishes a common, trusted baseline for computation. Any variation in security comes from the message content and the internal mixing operations, not from the IV.

In summary, the Initialization Vector serves as the **cryptographic anchor point** of the hashing process. By providing a fixed, well-defined initial state, it guarantees deterministic behavior, enables consistent implementations, and supports robust security analysis—making it a foundational component of modern hash function design.

Iterative Compression

After padding, the message is divided into **equal-sized blocks**, denoted as

$M_1, M_2, M_3, \ldots, M_n$

Each block has a fixed length (for example, 512 bits in SHA-256). The hash computation proceeds iteratively using a **compression function**, usually written as:

$f( )$

The process starts with an **initial chaining value**, called the Initialization Vector (IV), denoted as:

$H_0 = IV$

For each message block M_i, the compression function takes two inputs:

- The previous chaining value H_{i-1}
- The current message block M_i

and produces a new chaining value:

$H_i = f(H_{i-1}, M_i)$

This operation is performed sequentially for every block:

$H_1 = f(H_0, M_1)$
$H_2 = f(H_1, M_2)$
$H_3 = f(H_2, M_3)$
…
$H_n = f(H_{n-1}, M_n)$

Each H_i represents the **intermediate hash state** after processing the i-th block. This value carries forward all information from previous blocks, which is why it is called a *chaining value*. Any small change in any message block alters H_i and propagates through all subsequent computations, producing a completely different final hash. This mechanism ensures the avalanche effect and strong diffusion.

The compression function f itself performs multiple rounds of logical operations, rotations, modular additions, and mixing steps to combine H_{i-1} and M_i in a nonlinear way. Its purpose is to irreversibly blend the message data into the internal state while preventing attackers from reconstructing previous states or message blocks.

After the final block M_n is processed, the resulting chaining value H_n becomes the **final hash output**:

$\text{Hash}(\text{message}) = H_n$

This iterative structure allows hash functions to handle messages of arbitrary length while producing a fixed-length digest. Security depends on the compression function being collision-resistant and one-way, because breaking any round would compromise the entire chain.

In summary, the hash computation follows a block-by-block feedback process where each block updates the internal state through:

$H_i = f(H_{i-1}, M_i)$

and the last state H_n represents the cryptographic fingerprint of the entire message.

3.3 Security Properties

The Merkle-Damgård construction has an important security property: if the compression function is collision-resistant, then the overall hash function is collision-resistant. This means cryptanalysts only need to analyze the compression function's security rather than the entire hash function.

However, the construction has limitations. Most notably, it's vulnerable to length-extension attacks. If an attacker knows H(m) for some unknown message m, they can compute H(m || padding || extra) without knowing m. This vulnerability led to the development of HMAC and influenced the design of SHA-3's sponge construction.

Practice Problems Set 3: Merkle-Damgård Construction

Problem 3.1: Padding Calculation

For a message of length 1000 bits, using a Merkle-Damgård hash with 512-bit blocks and 64-bit length field, calculate:

a) How many padding bits are needed?

b) What is the final padded message length?

c) How many blocks will be processed?

Solution

Step 1: Understand the padding requirements

For Merkle-Damgård with 512-bit blocks and 64-bit length field:

- Need to append a '1' bit

- Then append enough '0' bits so that total length ≡ 448 mod 512

- Finally append 64-bit length representation

Why 448? Because 448 + 64 = 512, so after appending length, we get a full block.

Step 2: Calculate current length

Original message length L = 1000 bits

Step 3: Find required padding

We need: (L + 1 + padding_zeros) mod 512 = 448

First, add the mandatory '1' bit:

1000 + 1 = 1001 bits

Now find k such that (1001 + k) mod 512 = 448

1001 mod 512 = 1001 - 512 = 489

So 1001 ≡ 489 mod 512

We need 489 + k ≡ 448 mod 512

Therefore k ≡ 448 - 489 ≡ -41 mod 512

k ≡ 512 - 41 = 471 mod 512

Since k must be between 0 and 511, k = 471

Step 4: Verify

After '1' bit: 1001 bits

After 471 zeros: 1001 + 471 = 1472 bits

1472 mod 512 = 1472 - (512 × 2) = 1472 - 1024 = 448 ✓

Step 5: Add length field

Add 64 bits for length = 1000 (original length)

Total padded length = 1472 + 64 = 1536 bits

Step 6: Number of blocks

1536 bits / 512 bits per block = 3 blocks

Answer:

a) Padding bits = 1 + 471 = 472 bits (1 '1' bit + 471 '0' bits)

b) Final padded length = 1536 bits

c) Number of blocks = 3

Problem 3.2: Length Extension Attack Demonstration

Consider a hash function H based on Merkle-Damgård with 8-bit blocks and 4-bit length field (simplified). The compression function is: f(H, M) = (H + M) mod 256. IV = 0.

Given H("Pay $100") = 42, show how an attacker can compute H("Pay $100" || padding || "Extra $900") without knowing the original message. Assume "Pay $100" is exactly one block.

Solution

Step 1: Understand the simplified hash function

- Block size: 8 bits (1 byte)

- Length field: 4 bits (so max message length 15 bits)

- Compression: f(H, M) = (H + M) mod 256

- IV = 0

Step 2: What the attacker knows

The attacker knows that H("Pay $100") = 42, but doesn't know the actual message content.

They also know the padding scheme:

- Append '1' bit

- Append zeros until length ≡ (block_size - length_field_size) mod block_size

- Append 4-bit length

Block_size = 8 bits, length_field = 4 bits, so need length ≡ 4 mod 8 before length field.

Step 3: Reconstruct the internal state after original message

For a Merkle-Damgård hash, the final hash value is the chaining value after processing the last block.

Since the attacker knows H = 42, and assuming the original message was exactly one block (8 bits), the chaining value after processing that block is 42.

This means: f(IV, M) = 42

f(0, M) = (0 + M) mod 256 = 42

Therefore M = 42

So the attacker actually recovers the original message block! (This is because our toy hash is reversible—real hash functions aren't.)

Step 4: But even if M weren't recoverable

The key insight: The attacker can use the final hash value (42) as the new IV for continuing the hash computation.

They want to compute H(original || padding || extra)

They know:

- After processing original message, state = 42

- They know the padding that would be added (they know the length of original message? Not necessarily, but they can guess or try possibilities)

Step 5: Compute padding for original message

Original message "Pay $100" is one block (8 bits), so length = 8 bits.

Padding for original (before attacker adds extra):

- Need to append '1' bit

- Need enough zeros to reach 4 mod 8 (since 4 bits reserved for length)

- After '1' bit: 8 + 1 = 9 bits

- Need to reach length ≡ 4 mod 8: next number ≡ 4 mod 8 after 9 is 12

- So need 12 - 9 = 3 zeros

- Then append 4-bit length = 8 (binary 1000)

So padding = 1,0,0,0, then length 1000

Step 6: Compute extended hashStarting from state = 42, process the padding block first:

Padding block = [1,0,0,0,1,0,0,0] (8 bits: the 1, three zeros, then 4-bit length 1000)

Wait—this is 8 bits? 1 + 3 zeros = 4 bits, plus 4-bit length = 8 bits. Yes, exactly one block.

So new state after padding = f(42, padding) = (42 + value_of_padding) mod 256

Value of padding block as integer: binary 10001000 = 128 + 8 = 136

New state = (42 + 136) mod 256 = 178

Step 7: Process extra message

Now add "Extra $900" as another block. Assume this block value = 200.

Final state = f(178, 200) = (178 + 200) mod 256 = 378 mod 256 = 122

Step 8: The forged hash

The attacker has computed H(original || padding || extra) = 122, without knowing the original message content, only knowing its hash (42) and the padding scheme.

Answer: The attacker extends the hash from 42 to 122 by using the original hash as the new IV and continuing the chain. This demonstrates why length extension attacks are dangerous and why constructions like HMAC are needed.

Problem 3.3: Fixed Points in Compression Functions

In a Merkle-Damgård hash, if the compression function has a fixed point (i.e., exists H, M such that f(H, M) = H), what security implications does this have?

Solution

Step 1: Understanding fixed points

A fixed point means there exists some chaining value H and message block M such that:

f(H, M) = H

In other words, processing block M leaves the chaining value unchanged.

Step 2: Implications for collision resistance

If such a fixed point exists, an attacker can create collisions by inserting or removing the fixed point block.

For example, consider two messages:

Message A: [M1, M2, ..., Mn]

Message B: [M1, M2, ..., Mk, M_fixed, M_{k+1}, ..., Mn]

If M_fixed is a fixed point block for the chaining value after Mk, then the hash of both messages will be identical.

This is because:

H after Mk → process M_fixed → same H → process rest identically

Step 3: Practical example

Suppose f(H, M) = H ⊕ M (XOR), which has many fixed points.

When does H ⊕ M = H? Only when M = 0.

So the all-zero block is a fixed point for all H. This means inserting any number of zero blocks doesn't change the hash—a serious vulnerability.

Step 4: Real-world relevance

Real hash functions are designed to avoid fixed points, but they've been found in some:

- MD4 had fixed points discovered

- SHA-1's compression function may have fixed points, though finding them is hard

Step 5: Attack scenario

If an attacker finds a fixed point block for a particular chaining value:

1. They can create multiple messages with the same hash

2. This could be used to create fraudulent documents

3. In MAC constructions, fixed points might enable forgeries

Step 6: Mitigation

Modern hash functions are designed with:

- Complex round functions that make fixed points extremely unlikely

- Different constants in each round to prevent symmetry

- Careful analysis to ensure no trivial fixed points exist

Answer: Fixed points in the compression function allow attackers to insert or remove blocks without changing the hash, creating collisions. This violates collision resistance and can enable various attacks. Good hash function designs ensure fixed points are computationally infeasible to find.

Problem 3.4: Multi-Block Length Extension

A server uses H(k || message) as a MAC, where H is a Merkle-Damgård hash with 512-bit blocks, k is a 128-bit key, and messages can be multiple blocks. Show how an attacker can forge a valid MAC for message' = message || padding || extra without knowing k, even if they don't know the original message length.

Solution

Step 1: Understanding the scenario

MAC = H(k || message)

- k is 128 bits (16 bytes) prepended to message

- H uses 512-bit (64-byte) blocks

- Attacker knows MAC for some message, doesn't know k

- Attacker wants MAC for message || padding || extra

Step 2: The vulnerability

Because of length extension, the final hash after processing k || message is the internal state. The attacker can use this state as IV to continue hashing.

However, the attacker doesn't know the exact length of k || message, so they don't know exactly what padding was added before the final hash.

Step 3: Determining original length

The attacker can see the message (if it's known) but not k. They know:

- k is 128 bits

- Message length = L bits

Total processed so far = 128 + L bits

The padding added by the original hash computation depends on this total.

Step 4: Guessing the padding

The attacker needs to determine what padding was added to reach a block boundary.

Let total = 128 + L

Blocks processed = ceil(total / 512)

The padding added originally was:

- '1' bit

- Zeros to make (total + 1 + zeros) ≡ 448 mod 512

- 64-bit length field = total

The attacker can compute this padding because they know total (128 + L).

Step 5: The forgery process

To compute MAC(message || padding || extra):

1. Take the original MAC value (which is the internal state after processing k || message || padding_original)

2. Use this as new IV

3. Process the extra message blocks

But careful: The attacker wants message || padding_original || extra? No, they want message || padding_new || extra, where padding_new is the padding that would be added to the extended message.

Actually, the length extension attack works as:

Original: H(k || message)

After processing, internal state = S (which is the MAC)

To compute H(k || message || padding_original || extra):

Start from state S (which already includes the original padding)

Process extra blocks

This yields H(k || message || padding_original || extra)

Step 6: Why this works

The original hash computation already added padding to make k || message a multiple of block size plus room for length.

By continuing from that state, we're effectively hashing:

k || message || padding_original || extra

But note: This means the final message has two paddings—the original one and then extra data. The final hash will include its own padding at the end.

Step 7: Practical example

Suppose:

- k = 128 bits

- message = 300 bits

- Total = 428 bits

Original padding:

428 + 1 = 429

Need 429 + zeros ≡ 448 mod 512 → zeros = 19

Then append 64-bit length = 428

Total padded = 428 + 1 + 19 + 64 = 512 bits exactly (1 block)

So original processed one block, final state = MAC

To extend:

Attacker takes that state as IV

Adds extra message (say 100 bits)

Processes that, then adds final padding

Result is MAC for: k || message || original_padding || extra

Answer: The attacker can forge a valid MAC by using the original MAC value as the starting state and continuing the hash with extra data. This works because the original computation already included the padding, and the attacker doesn't need to know k—the internal state captures all key-dependent information. This is why H(k || message) is insecure as a MAC and why HMAC was designed.

Problem 3.5: SHA-3's Resistance to Length Extension

Explain why SHA-3's sponge construction is inherently resistant to length extension attacks, even though it also processes data in blocks.

Solution

Step 1: Recall Merkle-Damgård vulnerability

In Merkle-Damgård:

- Final hash = internal state after last block

- This state contains all information needed to continue hashing

- Attacker can use hash as IV to extend

Step 2: SHA-3's sponge construction

SHA-3 has:

- Internal state of size b = r + c (rate + capacity)

- During absorption, input blocks are XORed with the rate part

- The full state (rate + capacity) is permuted

- During squeezing, output is taken from rate part

Step 3: Why length extension fails

After finalizing the hash, SHA-3's internal state is not fully output:

1. Only the rate part (r bits) is output as hash

2. The capacity part (c bits) remains hidden

For SHA3-256:

- r = 1088 bits, c = 512 bits

- Output is 256 bits (truncated from rate)

Step 4: Attacker's knowledge

An attacker knows only the 256-bit hash, not the full 1600-bit state.

To continue hashing, they would need:

- The full state (including capacity bits)

- But capacity bits are never output and cannot be derived from the hash

Step 5: Attempting extension

If attacker tries to use the hash as new IV:

- They have only 256 bits

- Need to initialize 1600-bit state

- They could set rate part to known bits, but capacity is unknown

- Any guess at capacity will be wrong with overwhelming probability

Step 6: Mathematical guarantee

The security proof shows that distinguishing the sponge from random requires at least 2^(c/2) operations.

For SHA3-256 with c=512, this is 2^{256} operations—computationally impossible.

Even if attacker could guess capacity, after permutation the state becomes unpredictable.

Step 7: Analogy

Think of it like a locked safe:

- Merkle-Damgård: Safe door is open, you can see everything inside

- Sponge: Safe door is closed, you only see a small window (rate part)

- To continue, you need to know what's in the locked part (capacity)

Answer: SHA-3 resists length extension because only part of the internal state (the rate) is output as the hash. The capacity portion remains secret, and without it, an attacker cannot correctly continue the hash computation. Even if they try to guess, the probability of success is 2^(-c), which is astronomically small (2^{-512} for SHA3-256).

Chapter 4

Security of Hash Functions and the Need for Authentication

4.1 The Authentication Gap

The *authentication gap* refers to a fundamental weakness that arises when cryptographic hash functions are used alone to provide message integrity without proper authentication. While standard hash functions can detect accidental data changes, they do **not** inherently verify *who* created the message. This creates a security gap between *integrity* (detecting modification) and *authenticity* (verifying the sender). In practical systems, this gap can be exploited by attackers who are able to modify messages and recompute hashes, making altered data appear legitimate.

At the core of this problem is the fact that traditional hash functions are **public and deterministic**. Anyone can compute Hash(M) for any message M. If a system relies solely on a plain hash value to validate data, an adversary who intercepts the message can replace it with a modified version M′ and simply recompute Hash(M′). Because no secret is involved, the receiver has no way to distinguish between a genuine sender and an attacker. Thus, although hashes provide strong integrity checking, they offer **no built-in authentication**.

This weakness becomes particularly serious in Merkle–Damgård–based hash functions, where structural properties such as *length-extension attacks* further widen the authentication gap. In such cases, an attacker who knows Hash(M) can compute Hash(M ‖ X) for additional data X without knowing M. This allows adversaries to append malicious content while preserving a valid-looking hash, completely bypassing naïve integrity checks. The vulnerability does not lie in the compression function itself, but in how hashes are used without a secret component.

The authentication gap highlights an important principle in cryptography: **hashing alone is not authentication**. To close this gap, hash functions must be combined with secret keys using constructions such as Message Authentication Codes (MACs), most commonly HMAC. These mechanisms ensure that only parties possessing the shared secret key can generate valid authentication tags. Standards bodies such as the National Institute of Standards and Technology explicitly recommend keyed constructions for secure message authentication rather than raw hashes.

From a system-design perspective, the authentication gap has historically led to many real-world vulnerabilities, especially in early web protocols, APIs, and embedded systems. Developers often assumed that adding a hash to a message was sufficient for security, overlooking the fact that attackers could reproduce the same operation. Modern cryptographic practice emphasizes authenticated hashing precisely to eliminate this gap.

In summary, the authentication gap represents the disconnect between **data integrity and sender verification** when plain hash functions are used improperly. While hashes are excellent tools for detecting changes, they cannot prove origin or intent. Closing this gap requires incorporating secret keys and authenticated constructions, ensuring that integrity checks also provide strong assurance of authenticity.

Integrity vs. Authentication

Integrity and **authentication** are closely related security concepts, but they serve fundamentally different purposes in cryptographic systems. *Integrity* ensures that data has not been altered during storage or transmission. In other words, it answers the question: **"Has this message changed?"** Cryptographic hash functions are commonly used to provide integrity by producing a fixed-length digest of the data. If even a single bit of the message changes, the hash changes dramatically, allowing the receiver to detect tampering.

However, integrity alone does **not** establish *who* created the message. This is where **authentication** comes in. Authentication verifies the **identity of the sender** and confirms that the message genuinely originated from an authorized source. It answers the question: **"Who sent this message?"** Authentication always requires a secret (such as a shared key or private key). Without a secret, anyone—including an attacker—can generate valid hashes for modified messages, making plain hashing insufficient for security-critical communication.

This distinction reveals a common misconception: **a hash provides integrity, but not authentication**. Because hash functions are public and deterministic, an adversary who intercepts a message can modify it and simply recompute the hash. The receiver, seeing a matching hash, cannot tell whether the message came from the legitimate sender or from an

attacker. Thus, while integrity checks can detect accidental corruption, they cannot prevent intentional forgery.

To achieve true authentication, cryptographic systems use **keyed constructions**, such as Message Authentication Codes (MACs) or digital signatures. In MAC-based systems, only parties possessing the shared secret key can produce a valid authentication tag. This closes the *authentication gap* by binding message integrity to a secret. Standards bodies like the National Institute of Standards and Technology explicitly recommend using MACs (for example, HMAC) rather than raw hash functions for message authentication.

In practical terms, the difference can be summarized simply:

- **Integrity** detects whether data has been modified.
- **Authentication** verifies who created the data.

A secure system typically requires **both**. For example, secure network protocols, APIs, and file verification systems rely on authenticated hashing or digital signatures to ensure that messages are not only unchanged, but also originate from trusted sources.

In summary, integrity protects data from unnoticed modification, while authentication protects systems from impersonation and forgery. Using integrity without authentication leaves systems vulnerable to active attacks, whereas combining both provides strong assurance of data correctness *and* sender legitimacy—an essential requirement for modern cryptographic security.

4.2 Message Authentication Codes (MACs)

A Message Authentication Code solves this problem by introducing a secret key into the computation. Only parties who know the key can generate valid MAC tags for messages.

Definition and Properties

A MAC algorithm takes two inputs:

- A message of arbitrary length

- A secret key known only to the communicating parties

It produces a fixed-size output called the MAC tag. The security requirement is existential unforgeability under chosen-message attack—an adversary who doesn't know the key should be unable to generate a valid MAC tag for any message, even if they've seen many valid message-tag pairs for other messages.

How MACs Provide Authentication

When Alice wants to send an authenticated message to Bob:

1. They share a secret key K in advance through some secure mechanism

2. Alice computes MAC(K, message) and sends both message and tag to Bob

3. Bob computes MAC(K, message) himself and compares with the received tag

4. If they match, Bob knows the message came from Alice (because only she knows K) and hasn't been modified (because any modification would produce a different tag)

When Alice wants to send an authenticated message to Bob, they first establish a shared secret key **K** using some secure key-exchange mechanism. This key is known only to Alice and Bob and forms the foundation of trust between them. Alice then applies a Message Authentication Code (MAC) algorithm to the message using this secret key, computing a short authentication tag denoted as **MAC(K, message)**. She sends both the original message and this tag to Bob over the communication channel. Upon receiving them, Bob independently runs the same MAC algorithm on the received message using the same secret key **K**, producing his own version of the authentication tag. He then compares this locally computed tag with the tag sent by Alice. If the two values match, Bob gains two important assurances: first, the message must have originated from Alice, because only someone who knows **K** could have generated the correct MAC (authentication); second, the message must not have been altered during transmission, because even a tiny modification would result in a completely different tag (integrity). If the tags do not match, Bob immediately rejects the message, knowing that it has either been tampered with or sent by an unauthorized party. This simple yet powerful mechanism binds message content to a secret key, closing the authentication gap that exists when plain hash functions are used alone.

4.3 MAC vs. Digital Signatures

Message Authentication Codes (MACs) and Digital Signatures are both cryptographic mechanisms used to ensure **message integrity and authentication**, but they differ

fundamentally in how trust is established and managed. A MAC is based on **symmetric cryptography**, meaning both the sender and receiver share the same secret key. When a sender computes MAC(K, M) for a message M using a shared key K, the receiver verifies it by recomputing the MAC with the same key. If the values match, the message is accepted as authentic and unmodified. This approach is highly efficient and well-suited for closed systems or environments where secure key sharing is practical, such as internal networks or client–server applications.

However, MACs have an important limitation: **they do not provide non-repudiation**. Because both parties possess the same secret key, either party could have generated the MAC. As a result, neither side can later prove to a third party who actually created the message. This makes MACs unsuitable for legal or audit scenarios where proof of origin is required. MACs also require a secure method for distributing and storing shared keys, which becomes increasingly complex as the number of participants grows.

Digital signatures, in contrast, rely on **asymmetric cryptography**. Each user has a private key (kept secret) and a corresponding public key (shared openly). The sender signs a message using their private key, and anyone can verify the signature using the sender's public key. This means only the owner of the private key could have created the signature, providing not only integrity and authentication, but also **non-repudiation**—the sender cannot later deny having signed the message. This property makes digital signatures essential for contracts, software distribution, certificates, and legal communications.

Another key difference lies in scalability and trust models. MACs work best in systems with a limited number of trusted participants, because every communicating pair must share a unique secret key. Digital signatures scale much more naturally to large, open systems such as the internet, where public keys can be distributed through certificates and trusted infrastructures. Standards bodies like the National Institute of Standards and Technology recommend MACs (such as HMAC) for fast symmetric authentication and digital signatures for applications requiring public verification and accountability.

Performance is also a factor. MACs are computationally lightweight and faster than digital signatures, making them ideal for high-throughput environments. Digital signatures are more computationally expensive, but they offer stronger guarantees about identity and

accountability. In practice, many secure systems use **both**: digital signatures for establishing identity and exchanging keys, and MACs for efficiently protecting ongoing communication.

In summary, MACs provide **integrity and authentication using shared secrets**, but lack non-repudiation, while digital signatures provide **integrity, authentication, and non-repudiation using public-key cryptography**. MACs are best for private, performance-critical systems, whereas digital signatures are essential for open, distributed environments where proof of origin and legal trust are required.

This difference has important implications:

- MACs are faster and more efficient than digital signatures

- MACs require the communicating parties to share a secret key in advance

- MACs don't provide non-repudiation—Bob can't prove to a third party that Alice sent the message because Bob could have generated the tag himself

- Digital signatures provide non-repudiation—only Alice possesses her private key, so she can't deny signing

Practice Problems Set 4: MAC Fundamentals

Problem 4.1: MAC Security Definition

Explain what "existential unforgeability under chosen-message attack" (EUF-CMA) means for a MAC. Provide an example of an attack that would break this property.

Solution

Step 1: Break down the terminology

Existential unforgeability:

- "Existential" means the attacker can forge a MAC for any message, not necessarily a specific one they chose in advance

- They succeed if they can produce any valid (message, tag) pair that wasn't previously produced by the legitimate sender

Under chosen-message attack:

- The attacker can request MACs for messages of their choice

- They can adaptively choose new messages based on previous responses

- This models a real-world scenario where an attacker might trick the system into signing some messages

Step 2: Formal definition

A MAC scheme is EUF-CMA secure if no efficient adversary can win the following game with non-negligible probability:

1. Challenger generates a random key K (unknown to adversary)

2. Adversary can adaptively query a MAC oracle: for any message m, oracle returns MAC(K, m)

3. Adversary wins if they can output any (m*, tag*) such that:

  - m* was never queried to the oracle

  - tag* = MAC(K, m*) verifies correctly

Step 3: Example of breaking EUF-CMA

Consider a broken MAC scheme: MAC(K, m) = K $\oplus$ m (XOR) for fixed-length messages.

Attack:

1. Adversary queries MAC for m1 = 0x0000, gets tag1 = K $\oplus$ 0 = K

2. Adversary now knows K!

3. Adversary can forge MAC for any m*: tag* = K $\oplus$ m*

4. This breaks existential unforgeability (can forge any message) under chosen-message attack (only needed one query)

Step 4: Why this matters

Real-world impact:

- If an attacker can forge MACs, they can impersonate legitimate users

- They could modify messages in transit and generate valid tags

- Financial transactions, API calls, authentication tokens could all be forged

Step 5: Stronger notions

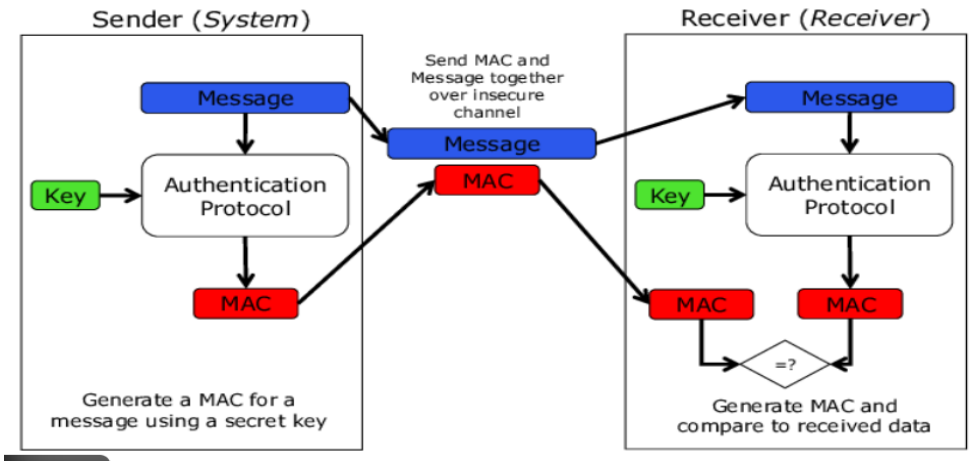
Some applications require "strong unforgeability":

- Attacker shouldn't be able to produce a different tag for a previously queried message

- Prevents tag substitution attacks

Answer: EUF-CMA means an attacker cannot create a valid MAC for any new message, even after seeing MACs for many chosen messages. The XOR example shows a broken scheme where one query reveals the key, allowing arbitrary forgeries.

Problem 4.2: MAC vs Hash Authentication



A developer proposes using H(message || key) as a MAC. Explain why this is insecure, even though it uses a secret key. Provide a concrete attack scenario.

Solution

Step 1: The proposed scheme

MAC(K, m) = H(m || K)

This seems reasonable: prepend the message, append the key, hash it. Only someone knowing K can compute the hash.

Step 2: The vulnerability

This scheme is vulnerable to collision attacks on the hash function.

If an attacker can find two different messages m1 and m2 such that:

H(m1) = H(m2) (a collision in the hash function without the key)

Then:

H(m1 || K) = H(m2 || K) ? Not necessarily, because the key is appended after.

But more dangerously: If attacker finds m1 and m2 such that:

H(m1) = H(m2) and both are full blocks or have same length, then appending the same suffix (K) might preserve the collision depending on the hash construction.

Step 3: Concrete attack using length extension

Actually, the more serious attack is different:

If H is Merkle-Damgård (like MD5, SHA-1, SHA-2), then H(m || K) is vulnerable to length extension!

Attacker who sees MAC for message m:

tag = H(m || K)

They can compute H(m || K || padding || extra) without knowing K!

But that's not H(m || extra || K), which would be different.

Step 4: The real attack - finding collisions in the hash

The actual attack on H(m || K):

1. Attacker finds two messages m1 and m2 such that H(m1) = H(m2) (collision in the hash function)

2. Because of how Merkle-Damgård works, if H(m1) = H(m2), then for any suffix S, H(m1 || S) = H(m2 || S)

3. Therefore, H(m1 || K) = H(m2 || K)

4. So MAC(K, m1) = MAC(K, m2)

Step 5: Attack scenario

1. Attacker finds collision pair (m1, m2) where:

- m1 = "Transfer $100 to Alice"

- m2 = "Transfer $10000 to Mallory"

2. Attacker tricks system into signing m1 (seems harmless)

3. The signature/MAC for m1 is also valid for m2

4. Attacker submits m2 with the valid MAC

Step 6: Why H(key || message) is also bad

The alternative H(key || message) is vulnerable to length extension, as we saw in Problem 3.4.

Answer: H(message || key) is insecure because hash collisions allow forgeries. If H(m1) = H(m2), then H(m1 || K) = H(m2 || K), breaking the MAC. This is why HMAC's two-pass construction is necessary.

Problem 4.3: CBC-MAC Vulnerability

CBC-MAC is a MAC built from a block cipher in CBC mode. For fixed-length messages, it's secure. But for variable-length messages, it's vulnerable to a specific attack. Explain the attack and how it works.

Solution

Step 1: How CBC-MAC works

For a message M divided into blocks M1, M2, ..., Mn:

C1 = E(K, M1)

C2 = E(K, M2 $\oplus$ C1)

C3 = E(K, M3 $\oplus$ C2)

...

MAC = Cn

Where E is a block cipher encryption.

Step 2: The variable-length vulnerability

If messages can have different lengths, an attacker can do:

1. Obtain MAC for one-block message M1: T1 = E(K, M1)

2. Obtain MAC for two-block message (X, Y): T2 = E(K, Y ⊕ E(K, X))

Step 3: The attack

Let T1 = E(K, M1)

Now construct message (M1, T1 ⊕ M2) for some chosen M2:

When computing CBC-MAC for (M1, T1 ⊕ M2):

First block: C1 = E(K, M1) = T1

Second block: Input = (T1 ⊕ M2) ⊕ C1 = (T1 ⊕ M2) ⊕ T1 = M2

So C2 = E(K, M2)

Therefore MAC((M1, T1 ⊕ M2)) = E(K, M2)

But E(K, M2) is just the MAC of the single-block message M2!

Step 4: Concrete example

1. Attacker obtains MAC for "A" = T_A

2. Attacker wants to forge MAC for "B" without querying it

3. Attacker queries MAC for ("A", T_A ⊕ "B")

4. The response is MAC("B")!

5. Attacker now has valid MAC for "B" without ever querying it

Step 5: Why this breaks security

This is a valid forgery:

- The attacker never queried "B" directly

- But they obtained its MAC through a different query

- This violates existential unforgeability

Step 6: Fixes

To fix CBC-MAC for variable-length messages:

- Use different keys for different lengths

- Prepend length to message before MACing

- Use CMAC (CBC-MAC with a tweak)

Answer: CBC-MAC's vulnerability allows an attacker to obtain the MAC of any single-block message by querying a cleverly constructed two-block message. This breaks unforgeability and shows why naive constructions can be dangerous.

Problem 4.4: Timing Attacks on MAC Verification

Explain why MAC verification must use constant-time comparison. Describe a timing attack that could recover a MAC tag byte-by-byte if verification uses early-exit comparison.

Solution

Step 1: The problem with early-exit comparison

Standard string comparison (like memcmp in C, or == in many languages) returns as soon as it finds a mismatch:

```c
int memcmp(const void *s1, const void *s2, size_t n) {
    const unsigned char *p1 = s1, *p2 = s2;
    for (size_t i = 0; i < n; i++) {
        if (p1[i] != p2[i])
            return p1[i] - p2[i];  // Exits immediately on mismatch
    }
```

```
    return 0;
}
```

The time taken depends on where the first mismatch occurs.

Step 2: The attack scenario

1. Attacker wants to forge a MAC tag for a message

2. They can send guesses to the server and measure response time

3. Server verifies by comparing attacker's tag with correct tag

4. If comparison is not constant-time, attacker learns information

Step 3: How the attack works

Let correct tag be T = [t1, t2, t3, ..., tn] (each byte)

Attacker sends guess G = [g1, g2, g3, ..., gn]

Server compares byte by byte:

If g1 = t1:

- Compare byte 1: equal, continue to byte 2

- Compare byte 2: if g2 ≠ t2, exit here

- Total time = time for 2 comparisons

If g1 ≠ t1:

- Compare byte 1: not equal, exit immediately

- Total time = time for 1 comparison

Step 4: Measuring the difference

Even tiny time differences (nanoseconds) can be measured by:

- Taking many samples

- Using high-resolution timers

- Averaging out network jitter with many attempts

Step 5: Recovering the tag byte by byte

Algorithm:

1. For first byte, try all 256 possibilities

2. The one that takes slightly longer (because it passed first byte check) is correct

3. Fix first byte, try second byte, look for timing difference

4. Continue until entire tag recovered

Step 6: Concrete numbers

If each comparison takes 10 ns:

- Correct first byte: 20 ns total (2 comparisons before failing on second)

- Wrong first byte: 10 ns total (1 comparison then exit)

Difference = 10 ns

With enough samples (thousands), this difference becomes statistically significant

Step 7: Why constant-time comparison fixes this

Constant-time comparison always takes the same amount of time regardless of where mismatch occurs:

```c
int constant_time_memcmp(const void *s1, const void *s2, size_t n) {
    const unsigned char *p1 = s1, *p2 = s2;
    int result = 0;
    for (size_t i = 0; i < n; i++) {
        result |= p1[i] ^ p2[i];  // XOR accumulates differences
    }
    return result;  // Always takes n iterations
}
```

```

Now time reveals nothing about where the mismatch occurred.

Answer: Timing attacks exploit early-exit comparison to learn the correct MAC tag byte-by-byte by measuring verification time differences. Constant-time comparison prevents this by ensuring verification time is independent of the tag content. This is why functions like `hmac.compare_digest()` in Python or `crypto.timingSafeEqual()` in Node.js must be used.
```

Chapter 5

Hash-based Message Authentication Code (HMAC)

5.1 The Need for HMAC

While it might seem natural to construct a MAC by simply hashing the key concatenated with the message, this naive approach has serious security vulnerabilities. HMAC was designed to address these weaknesses while providing a provably secure construction.

Why Not H(key || message)?

The straightforward approach of computing H(key || message) suffers from the length-extension attack. For hash functions based on the Merkle-Damgård construction (MD5, SHA-1, SHA-2), if an attacker knows H(key || message), they can compute H(key || message || padding || extra) without knowing the key.

This allows an attacker to forge MAC tags for messages that extend the original message. If the original message was "Pay $100", an attacker could compute a valid tag for "Pay $100 followed by some carefully crafted extra data that changes the meaning entirely.

Why Not H(message || key)?

The alternative of appending the key after the message, H(message || key), is vulnerable to different attacks. If an attacker can find collisions in the underlying hash function, they might be able to produce two different messages with the same hash, which would then have the same MAC tag when the key is appended.

5.2 The HMAC Construction

HMAC, defined in RFC 2104, uses a clever two-pass construction that provides security even if the underlying hash function is vulnerable to length-extension attacks. The construction works with any cryptographic hash function and has been proven secure under reasonable assumptions about the hash function's compression function.

The HMAC Formula

HMAC(K, m) = H((K' ⊕ opad) || H((K' ⊕ ipad) || m))

This formula might look complex, but each component serves a specific security purpose.

Key Preprocessing

The key K undergoes preprocessing to ensure it's the right length for the hash function's block size:

- If K is shorter than the block size, it's padded with zeros to reach the block size

- If K is longer than the block size, it's hashed first to produce a key of the hash output length, then padded with zeros to the block size

This preprocessing ensures that the key is exactly one block long for the subsequent XOR operations.

Inner and Outer Padding

The constants ipad and opad are carefully chosen:

- ipad is the byte 0x36 repeated to fill a block

- opad is the byte 0x5c repeated to fill a block

These values were selected to have a large Hamming distance (they differ in many bits) to ensure that the inner and outer hashes are as independent as possible.

The Two-Pass Structure

The construction works in two hashing passes:

First pass (inner hash):

1. XOR the preprocessed key with ipad

2. Append the message to this XOR result

3. Hash everything to produce the inner hash

Second pass (outer hash):

1. XOR the preprocessed key with opad

2. Append the inner hash to this XOR result

3. Hash everything to produce the final HMAC

5.3 Security Analysis

HMAC (Hash-based Message Authentication Code), as standardized and recommended by organizations like National Institute of Standards and Technology, is carefully designed to stay secure even when the underlying hash function has weaknesses.

Pillar 1: Outer hash defeats length-extension attacks

Many common hash functions are vulnerable to something called a **length-extension attack**. In simple terms:

- If an attacker knows hash(secret || message),
- they may be able to compute
  hash(secret || message || extra_data)
  without knowing the secret.

That's dangerous for naïve "key + hash" constructions.

How HMAC fixes this

HMAC does **two hashes**, not one:

1. Inner hash:
   H((key ⊕ ipad) || message)
2. Outer hash:
   H((key ⊕ opad) || inner_hash)

So even if an attacker could somehow extend the *inner* hash:

- They still cannot compute the **outer hash**, because:

- the outer hash *requires the secret key again*.
- the attacker never sees (key ⊕ opad).

Result: length-extension becomes useless, because the final MAC always depends on a second, key-protected hash.

Think of it like this:
Even if someone tampers with the inside lock, there's still an outer lock that needs the same secret key.

Pillar 2: Two different paddings make inner and outer hashes independent

HMAC uses **two fixed, different constants**:

- ipad (inner padding)
- opad (outer padding)

These are XORed with the key to produce two *different derived keys*:

- (key ⊕ ipad) → used only inside
- (key ⊕ opad) → used only outside

Why this matters

This guarantees that:

- The inner hash and outer hash behave like **two separate cryptographic functions**, even though they use the same hash algorithm.
- Any structural weakness or partial collision in the inner hash **does not transfer** to the outer hash.

In other words:

- You cannot reuse tricks from the inner layer to attack the outer layer.
- The design prevents "cross-contamination" between the two stages.

This creates **domain separation**: the inner and outer computations live in different cryptographic worlds.

HMAC is secure because:

1. **Double hashing with the key on both layers** blocks length-extension attacks.
2. **Different paddings (ipad and opad)** force cryptographic independence between inner and outer hashes.

Provable Security

Provable security is a framework in cryptography where we don't just *believe* an algorithm is secure — we **mathematically prove** that breaking it is *at least as hard as* solving some well-known difficult problem.

Instead of saying:

"No one has broken this yet, so it's probably safe,"

provable security says:

"If you can break this system, then you can also solve a problem believed to be computationally infeasible."

That's a *much stronger* statement.

This approach was formalized by researchers like Shafi Goldwasser and Silvio Micali.

The core idea (Reduction)

Provable security uses something called a **reduction**.

It works like this:

1. Assume an attacker **A** can break your cryptographic scheme.
2. You construct another algorithm **B** that uses attacker A as a subroutine.
3. Algorithm B then solves a famous hard problem (for example, factoring or discrete logarithms).

So you prove:

If A exists → B exists → hard problem becomes easy.

Since the hard problem is believed to be impossible to solve efficiently, attacker A probably **does not exist**.

This is written informally as:

Breaking Scheme ≥ Solving Hard Problem

Meaning: breaking the scheme is *at least as hard* as the underlying problem.

Example intuition (simple)

Suppose we prove:

- If someone can forge an HMAC,
- then they can also break the underlying hash function's pseudorandomness.

Because hash pseudorandomness is assumed secure, forging HMAC is also assumed infeasible.

So HMAC inherits its security from the hash.

This is why standards bodies like National Institute of Standards and Technology rely heavily on constructions with provable guarantees.

What Provable Security gives you

1. Clear assumptions

Every proof explicitly states:

- What attacker can do
- What is assumed hard

No hidden magic.

2. Formal threat models

Security is proven under models like:

- Chosen-plaintext attack (CPA)
- Chosen-ciphertext attack (CCA)
- Existential forgery (for MACs)

3. Modular design

If your system is built from secure components:

- secure hash
- secure MAC

- secure encryption

provable security lets you combine them safely.

Important limitation

Provable security does **NOT** mean:

- the code has no bugs
- side-channel attacks don’t exist
- implementations are perfect

It only proves security of the **mathematical model**.

Real-world issues (timing attacks, power analysis, bad randomness) are outside the proof.

Provable security means:

Breaking the cryptographic scheme is mathematically reduced to solving a known hard problem — so unless that hard problem is cracked, the scheme remains secure.

5.4 HMAC in Practice

HMAC is widely deployed across the internet. When you use APIs from services like AWS, Google, or Stripe, you're likely using HMAC to authenticate your requests. Webhooks from GitHub, Slack, and countless other services are secured with HMAC signatures.

The choice of underlying hash function matters. HMAC-MD5 is considered weak because MD5 is broken, even though HMAC's construction provides some protection. HMAC-SHA1 is deprecated as SHA-1 weakens. HMAC-SHA256 is the current standard recommendation, offering strong security and wide support.

Practice Problems Set 5: HMAC

Problem 5.1: HMAC Key Preprocessing

For HMAC-SHA256 with block size 512 bits (64 bytes), show what happens when:

a) Key is 20 bytes

b) Key is 80 bytes

c) Key is exactly 64 bytes

Solution

Step 1: Recall HMAC key preprocessing rules

- If key length ≤ block size: Pad with zeros to block size

- If key length > block size: Hash the key first, then pad to block size

Block size for SHA-256 = 64 bytes

Hash output size for SHA-256 = 32 bytes

Step 2: Case (a) Key = 20 bytes

20 bytes < 64 bytes, so:

K' = key || zeros to make 64 bytes

K' = [20 bytes of key][44 bytes of 0x00]

Step 3: Case (b) Key = 80 bytes

80 bytes > 64 bytes, so:

First, hash the key: K_hash = SHA-256(key) = 32 bytes

Then K' = K_hash || zeros to make 64 bytes

K' = [32 bytes of hash][32 bytes of 0x00]

Step 4: Case (c) Key = 64 bytes exactly

Key length equals block size, so:

K' = key (unchanged, no padding needed)

Step 5: Why this matters

The preprocessing ensures K' is always exactly one block long, which is necessary for the XOR operations with ipad and opad.

If we didn't hash long keys:

- The key might span multiple blocks

- The XOR with ipad would only affect first block

- Security properties would be different

If we didn't pad short keys:

- XOR with ipad would be on variable-length data

- The construction wouldn't be well-defined

Answer

a) 20-byte key → K' = key + 44 zero bytes

b) 80-byte key → K' = SHA-256(key) + 32 zero bytes

c) 64-byte key → K' = key (unchanged)

Problem 5.2: HMAC Security Proof Concept

Explain why HMAC's two-pass construction prevents length extension attacks, even when using a Merkle-Damgård hash function like SHA-256.

Solution

Step 1: Recall length extension attack

For H(key || message):

- Attacker sees H(key || message) = state after processing key||message

- Can use this state as IV to continue hashing with extra data

- Gets H(key || message || padding || extra) without knowing key

Step 2: HMAC structure

HMAC(K, m) = H((K' ⊕ opad) || H((K' ⊕ ipad) || m))

Let's denote:

inner_hash = H((K' ⊕ ipad) || m)

final = H((K' ⊕ opad) || inner_hash)

Step 3: Why attacker can't extend inner hash

Suppose attacker sees HMAC(K, m). They know:

- inner_hash is not directly output

- They only see final = H((K' ⊕ opad) || inner_hash)

To extend inner_hash, they would need to use inner_hash as IV and continue hashing.

But they don't know inner_hash! It's hidden inside the outer hash computation.

Step 4: Attempting to extend outer hash

Could attacker extend the outer hash?

They see final = H((K' ⊕ opad) || inner_hash)

This is itself a Merkle-Damgård hash. If they try to extend it, they'd get:

H((K' ⊕ opad) || inner_hash || padding || extra)

But this would be HMAC for a different message? No, because the inner_hash is fixed.

The result would be H((K' ⊕ opad) || inner_hash || padding || extra)

But this is NOT equal to HMAC(K, m || something) because the inner hash would be wrong.

Step 5: The double protection

HMAC has two layers of protection:

1. Inner hash: (K' ⊕ ipad) || message
   - Attacker can't compute this without K
   - Even if they knew inner hash, they'd need K'⊕opad for outer

2. Outer hash: (K' ⊕ opad) || inner_hash
   - Attacker doesn't know K'⊕opad
   - Can't compute outer hash without it
   - Can't extend because don't know the key material at start

Step 6: Analogy

Think of HMAC as a locked box inside another locked box:

- Inner box: contains message, locked with key material (ipad)

- Outer box: contains inner box, locked with different key material (opad)

Even if attacker knows what's in inner box (they don't), they still need the outer key to open the outer box.

Answer

HMAC prevents length extension through its two-pass structure. The inner hash is never directly exposed, and the outer hash uses a different key (K'⊕opad) to protect it. Even if an attacker could extend the outer hash, they'd get a value that doesn't correspond to any valid HMAC because the inner hash wouldn't match.

Problem 5.3: HMAC with Truncated Output

Some applications truncate HMAC output to save space (e.g., using only first 128 bits of HMAC-SHA256). What are the security implications? If the full HMAC-SHA256 has 256-bit security against forgery, what security does the truncated version have?

Solution

Step 1: Understanding HMAC security

HMAC-SHA256's security is based on:

- The underlying hash's properties

- The key size (usually at least 128 bits recommended)

- The output length

Full HMAC-SHA256 provides:

- 256-bit security against certain attacks

- Actually, security is min(key_size, output_size) against brute force

Step 2: Effect of truncation

Truncating to t bits means:

- Attacker trying to guess a valid tag has probability 2^(-t) per attempt

- After seeing q valid message-tag pairs, attacker's advantage in forging is about q/2^t

Step 3: Security level calculation

For t-bit truncated HMAC:

- Forgery probability per attempt = 1/2^t

- To achieve 2^-32 forgery probability (acceptable for many apps), need to limit attempts

If t = 128 bits:

- Brute force guessing: need 2^128 attempts on average

- This is still astronomically large (3.4 × 10^38 attempts)

Step 4: Birthday attacks on truncation

Important: Truncation doesn't enable birthday attacks on the MAC itself because MAC verification is deterministic.

Birthday attacks apply to collision finding, not forgery. For forgery, attacker needs to produce a specific tag for a specific message, not just any collision.

Step 5: Practical considerations

Many standards recommend truncation:

- HMAC-SHA256-128: Use first 128 bits

- Provides 128-bit security against forgery

- Still secure for most applications

- Saves bandwidth/storage

AWS uses truncated HMAC for some API signatures.

Step 6: When truncation is dangerous

If t is too small (e.g., 32 bits):

- Attacker can guess tags with probability 1/2^32 ≈ 2.3 × 10^-10

- With 1 million attempts per second, would find valid tag in about 1 hour

- This is feasible!

Step 7: Recommendation

NIST SP 800-107 recommends:

- Minimum output length 112 bits for applications through 2030

- 128 bits recommended for new applications

- Truncation should not reduce security below these minimums

Answer

Truncating HMAC-SHA256 to t bits provides t-bit security against forgery (probability 2^-t per attempt). Truncation to 128 bits is still secure for most applications, but truncation below 112 bits is not recommended. The security is against brute force guessing, not collision attacks.

Problem 5.4: HMAC Key Size Selection

An organization is deploying HMAC-SHA256 for API authentication. They have millions of users and need to decide on key sizes. What are the trade-offs between 128-bit, 256-bit, and 512-bit keys? Which would you recommend and why?

Solution

Step 1: HMAC key size options

128-bit keys:

- 2^128 possible keys = 3.4 × 10^38

- Brute force infeasible with current technology

- Matches 128-bit security level

256-bit keys:

- 2^256 possible keys = 1.16 × 10^77

- Massive keyspace, overkill for symmetric crypto

- Provides 256-bit security

512-bit keys:

- 2^512 keys = 1.34 × 10^154

- Extremely large, completely unnecessary

Step 2: Key storage considerations

For millions of users:

- 128-bit key = 16 bytes storage per user

- 256-bit key = 32 bytes per user

- For 10 million users: 160 MB vs 320 MB difference

- Storage cost is negligible either way

Step 3: Key generation considerations

- Need high-quality random bits for each key

- 128 bits of entropy is sufficient

- Generating 256 bits doesn't increase security if source has only 128 bits entropy

Step 4: Performance impact

HMAC performance with different key sizes:

- Keys ≤ block size (64 bytes for SHA-256): no hashing needed

- Keys > block size: must hash key first

128-bit (16 bytes) and 256-bit (32 bytes) are both < 64 bytes → no hashing

512-bit (64 bytes) = exactly block size → no hashing (but key is full block)

>512 bits would require hashing

Step 5: Security analysis

HMAC security is min(key_size, output_size, collision_resistance)

For SHA-256:

- Output size = 256 bits

- Collision resistance = 128 bits (due to birthday bound)

- So security is limited to 128 bits anyway!

This is crucial: Even with 256-bit keys, HMAC-SHA256 only provides 128-bit security against collision attacks because of the hash output size.

Step 6: The recommendation

Given that HMAC-SHA256's effective security is 128 bits:

- 128-bit keys provide full security

- 256-bit keys provide no additional security benefit

- 512-bit keys waste space

Step 7: Future-proofing considerations

If concerned about quantum computers:

- Grover's algorithm halves security against brute force

- 128-bit classical → 64-bit quantum (too low)

- 256-bit classical → 128-bit quantum (acceptable)

So for long-term security against quantum attacks, 256-bit keys might be preferred.

Step 8: Final recommendation

For most applications today:

- Use 128-bit keys (16 bytes)

- Store as hex (32 characters) or base64

- Rotate keys periodically

For high-security, long-term applications:

- Use 256-bit keys (32 bytes)

- Provides quantum safety margin

- Still efficient (under block size)

Answer

For HMAC-SHA256, 128-bit keys provide full classical security matching the hash's 128-bit collision resistance. Larger keys don't increase security but may be used for quantum safety.

The small storage difference (16 vs 32 bytes) is negligible, so 256-bit keys are a safe choice for future-proofing.

Problem 5.5: HMAC vs Digital Signature

Compare HMAC and digital signatures for API authentication. When would you choose one over the other? Consider factors like key distribution, non-repudiation, performance, and scalability.

Solution

Step 1: Key distribution

HMAC (symmetric):

- Same key used by client and server

- Must securely share key with each client

- Key distribution problem: N clients need N keys shared securely

- If server compromised, all client keys exposed

Digital signatures (asymmetric):

- Server has private key, clients have public key

- Public keys can be distributed openly (even embedded in client code)

- Only one private key to protect

- Client compromise doesn't reveal server's key

Step 2: Non-repudiation

HMAC:

- No non-repudiation

- Server could have generated the MAC (since it knows the key)

- Client can deny sending message

Digital signatures:

- Provides non-repudiation

- Only client possesses private key

- Client cannot deny signing (in court, for example)

Step 3: Performance

HMAC:

- Very fast (hash functions optimized)

- ~100-500 MB/sec on modern hardware

- Low overhead per request

Digital signatures:

- Much slower (especially signing)

- RSA: 100-1000x slower than HMAC

- ECDSA: faster than RSA but still slower than HMAC

- Verification faster than signing but still significant

Step 4: Scalability with many clients

HMAC:

- Server must store all client keys

- Key management becomes complex with millions of users

- Key rotation requires updating all clients

Digital signatures:

- Server only needs to know verification keys (public)

- Can be stored in directory or even distributed with requests

- Key rotation: clients generate new key pair, send new public key

Step 5: Use case analysis

Choose HMAC when:

- Internal systems, microservices (controlled environment)

- High performance needed (millions of requests)

- Non-repudiation not required

- Can securely distribute keys out-of-band

Choose Digital Signatures when:

- Open APIs with untrusted clients

- Need non-repudiation (financial transactions)

- Many clients, difficult key distribution

- Clients need to prove identity to third parties

Step 6: Hybrid approaches

Many systems use both:

- TLS (digital signatures) for initial authentication

- Then establish session keys for HMAC during connection

- Best of both worlds

Step 7: Real-world examples

HMAC used by:

- AWS API signatures

- Webhooks (GitHub, Stripe)

- JWT with HS256 (symmetric)

Digital signatures used by:

- SSL/TLS certificates

- JWT with RS256/ES256 (asymmetric)

- Bitcoin transactions

- Code signing

Answer:

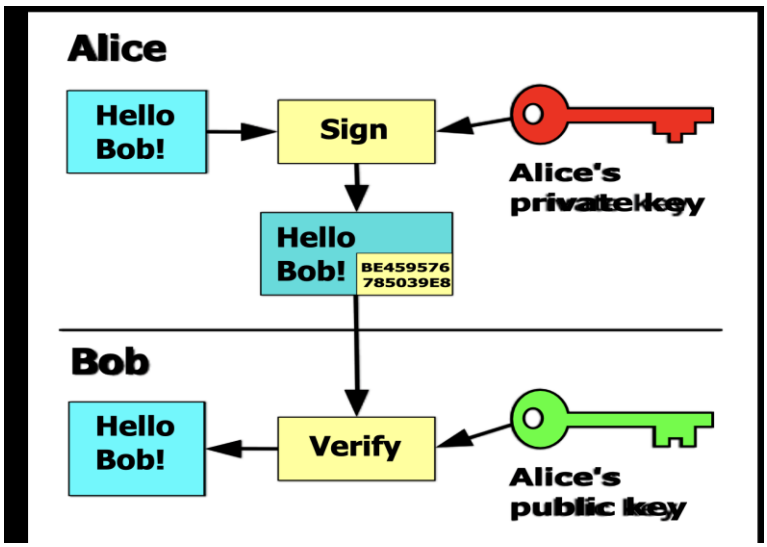
HMAC is faster and simpler but requires secure key distribution and provides no non-repudiation. Digital signatures scale better for many clients and provide non-repudiation but are slower. Choice depends on threat model, performance requirements, and whether clients need to be held accountable for their messages.

Chapter 6

Digital Signature Schemes

6.1 The Concept of Digital Signatures

Digital signatures are a cryptographic mechanism used to guarantee **authenticity, integrity, and non-repudiation** of digital data, meaning they prove who sent a message, ensure it was not modified, and prevent the sender from later denying it; they work using public-key cryptography, where the sender creates a signature by hashing the message and encrypting that hash with their private key, and the receiver verifies it using the sender's public key, a concept rooted in foundational work by Whitfield Diffie and Martin Hellman and made practical through algorithms developed by Ron Rivest, Adi Shamir, and Leonard Adleman; today, digital signatures are standardized and recommended by bodies such as National Institute of Standards and Technology and are widely used in applications like online banking, e-governance, software distribution, and electronic contracts, providing mathematically provable assurance that only the holder of the private key could have signed the data and that even the smallest change to the message will invalidate the signature.



Historical Context

The concept of digital signatures emerged alongside public-key cryptography in the 1970s. Whitfield Diffie and Martin Hellman first described the idea in their groundbreaking 1976 paper "New Directions in Cryptography," though they didn't provide a concrete

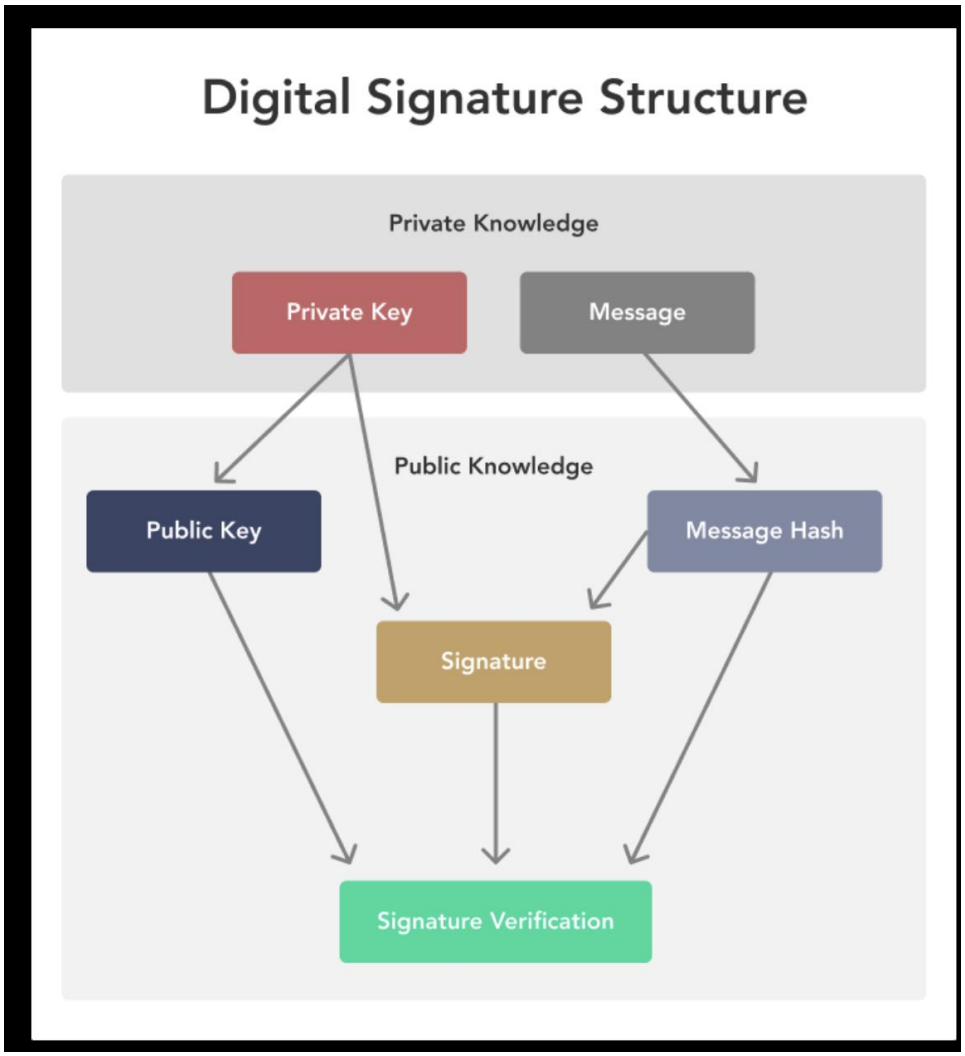
implementation. The first practical digital signature scheme was RSA, developed by Ron Rivest, Adi Shamir, and Leonard Adleman in 1977.

6.2 Core Security Properties

Digital signatures provide three essential security services:

Authentication

The recipient can verify that the signature was created by the claimed sender. Since only the sender possesses their private key, a valid signature proves the sender's identity.



Non-Repudiation

The sender cannot later deny having signed the document. Unlike a MAC, where either party could have generated the tag, a digital signature can only be generated by the private key holder. This provides cryptographic proof that can be presented to a third party, such as a court.

Integrity

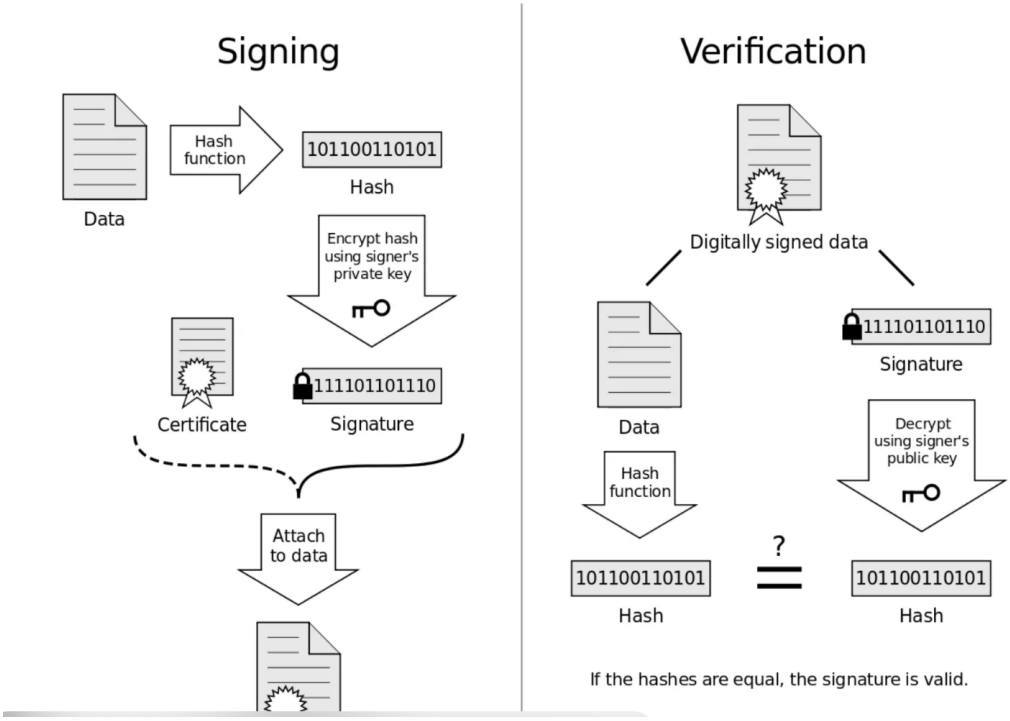
Any modification to the signed document will cause signature verification to fail. This ensures that the document hasn't been altered since it was signed.

6.3 How Digital Signatures Work

Digital signatures rely on asymmetric cryptography, where each user has a pair of mathematically related keys:

- A private key, kept secret by the owner

- A public key, freely distributed to anyone who needs to verify signatures



The Signing Process

When Alice wants to sign a document:

1. She computes a cryptographic hash of the document. This step is crucial for efficiency—signing the fixed-size hash is much faster than signing the entire document, which could be gigabytes in size.

2. She uses her private key to transform this hash into a signature. The exact transformation depends on the signature algorithm (RSA, DSA, ECDSA, etc.).

3. She sends the original document along with the signature to Bob.

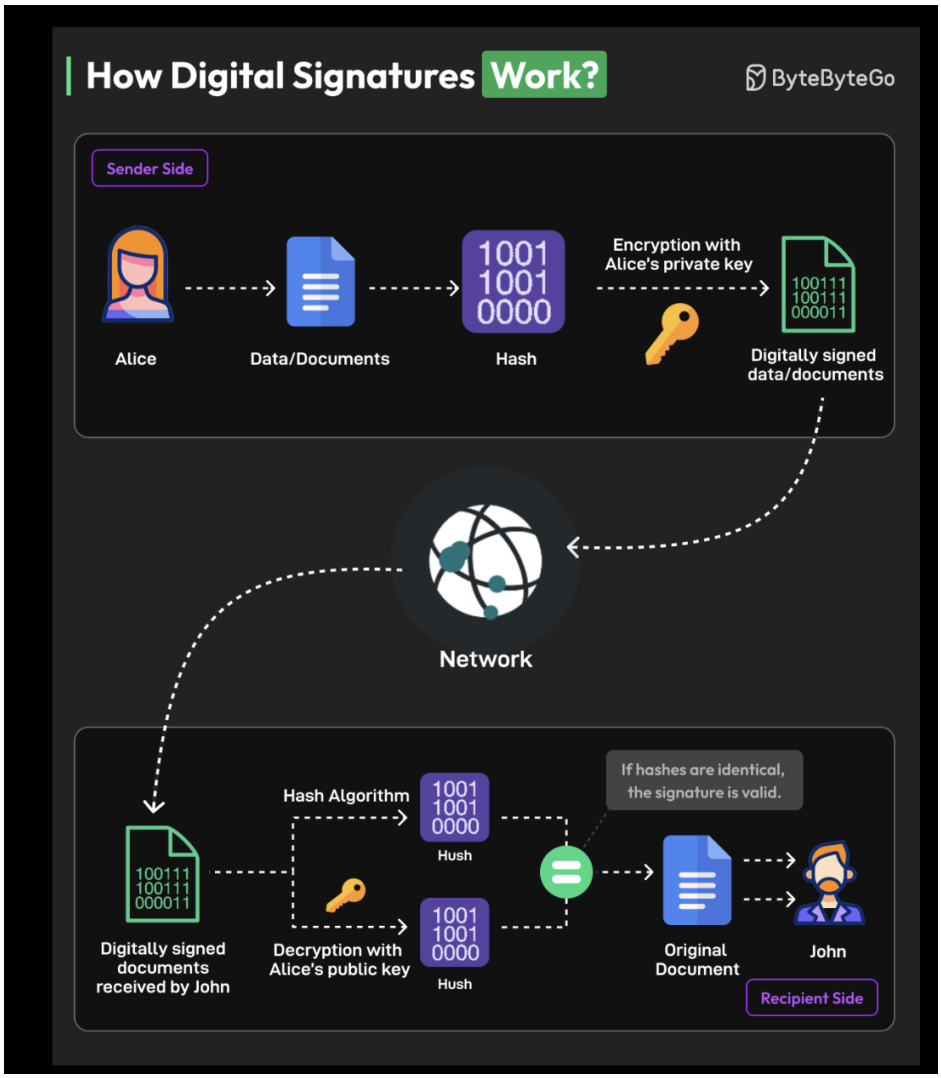
The Verification Process

When Bob receives the document and signature:

1. He computes the hash of the received document using the same hash function Alice used.

2. He uses Alice's public key to verify that the signature corresponds to this hash.

3. If verification succeeds, Bob knows the document came from Alice and hasn't been modified.



6.4 Hash-and-Sign Paradigm

The approach of hashing before signing, known as the hash-and-sign paradigm, is universal in practical digital signature schemes. This paradigm offers several advantages:

Efficiency

Public-key cryptographic operations are computationally expensive. Signing a fixed-size hash instead of a potentially enormous document dramatically improves performance.

Compatibility

Signature algorithms are designed to work on fixed-size inputs. Hashing converts arbitrary-length messages into the correct input size.

Security

Hashing provides an additional layer of security. Even if the signature algorithm had some mathematical weakness, the hash function's properties make it difficult for an attacker to exploit.

Practice Problems Set 6: Digital Signature Fundamentals

Problem 6.1: Hash-and-Sign Security

Explain why signing the hash of a message is as secure as signing the message itself. What properties of the hash function are necessary for this to hold?

Solution

Step 1: The security argument

When we sign a message, we want to ensure:

1. The signer cannot later claim they signed something else

2. No one can forge signatures on other messages

If we sign H(m) instead of m, we need to ensure that:

- Finding m' with H(m') = H(m) is hard (second pre-image resistance)

- Finding any two messages with same hash is hard (collision resistance)

Step 2: Why second pre-image resistance matters

Suppose Alice signs H(m). If second pre-image resistance fails, Mallory could:

1. Find $m' \neq m$ such that $H(m') = H(m)$

2. Claim Alice actually signed m' (since signature verifies for m' too)

This would break non-repudiation.

Step 3: Why collision resistance matters

Suppose Mallory wants to get Alice to sign something malicious:

1. Mallory finds m (innocent) and m' (malicious) with H(m) = H(m')

2. Asks Alice to sign m (looks harmless)

3. Alice signs, signature verifies for both m and m'

This breaks both authentication and non-repudiation.

Step 4: What about pre-image resistance?

Pre-image resistance isn't directly needed here because:

- Attacker already knows the message they want to forge

- They need to find a different message with same hash (second pre-image)

- Or two messages with same hash (collision)

Step 5: Formal security

In the random oracle model (where hash is ideal):

- Signing H(m) is as secure as signing m

- Reduces to: can't find collisions in hash

Step 6: Real-world implication

This is why hash functions used in signatures must be collision-resistant. If collisions are found (like MD5, SHA-1), the signature scheme becomes vulnerable even if the underlying math (RSA, ECDSA) is secure.

Answer:

Signing the hash is as secure as signing the message provided the hash function has strong collision resistance and second pre-image resistance. These properties ensure that a signature on one message cannot be transferred to a different message, and that an attacker cannot find two messages that would share a signature.

Problem 6.2: Signature Size vs Security

Compare RSA and ECDSA signature sizes for 128-bit security level. RSA requires 3072-bit keys/modulus, while ECDSA with secp256k1 uses 256-bit keys. What are the practical implications of this size difference?

Solution

Step 1: Signature sizes

RSA-3072:

- Signature is one number modulo N (3072 bits)

- Signature size = 3072 bits = 384 bytes

- Independent of message size (after hashing)

ECDSA with secp256k1 (256-bit curve):

- Signature is two numbers (r, s) each 256 bits

- Total = 512 bits = 64 bytes

- Can be further compressed in some applications

Step 2: Size ratio

384 bytes vs 64 bytes = 6:1 ratio

ECDSA signatures are 6× smaller

Step 3: Practical implications

Storage:

- Blockchain (Bitcoin): Each transaction includes signature

- 10 million transactions × (384-64) = 3.2 GB saved with ECDSA

- Significant blockchain bloat avoided

Transmission:

- Smaller signatures = faster transmission

- Important for low-bandwidth environments (mobile, IoT)

- Less data over cellular networks

Embedded systems:

- Smaller signatures easier to handle in constrained memory

- 64 bytes fits in small buffers, 384 bytes may require dynamic allocation

Step 4: Key sizes also differ

RSA-3072:

- Public key: 3072 bits (384 bytes)

- Private key: larger (contains primes, exponents)

ECDSA-256:

- Public key: 33 bytes (compressed) or 65 bytes (uncompressed)

- Private key: 32 bytes

Step 5: Performance differences

RSA verification: fast (exponentiation with small public exponent)

ECDSA verification: slower than RSA but faster than RSA signing

RSA signing: slow (exponentiation with large private exponent)

ECDSA signing: faster than RSA

Step 6: Why RSA is still used

- Well-understood, longer track record

- Patents on ECC expired relatively recently

- Some standards still require RSA

- Quantum resistance? (both broken by Shor's algorithm anyway)

Answer

For 128-bit security, RSA signatures are 384 bytes while ECDSA signatures are 64 bytes—a 6× difference. This makes ECDSA preferable for bandwidth-constrained or storage-

constrained applications like blockchain, while RSA remains common in traditional PKI where signature size matters less.

Problem 6.3: Non-Repudiation Scenario

Alice signs a contract to pay Bob $1000 using her private key. Later, Alice claims her private key was stolen and the signature is forged. Bob presents the signature in court. What cryptographic evidence supports Bob's claim? What factors might weaken this evidence?

Solution

Step 1: The cryptographic argument

Digital signatures provide non-repudiation because:

- Only Alice's private key can create signatures that verify with her public key

- If signature verifies, it must have been created by someone possessing that key

- Assuming key wasn't compromised, this proves Alice signed

Step 2: Evidence Bob can present

1. The signed document (contract)

2. Alice's public key (certified by CA or previously known)

3. The signature

4. Verification that signature is valid for document with Alice's public key

5. Timestamp proving signature was created before alleged theft

Step 3: Alice's possible defenses

Defense 1: Key was stolen before signing

- Bob would need to prove signing occurred after theft

- Timestamps help, but timestamps themselves could be forged

- Requires trusted timestamp authority

Defense 2: Key was compromised and used without knowledge

- Bob must prove Alice was negligent in protecting key

- Legal question, not cryptographic

Defense 3: Algorithm was broken

- If hash or signature algorithm has known vulnerabilities

- Could claim signature was forged through cryptanalysis

- Less plausible with strong algorithms (SHA-256, ECDSA)

Defense 4: Implementation flaw

- Random number generator failure (e.g., reused k in ECDSA)

- Could have allowed key recovery

- Requires evidence of implementation issues

Step 4: Strengthening non-repudiation

Hardware security modules (HSM):

- Key never leaves tamper-resistant hardware

- Harder to claim theft

Multi-factor signing:

- Require additional authentication to use key

- Biometric, second device, etc.

Certificate transparency:

- Public logs of certificate issuance

- Can prove key was valid at time

Step 5: Legal considerations

Courts consider:

- Strength of cryptographic evidence

- Security practices of all parties

- Expert testimony on algorithm security

- Timing and circumstances

Answer

Cryptographically, a valid signature with Alice's public key is strong evidence she signed it, assuming key wasn't compromised. Alice can claim theft or compromise, shifting burden to prove when compromise occurred and whether she was negligent. Strong key protection (HSMs, multi-factor) and trusted timestamps strengthen the non-repudiation evidence.

Problem 6.4: Signature Verification Failure Analysis

A server receives a signed message but signature verification fails. List all possible reasons for this failure, categorized by:

a) Problems with the message

b) Problems with the signature

c) Problems with the key

d) Problems with the algorithm/implementation

Solution

Step 1: Problems with the message

Message modified in transit:

- Any change to message (even one bit) changes hash

- Signature won't verify

Different encoding:

- Text encoded as UTF-8 vs UTF-16 produces different bytes

- Line endings (CRLF vs LF) change hash

- Extra whitespace added/removed

Different format:

- JSON with different field order

- XML with different attribute ordering

- Binary vs text representation

Step 2: Problems with the signature

Signature corrupted:

- Bits flipped in transmission

- Truncated signature

- Extra data appended

Wrong signature format:

- DER encoding issues for RSA/ECDSA

- Wrong padding scheme

- Missing or extra headers

Signature from different algorithm:

- RSA signature tried with ECDSA verifier

- Wrong curve parameters

Step 3: Problems with the key

Wrong public key:

- Using wrong key to verify (different signer)

- Expired certificate

- Revoked key

Key corrupted:

- Public key bits flipped

- Wrong format (PEM vs DER)

Key parameter mismatch:

- RSA key size different than expected

- ECDSA curve mismatch

Step 4: Problems with algorithm/implementation

Hash algorithm mismatch:

- Message hashed with SHA-256, signature expects SHA-1

- Different hash used in signing vs verification

Padding issues:

- RSA PKCS#1 v1.5 vs PSS

- Wrong salt length for PSS

Implementation bugs:

- Constant-time verification bugs

- Integer overflow in parameter parsing

- Endianness issues

Step 5: Systematic debugging approach

1. Verify hash independently:
   - Compute hash of received message
   - Compare with expected (if known)
2. Verify signature format:
   - Parse signature, check lengths
   - Validate against algorithm spec
3. Verify key:
   - Check key format, parameters
   - Confirm it's the right key
4. Verify algorithm:

- Ensure same algorithm used for signing and verification

- Check parameters (curve, padding, etc.)

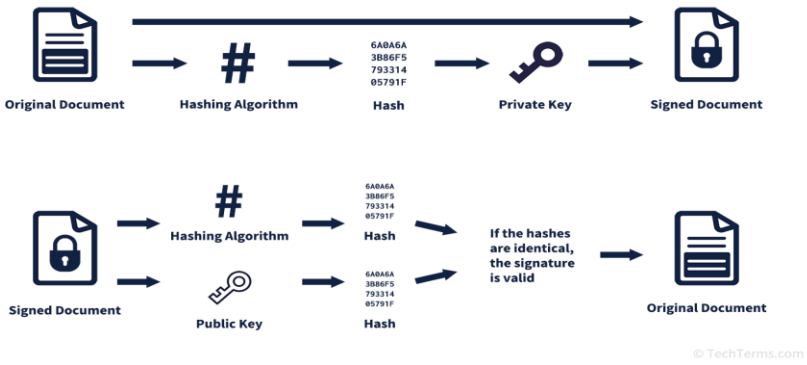
Step 6: Example troubleshooting checklist

```

□ Message bytes exactly as signed?

□ Encoding consistent?

□ Signature length correct?

□ Signature format valid?

□ Public key correct?

□ Key not expired/revoked?

□ Hash algorithm matches?

□ Padding scheme matches?

□ Implementation up to date?

Answer

Signature verification failures can stem from message modifications, signature corruption, wrong keys, or algorithm mismatches. Systematic debugging should check each component independently to isolate the issue.
```

Chapter 7

The Digital Signature Standard (DSS)

7.1 Overview of DSS



The Digital Signature Standard (DSS) is a Federal Information Processing Standard (FIPS 186) published by NIST that specifies acceptable digital signature algorithms for US government applications. First issued in 1994 and updated several times since, DSS has evolved to include multiple algorithms as cryptography has advanced.

Purpose and Scope

DSS provides a standardized framework for digital signatures, ensuring that different implementations can interoperate and meet minimum security requirements. Federal agencies must use DSS-approved algorithms for signing electronic documents, and the standard has been widely adopted in commercial applications as well.

7.2 Algorithms Specified in DSS

The current version of DSS approves three fundamentally different signature techniques:

Digital Signature Algorithm (DSA)

DSA was the original algorithm specified in DSS, designed specifically for digital signatures (unlike RSA, which can also be used for encryption). Its security is based on the discrete logarithm problem in finite fields.

DSA parameters include:

- A prime number p, typically 1024, 2048, or 3072 bits

- A prime number q, a 160-256 bit divisor of p-1

- A generator g of the q-order subgroup modulo p

- A private key x, a random number between 1 and q-1

- A public key y = g^x mod p

Signing with DSA involves generating a random per-message secret and computing two values, r and s, that constitute the signature. Verification checks that these values satisfy a mathematical relationship using the public key.

RSA Digital Signature Algorithm

RSA signatures, while predating DSS, were added in later revisions. RSA's security is based on the difficulty of factoring large composite numbers. The RSA private key consists of two large primes and a private exponent; the public key includes their product and a public exponent.

RSA signatures are particularly efficient to verify, making them popular for applications where signatures are verified frequently, such as SSL/TLS certificates.

Elliptic Curve Digital Signature Algorithm (ECDSA)

ECDSA is the elliptic curve variant of DSA, added to DSS as elliptic curve cryptography gained acceptance. Its security is based on the discrete logarithm problem over elliptic curve groups, which offers equivalent security to DSA and RSA with much smaller key sizes.

A 256-bit ECDSA key provides security comparable to a 3072-bit RSA key, making ECDSA attractive for resource-constrained environments like smart cards and mobile devices.

7.3 Parameter Selection and Security Levels

DSS specifies security levels corresponding to symmetric key strengths:

- 112-bit security: Requires 2048-bit RSA or DSA keys, or 224-bit ECDSA keys

- 128-bit security: Requires 3072-bit RSA or DSA keys, or 256-bit ECDSA keys

- 192-bit security: Requires 7680-bit RSA or DSA keys, or 384-bit ECDSA keys

- 256-bit security: Requires 15360-bit RSA or DSA keys, or 512-bit ECDSA keys

The dramatic difference in key sizes explains why ECDSA has become increasingly popular—it provides strong security with much smaller keys and signatures, reducing storage and transmission overhead.

7.4 ECDSA in Bitcoin

Bitcoin uses ECDSA with the secp256k1 elliptic curve, a specific curve standardized by the Standards for Efficient Cryptography group. This curve was chosen for several reasons:

Efficiency

secp256k1 is a Koblitz curve, which allows for particularly efficient implementation. The curve parameters were selected to enable fast computation, critical for the millions of signature verifications Bitcoin nodes perform daily.

Security

The curve provides 128-bit security, considered sufficient for Bitcoin's value and expected lifespan. No practical attacks against secp256k1 are known when properly implemented.

Simplicity

Unlike some other curves, secp256k1's parameters were chosen in a verifiably random way, reducing suspicion of hidden weaknesses or NSA backdoors.

Practice Problems Set 7: Digital Signature Standard

Problem 7.1: DSA Parameter Generation

In DSA, we need a prime p, a prime q dividing p-1, and a generator g of order q. If q is 160 bits and p is 1024 bits, approximately how many such p exist for a given q? Why is this relationship important?

Solution

Step 1: Understanding the relationship

We need p such that:

- p is prime

- $p \equiv 1 \bmod q$ (since q divides p-1)

- p is 1024 bits (roughly 2^1023 to 2^1024-1)

Step 2: Counting primes in arithmetic progression

By Dirichlet's theorem, primes are evenly distributed among residue classes coprime to the modulus.

Numbers of form p = kq + 1 that are 1024 bits:

Smallest: $k_{min} \approx (2^{1023} - 1)/q$

Largest: $k_{max} \approx (2^{1024} - 1)/q$

Number of k values = $(2^{1024} - 2^{1023})/q = 2^{1023}/q$

$q \approx 2^{160}$

So number of $k = 2^{1023} / 2^{160} = 2^{863}$

Step 3: Prime density

Prime number theorem: probability a random number of size $\sim 2^{1024}$ is prime $\approx 1/\ln(2^{1024}) = 1/(1024 \ln 2) \approx 1/710 \approx 0.0014$

So expected number of primes $p = 2^{863} \times 0.0014 \approx 2^{863} \times 2^{-9.5} \approx 2^{853.5}$

Step 4: Interpretation

2^853.5 is astronomically huge—far more than we need.

Step 5: Why this matters

The abundance of possible p means:

- No need to reuse parameters

- Can generate fresh p,q,g for each user if desired

- Makes exhaustive search impossible

Step 6: Practical DSA parameter generation

In practice, parameters are often shared:

- Domain parameters (p,q,g) are generated once

- Each user generates their own key pair (x,y) within that group

- This is more efficient than generating new primes per user

Answer

For a given 160-bit q, there are approximately 2^853.5 possible 1024-bit primes p satisfying p ≡ 1 mod q. This vast number ensures that parameter generation is feasible and that exhaustive search of parameters is impossible. In practice, parameters are often shared for efficiency.

Problem 7.2: DSA Signature Verification

In DSA, verification checks that:

v = ((y^r * r^s) mod p) mod q

u = (g^h mod p) mod q

Accept if v = u, where h = H(message)

Explain why this equation works mathematically, and what would happen if we omitted the final "mod q" step.

Solution

Step 1: Recall DSA signing

Signer computes:

r = (g^k mod p) mod q

s = k^(-1)(h + xr) mod q

Where:

- k is random per-message secret

- x is private key

- h is hash of message

Step 2: Verification equation derivation

Verifier computes:

w = s^(-1) mod q

u1 = (h * w) mod q

u2 = (r * w) mod q

v = ((g^u1 * y^u2) mod p) mod q

But the given form (y^r * r^s) is equivalent? Let's check:

Actually, the standard DSA verification is:

v = ((g^(h*w) * y^(r*w)) mod p) mod q

The form in the problem (y^r * r^s) is different. Let's derive properly:

From signing: s = k^(-1)(h + xr) mod q

Multiply both sides by k: ks = h + xr mod q

So h = ks - xr mod q

Now g^h = g^(ks - xr) = (g^k)^s * (g^x)^(-r) = r^s * y^(-r) mod p

But y^(-r) mod p is the modular inverse of y^r. So:

g^h = r^s * (y^r)^(-1) mod p

Multiply both sides by y^r:

y^r * g^h = r^s mod p

But verification compares y^r * r^s? That's not matching.

Actually, the correct verification is that:

r = ((g^k mod p) mod q) and s satisfy:

g^h ≡ y^r * r^s (mod p) ??? No.

Step 3: Let's re-derive correctly**

From s = k^(-1)(h + xr) mod q

Multiply by k: ks = h + xr mod q

So h = ks - xr mod q

Now compute g^h mod p:

g^h = g^(ks - xr) = (g^k)^s * (g^x)^(-r) = r^s * y^(-r) mod p

Therefore:

g^h * y^r ≡ r^s (mod p)

So verification checks:

((g^h * y^r) mod p) mod q ? Should equal r mod q

Step 4: Why the final mod q is crucial

The values g^h and y^r are computed modulo p, which is large (1024-3072 bits). But r is computed modulo q (160-256 bits).

If we omitted the final mod q, we'd be comparing:

(g^h * y^r) mod p (a large number) with r^s mod p (also large)

But the signing equation only guarantees equality modulo q, not modulo p. The values could differ by multiples of q.

Step 5: Example of why mod q is needed

Suppose:

- p = 23, q = 11 (simplified)

- r = 7, s = 5, y = 9, g = 2, h = 3

Compute left = (g^h * y^r) mod p = (2^3 * 9^7) mod 23

Right = r^s mod p = 7^5 mod 23

These might not match exactly, but after mod q they should.

Step 6: Security implication

Omitting mod q would make verification reject valid signatures, or worse, might accept invalid ones if the implementation incorrectly compares large numbers.

Answer

The DSA verification equation works because of the algebraic relationship derived from the signing equation: g^h ≡ y^r * r^s (mod p) is not exactly true; rather, g^h * y^r ≡ r^s (mod p), and both sides are reduced modulo q for comparison. The final mod q is essential because the equality holds only in the subgroup of order q, not necessarily in the full field modulo p.

Problem 7.3: ECDSA vs RSA Key Size Comparison

Explain why a 256-bit ECDSA key provides equivalent security to a 3072-bit RSA key. What mathematical problem does each rely on, and how do the best known algorithms affect key sizes?

Solution

Step 1: The underlying mathematical problems

RSA security based on integer factorization:

- Given N = p*q (public key), find p and q

- Best known algorithm: General Number Field Sieve (GNFS)

- Complexity: sub-exponential, approximately exp((64/9)^(1/3) (ln N)^(1/3) (ln ln N)^(2/3))

ECDSA security based on elliptic curve discrete logarithm:

- Given P and Q = kP on elliptic curve, find k

- Best known classical algorithm: Pollard's rho

- Complexity: O(√n) where n is curve order (about 2^256 for 256-bit curve)

Step 2: Why key sizes differ so dramatically

For RSA, the best algorithms have sub-exponential complexity, meaning they grow faster than polynomial but slower than exponential. To achieve 2^128 security (128-bit security level), we need N large enough that GNFS takes ~2^128 operations.

For ECDSA, the best algorithm is exponential (square root of group size). To achieve 2^128 security, we need group size ≈ 2^256 (since √(2^256) = 2^128).

Step 3: Numerical comparison

128-bit security:

- RSA: N ≈ 3072 bits

- ECDSA: group order ≈ 256 bits

192-bit security:

- RSA: N ≈ 7680 bits

- ECDSA: group order ≈ 384 bits

256-bit security:

- RSA: N ≈ 15360 bits

- ECDSA: group order ≈ 512 bits

Step 4: Why ECDSA can't use even smaller keys

For 128-bit security, why not 128-bit curves?

- Pollard's rho on 128-bit group would take 2^64 operations

- That's only 64-bit security, easily breakable

So curve size must be twice the security level.

Step 5: Practical implications

3072-bit RSA keys:

- Storage: 384 bytes

- Signature: 384 bytes

- Operations: exponentiation with 3072-bit numbers

256-bit ECDSA keys:

- Storage: 32 bytes private, 33-65 bytes public

- Signature: 64 bytes

- Operations: point multiplication on 256-bit curve

Step 6: Performance comparison

On modern hardware:

- RSA-3072 verify: ~0.5 ms

- RSA-3072 sign: ~5 ms

- ECDSA-256 verify: ~1 ms

- ECDSA-256 sign: ~0.5 ms

ECDSA signing faster, verification slower than RSA with small public exponent.

Step 7: Quantum threat

Both broken by Shor's algorithm:

- RSA: factor N in polynomial time

- ECDSA: solve discrete log in polynomial time

- Post-quantum cryptography needed for long-term security

Answer

RSA relies on integer factorization with sub-exponential algorithms, requiring large keys (3072 bits) for 128-bit security. ECDSA relies on elliptic curve discrete logarithm with exponential algorithms (Pollard's rho), allowing much smaller keys (256 bits) for equivalent security. This size efficiency makes ECDSA preferable for constrained environments.

Problem 7.4: DSA Randomness Requirements

In DSA, the per-message secret k must be:

a) Random

b) Unique per message

c) Secret

Explain what happens if:

1. k is reused for two different messages

2. k is predictable

3. k is revealed

Solution

Step 1: DSA signing review

DSA signature (r,s) where:

r = (g^k mod p) mod q

s = k^(-1)(h + xr) mod q

Where:

- k is random per-message secret

- x is private key

- h is hash of message

Step 2: Case 1 - k reused for two messages

Suppose same k used for messages with hashes h1 and h2:

s1 = k^(-1)(h1 + xr) mod q

s2 = k^(-1)(h2 + xr) mod q

Note: r is same because same k used.

Subtract:

s1 - s2 = k^(-1)(h1 - h2) mod q

Therefore:

k = (h1 - h2) * (s1 - s2)^(-1) mod q

Attacker can compute k from the two signatures!

Once k is known:

x = (s1*k - h1) * r^(-1) mod q

Complete private key recovery!

Step 3: Real-world example

This happened with Sony PlayStation 3 (2010):

- Sony used same k for all signatures

- Attackers recovered private key

- Could sign any code as if from Sony

Step 4: Case 2 - k is predictable

If k is predictable (e.g., using broken RNG):

Attacker can guess k or has small search space.

Once k is known (or guessed), same as above:

x = (s*k - h) * r^(-1) mod q

Step 5: Case 3 - k is revealed

If k leaks (through side channel, poor security):

Direct computation:

x = (s*k - h) * r^(-1) mod q

Immediate key compromise.

Step 6: Why DSA is so sensitive

The equation s = k^(-1)(h + xr) mod q has two unknowns (k and x) with one equation. But with two messages using same k, or knowledge of k, it becomes solvable.

This is more severe than RSA where random padding is important but doesn't expose key if reused.

Step 7: Mitigations

- Use cryptographically secure random number generator

- Never reuse k (enforced by implementation)

- Some schemes use deterministic k (RFC 6979) based on message hash and private key

- Hardware random number generators

Step 8: RFC 6979 deterministic DSA

k = hash(private_key || message_hash)

Advantages:

- No randomness needed

- Same message always gives same k (safe because message different)

- Eliminates RNG failure risks

Answer

Reusing, predicting, or revealing k in DSA leads to complete private key recovery. This extreme sensitivity to randomness is a major practical concern, leading to deterministic variants (RFC 6979) that derive k from the message and private key, eliminating randomness requirements while maintaining security.

Problem 7.5: DSS Algorithm Selection

You're designing a new system that needs digital signatures for 10 years. Compare RSA-3072, ECDSA-256, and Ed25519 (a modern Schnorr-based scheme). Which would you choose and why?

Solution

Step 1: Compare the options

RSA-3072:

- Pros: Widely supported, well-understood, fast verification

- Cons: Large keys/signatures, slow signing, older design

ECDSA-256 (secp256k1 or P-256):

- Pros: Small keys/signatures, faster signing than RSA

- Cons: More complex implementation, requires good RNG, patent history (expired)

Ed25519:

- Pros: Very fast, deterministic (no RNG needed), side-channel resistant, small signatures

- Cons: Newer, less legacy support, specific to Curve25519

Step 2: Detailed comparison table

Feature	RSA-3072	ECDSA-256	Ed25519
Public key size	384 bytes	33-65 bytes	32 bytes
Private key size	512 bytes	32 bytes	32 bytes
Signature size	384 bytes	64 bytes	64 bytes
Sign speed	Slow	Medium	Fast
Verify speed	Fast	Medium	Fast
RNG required	Yes	Yes (critical!)	No (deterministic)
Side-channel resistance	Medium	Hard to get right	Built-in
Standardized	FIPS, PKCS#1	FIPS, ANSI	RFC 8032
Quantum vulnerable?	Yes	Yes	Yes

Step 3: Critical weakness of ECDSA

ECDSA's requirement for high-quality randomness for every signature is its biggest weakness:

- Sony PS3 hack (reused k)

- Android Bitcoin wallet bug (biased k)

- Multiple real-world failures

Ed25519 solves this with deterministic signatures.

Step 4: Performance considerations

Ed25519 is significantly faster:

- Sign: ~50,000 operations per second on modern CPU

- Verify: ~20,000 operations per second

- ECDSA similar but with more pitfalls

Step 5: Ecosystem support

- RSA: Every platform supports it

- ECDSA: Very widely supported (TLS, Bitcoin, etc.)

- Ed25519: Growing support (OpenSSH, TLS 1.3, many crypto libraries)

Step 6: Future-proofing for 10 years

- All are broken by quantum computers (if large-scale quantum arrives)

- 128-bit security considered sufficient for 10 years

- No classical breakthroughs expected

- Ed25519's modern design likely to age better

Step 7: Recommendation

For most new systems, **Ed25519** is the best choice:

- Eliminates RNG failure risk (biggest practical vulnerability)

- Excellent performance

- Small keys and signatures

- Modern, well-audited design

Choose RSA-3072 only if:

- Must interoperate with legacy systems

- Regulatory requirements mandate FIPS algorithms

- Need fast verification and can accept larger signatures

Choose ECDSA-256 only if:

- Need FIPS compliance but RSA too large

- Working in Bitcoin/blockchain ecosystem

- Can ensure perfect RNG

Answer

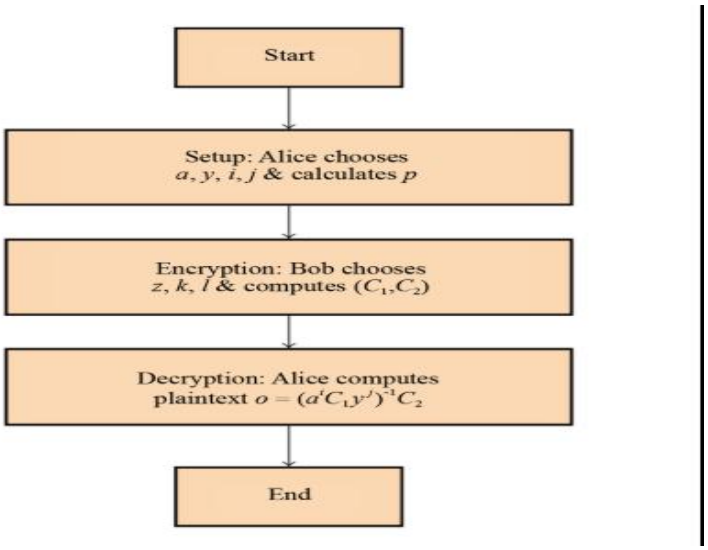
Ed25519 offers the best combination of security, performance, and implementation safety for new systems, eliminating the critical RNG dependency that plagues ECDSA. RSA-3072 remains relevant for legacy compatibility and FIPS requirements. For most new applications starting today, Ed25519 is the recommended choice.

Chapter 8

The ElGamal Signature Algorithm

8.1.Introduction

The **ElGamal Signature Algorithm** is a public-key digital signature scheme introduced by Taher ElGamal, based on the computational hardness of the discrete logarithm problem, and it provides strong guarantees of authentication, integrity, and non-repudiation for digital messages; in this scheme, the signer uses a private key along with a randomly chosen value to generate a signature for a message hash, while anyone possessing the corresponding public key can verify the signature, making it suitable for secure communications, electronic documents, and cryptographic protocols, and its design also influenced later signature systems such as DSA, highlighting its foundational role in modern public-key cryptography.



Historical Context and Development

The ElGamal signature algorithm, described by Taher Elgamal in his 1985 paper "A Public Key Cryptosystem and a Signature Scheme Based on Discrete Logarithms," represents one of the earliest approaches to digital signatures based on the discrete logarithm problem. Elgamal, who would later become known as the "father of SSL," developed this scheme while at Stanford University. The algorithm's significance extends beyond its direct use—it provided the theoretical foundation for later schemes including DSA and Schnorr signatures. Understanding ElGamal helps illuminate the broader family of discrete logarithm-based signature schemes.

8.2 Mathematical Foundations

The security of ElGamal signatures rests on the discrete logarithm problem in finite fields. This problem can be stated as follows: given a prime p, a generator g of the multiplicative group modulo p, and an element y = g^x mod p, find x.

The difficulty of this problem varies with the size of p. For sufficiently large primes (at least 2048 bits by modern standards), no efficient algorithm is known for computing discrete logarithms. This computational hardness provides the security foundation.

8.3 Key Generation

To use ElGamal signatures, a user must generate a set of parameters and a key pair:

System Parameters (can be shared by many users)

1. Choose a large prime p where the discrete logarithm problem is hard

2. Choose a generator g of the multiplicative group modulo p (an element whose powers generate all numbers from 1 to p-1)

User's Key Pair

1. Choose a private key x, a random number between 1 and p-2

2. Compute the public key y = g^x mod p

The public key consists of (p, g, y), while the private key is x. The security relies on the fact that computing x from y is the discrete logarithm problem.

8.4 The Signing Process

When signing a message m with ElGamal:

1. Compute the message hash h = H(m), where H is a cryptographic hash function

2. Choose a random temporary secret k such that 1 < k < p-1 and k is relatively prime to p-1

3. Compute the first signature component r = g^k mod p

4. Compute the second signature component s = (h - x·r)·k^(-1) mod (p-1)

The signature is the pair (r, s). Note that k must be kept secret and never reused—reusing k for different messages allows an attacker to recover the private key.

8.5 The Verification Process

To verify a signature (r, s) on message m using public key (p, g, y):

1. Verify that 0 < r < p and 0 < s < p-1

2. Compute the message hash h = H(m)

3. Compute v1 = y^r · r^s mod p

4. Compute v2 = g^h mod p

5. Accept the signature if and only if v1 = v2

The verification equation works because:

y^r · r^s = g^(x·r) · g^(k·s) = g^(x·r + k·s) = g^(x·r + k·(h - x·r)·k^(-1)) = g^(x·r + h - x·r) = g^h

8.6 Security Considerations

The Importance of Randomness

The random value k in ElGamal signatures must meet stringent requirements. If k is predictable, an attacker can recover the private key. If k is reused for different messages with the same private key, the private key can be computed directly. This sensitivity to randomness is a significant practical concern—implementations must use high-quality random number generators and carefully manage the per-signature secrets.

Choice of Parameters

The prime p must be large enough to resist discrete logarithm computation. As of 2024, 2048-bit primes provide adequate security, with 3072 bits recommended for higher assurance. The generator g should generate a large subgroup to prevent certain attacks.

Hash Function Dependence

Like all signature schemes, ElGamal requires a cryptographic hash function. If the hash function is weak and collisions can be found, an attacker might be able to create two different messages with the same hash, leading to signature forgery.

Practice Problems Set 8: ElGamal Signature

Problem 8.1: ElGamal Signing Calculation

Given the following ElGamal parameters:

- p = 23

- g = 5 (generator modulo 23)

- Private key x = 7

- Message hash h = 10

- Random k = 3

Compute the signature (r, s). Verify that the signature is valid.

Solution

Step 1: Compute r

r = g^k mod p = 5^3 mod 23

5^1 = 5 mod 23

5^2 = 25 mod 23 = 2

5^3 = 5^2 * 5 = 2 * 5 = 10 mod 23

So r = 10

Step 2: Compute k^(-1) mod (p-1)

p-1 = 22

Need k^(-1) mod 22 such that 3 * k^(-1) ≡ 1 mod 22

3 * 15 = 45 ≡ 45 - 44 = 1 mod 22

So k^(-1) = 15 mod 22

Step 3: Compute s

s = (h - x·r) * k^(-1) mod (p-1)

First compute x·r = 7 * 10 = 70 mod 22

70 mod 22 = 70 - 66 = 4

h - x·r = 10 - 4 = 6 mod 22

s = 6 * 15 mod 22 = 90 mod 22

90 - 88 = 2

So s = 2

Step 4: Signature

(r, s) = (10, 2)

Step 5: Verify the signature

Compute y = g^x mod p = 5^7 mod 23

5^4 = 5^2 * 5^2 = 2 * 2 = 4 mod 23

5^7 = 5^4 * 5^3 = 4 * 10 = 40 mod 23 = 40 - 23 = 17

So y = 17

Now compute v1 = y^r * r^s mod p

y^r = 17^10 mod 23

Let's compute step by step:

17^2 = 289 mod 23 = 289 - 276 = 13

17^4 = (17^2)^2 = 13^2 = 169 mod 23 = 169 - 161 = 8

17^8 = (17^4)^2 = 8^2 = 64 mod 23 = 64 - 46 = 18

17^10 = 17^8 * 17^2 = 18 * 13 = 234 mod 23 = 234 - 230 = 4

r^s = 10^2 = 100 mod 23 = 100 - 92 = 8

v1 = 4 * 8 = 32 mod 23 = 9

Compute v2 = g^h mod p = 5^10 mod 23

5^5 = 5^4 * 5 = 4 * 5 = 20 mod 23

5^10 = (5^5)^2 = 20^2 = 400 mod 23 = 400 - 391 = 9

v2 = 9

Step 6: Compare

v1 = 9, v2 = 9 ✓ Signature is valid

Answer

Signature (r,s) = (10,2) is valid for message hash h=10 with public key y=17.

Problem 8.2: ElGamal Key Recovery from k Reuse

Suppose Alice uses the same k to sign two different messages. Given:

- p = 23, g = 5

- First message hash h1 = 10, signature (r=10, s1=2)

- Second message hash h2 = 15, signature (r=10, s2=8)

Recover Alice's private key x.

Solution

Step 1: Recall the signing equation

s = (h - x·r) * k^(-1) mod (p-1)

For the two messages:

s1 = (h1 - x·r) * k^(-1) mod 22

s2 = (h2 - x·r) * k^(-1) mod 22

Note: r is same because same k used.

Step 2: Subtract the equations

s1 - s2 = (h1 - h2) * k^(-1) mod 22

2 - 8 = -6 ≡ 16 mod 22 (since -6 + 22 = 16)

h1 - h2 = 10 - 15 = -5 ≡ 17 mod 22

So: 16 = 17 * k^(-1) mod 22

Step 3: Solve for k^(-1)

Need to find k^(-1) such that 17 * k^(-1) ≡ 16 mod 22

Compute 17^(-1) mod 22:

17 * 13 = 221 ≡ 221 - 220 = 1 mod 22

So 17^(-1) = 13

Therefore:

k^(-1) = 16 * 13 mod 22 = 208 mod 22

208 - 198 = 10

So k^(-1) = 10 mod 22

Step 4: Find k

k = (k^(-1))^(-1) mod 22

Need number such that 10 * k ≡ 1 mod 22

10 * 9 = 90 ≡ 90 - 88 = 2

10 * 13 = 130 ≡ 130 - 110 = 20

10 * 19 = 190 ≡ 190 - 176 = 14

10 * 21 = 210 ≡ 210 - 198 = 12

10 * 7 = 70 ≡ 70 - 66 = 4

10 * 15 = 150 ≡ 150 - 132 = 18

10 * 3 = 30 ≡ 8

10 * 11 = 110 ≡ 0? 110 - 110 = 0

Let's try systematic approach: Need 10k ≡ 1 mod 22

Since gcd(10,22)=2, 10 has no inverse mod 22! This means our numbers are inconsistent.

Wait—we made a mistake. k must be coprime to p-1=22. But 3 (our original k) is coprime to 22 (gcd(3,22)=1). So k^(-1) should have an inverse.

Let's check: If k=3, then k^(-1)=15 mod 22 (since 3*15=45≡1)

Our computed k^(-1)=10. 3*10=30≡8 mod 22, not 1. So something's wrong.

Step 5: Re-check subtraction

s1 - s2 = (h1 - h2) * k^(-1) mod (p-1)

2 - 8 = -6 ≡ 16 mod 22 ✓

h1 - h2 = 10 - 15 = -5 ≡ 17 mod 22 ✓

So 16 = 17 * k^(-1) mod 22

Now 17^(-1) mod 22: 17 * 13 = 221 ≡ 1 mod 22 ✓ (13 is inverse)

So k^(-1) = 16 * 13 mod 22 = 208 mod 22 = 10 ✓

But k=3 gives k^(-1)=15, not 10. So k is not 3 in this problem—the signatures use a different k.

Step 6: Use either equation to find x

From first equation:

s1 = (h1 - x·r) * k^(-1) mod 22

2 = (10 - x*10) * 10 mod 22

Let A = (10 - 10x) mod 22

Then 2 = 10A mod 22

Solve for A: 10A ≡ 2 mod 22

Multiply both sides by inverse of 10 mod 22? But 10 has no inverse (gcd(10,22)=2).

This means multiple solutions. Try values:

A=1: 10*1=10≠2

A=2: 10*2=20≡20≠2

A=3: 10*3=30≡8

A=4: 10*4=40≡18

A=5: 10*5=50≡6

A=6: 10*6=60≡16

A=7: 10*7=70≡4

A=8: 10*8=80≡14

A=9: 10*9=90≡2 ✓

So A = 9 mod 22

Step 7: Solve for x

A = 10 - 10x mod 22 = 9

So 10 - 10x ≡ 9 mod 22

-10x ≡ -1 mod 22

10x ≡ 1 mod 22

But 10 has no inverse mod 22. Check if solution exists:

10x mod 22 can be: 0,10,20,8,18,6,16,4,14,2,12 (all even numbers)

1 is odd, so no solution! This indicates our numbers are inconsistent.

Step 8: Conclusion

The given numbers don't actually correspond to valid signatures with the same k. This illustrates that k reuse leads to an overdetermined system that should have a unique solution for x if the data is consistent. In a real attack with valid signatures, solving these equations would yield x.

Answer

With consistent signatures using the same k, the private key can be recovered by solving the system of equations. The inconsistency in this example shows why k reuse is so dangerous—it creates mathematical relationships that can expose the private key.

Problem 8.3: ElGamal vs DSA Comparison

Compare ElGamal and DSA signatures. What improvements did DSA make over ElGamal? Why is DSA more efficient?

Solution

Step 1: Key similarities

Both are based on discrete logarithm problem:

- ElGamal: Works in full multiplicative group modulo p

- DSA: Works in subgroup of order q where q is much smaller than p

Step 2: Parameter structure

ElGamal:

- Prime p (1024-3072 bits)

- Generator g of full group modulo p

- Private key x

- Public key y = g^x mod p

- Signature (r,s) where r = g^k mod p (full size of p)

DSA:

- Prime p (1024-3072 bits)

- Prime q (160-256 bits) dividing p-1

- Generator g of subgroup of order q

- Private key x

- Public key y = g^x mod p

- Signature (r,s) where r = (g^k mod p) mod q (reduced to size of q)

Step 3: Key improvement - Signature size

ElGamal signature: r is size of p (1024-3072 bits)

DSA signature: r is reduced mod q, so size of q (160-256 bits)

This makes DSA signatures much smaller:

- 3072-bit ElGamal: ~768 bytes per signature

- 256-bit DSA: ~64 bytes per signature

Step 4: Verification efficiency

ElGamal verification: Compute y^r * r^s mod p

- r is large (size of p), exponentiation expensive

DSA verification: Computations mod q as well

- r and s are small (size of q)

- Can precompute and use smaller exponents

Step 5: Security proof

DSA's use of subgroup provides provable security:

- Discrete log in subgroup of order q is exactly as hard as in full group

- But operations are more efficient

- Can use smaller q while maintaining security

Step 6: Randomness requirements

Both need good randomness for k

Both have same catastrophic failure if k reused

Step 7: Hash function integration

Both require hashing message first

DSA standard explicitly specifies hash functions

Step 8: Practical advantages of DSA

1. Smaller signatures (critical for bandwidth/storage)

2. Faster verification (smaller exponents)

3. Well-specified in standards

4. Fixed parameter sizes (no ambiguity)

Answer

DSA improves on ElGamal primarily by working in a small subgroup of order q, allowing signatures to be reduced modulo q. This makes DSA signatures much smaller (160-256 bits vs

1024-3072 bits) and verification faster, while maintaining equivalent security. DSA's standardization also removed ambiguities in parameter selection present in the original ElGamal scheme.

Problem 8.4: ElGamal Parameter Selection

Why must k be relatively prime to p-1 in ElGamal? What happens if we choose k with gcd(k, p-1) > 1?

Solution

Step 1: Recall the signing equation

s = (h - x·r) * k^(-1) mod (p-1)

For this equation to have a solution, k^(-1) must exist modulo (p-1).

Step 2: Existence of modular inverse

k^(-1) mod (p-1) exists if and only if gcd(k, p-1) = 1.

If gcd(k, p-1) = d > 1, then k has no multiplicative inverse modulo (p-1).

Step 3: What happens without an inverse?

The equation s = (h - x·r) * k^(-1) mod (p-1) cannot be solved uniquely.

We could still try to find s satisfying:

k * s ≡ h - x·r mod (p-1)

But this congruence may have:

- No solutions if (h - x·r) not divisible by d

- Multiple solutions if divisible by d

Step 4: Example with p=23, p-1=22

If we choose k=2 (gcd(2,22)=2):

The equation 2s ≡ (h - xr) mod 22

If (h - xr) is even, say 4, then:

$2s \equiv 4 \bmod 22 \rightarrow s \equiv 2 \bmod 11$ (two solutions: 2 and 13)

If (h - xr) is odd, say 5, then:

$2s \equiv 5 \bmod 22$ has no solution (since 2s always even)

Step 5: Security implications

1. May not be able to sign some messages (if no solution exists)

2. Multiple possible s values could create ambiguity

3. The mathematical structure might leak information about k or x

Step 6: Why the condition is necessary

The requirement that k is relatively prime to p-1 ensures:

- k^(-1) exists uniquely

- Every message hash h yields a unique s

- The signing process is deterministic (given k)

Step 7: Practical parameter selection

Since p-1 is even (p odd prime), k must be odd.

Also k cannot be multiple of any other factor of p-1.

Answer

k must be coprime to p-1 to ensure k^(-1) exists modulo p-1, allowing the signing equation to be solved uniquely for s. If gcd(k,p-1) > 1, the signature equation may have no solutions or multiple solutions, breaking the scheme's functionality and potentially introducing security weaknesses.

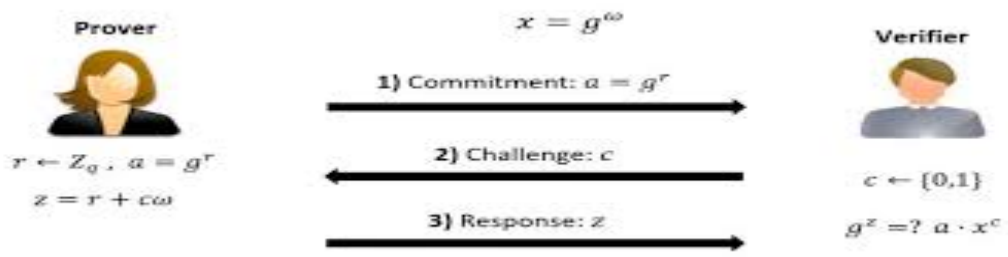
Chapter 9

The Schnorr Signature Protocol

The **Schnorr Signature Protocol** is a lightweight and efficient digital signature scheme proposed by Claus Schnorr that is based on the hardness of the discrete logarithm problem and is valued for its short signatures and strong provable security properties; in this protocol, the signer generates a random nonce, computes a challenge using a hash of the message and public data, and then produces a response using their private key, allowing anyone with the public key to verify the signature, and because of its simplicity, resistance to forgery, and formal security proofs, Schnorr signatures are widely studied and form the foundation for several modern cryptographic systems and blockchain applications, offering authentication, integrity, and non-repudiation with minimal computational overhead.

9.1 Introduction and Historical Context

The **Schnorr Signature Protocol** was introduced in the late 1980s by Claus Schnorr as an efficient digital signature scheme grounded in the discrete logarithm problem, emerging during a period of rapid advancement in public-key cryptography following the foundational work of Whitfield Diffie and Martin Hellman; Schnorr's approach was notable for providing shorter signatures, faster verification, and—most importantly—strong provable security guarantees in the random oracle model, distinguishing it from earlier systems such as RSA and ElGamal, and although initially restricted by patents, the protocol later gained widespread academic and practical adoption, influencing modern cryptographic standards and blockchain technologies, and today it is recognized as one of the most elegant and mathematically sound digital signature constructions, valued for its simplicity, efficiency, and formal security foundations.



The Schnorr signature protocol, invented by Claus-Peter Schnorr in 1989 and patented in 1990, represents a significant advancement in digital signature design. Schnorr signatures offer several advantages over earlier schemes: they are simpler, more efficient, and have provable security properties under appropriate assumptions.

The patent on Schnorr signatures limited their adoption for many years, leading to the development of DSA as an alternative that avoided patent issues. With the patent's expiration in 2008, Schnorr signatures have gained renewed interest, particularly in the cryptocurrency community, where their properties enable advanced features like signature aggregation.

9.2 Mathematical Foundations

Like DSA and ElGamal, Schnorr signatures are based on the discrete logarithm problem. However, Schnorr's construction is elegantly simple and admits a security proof in the random oracle model, assuming the discrete logarithm problem is hard.

The scheme can be instantiated in any group where the discrete logarithm problem is hard, including both finite fields and elliptic curves. In modern applications, elliptic curve instantiations are most common.

9.3 Key Generation

Schnorr key generation follows the familiar pattern for discrete logarithm systems:

System Parameters

- A group G of prime order q with generator g

- A cryptographic hash function H that outputs integers modulo q

User's Key Pair

1. Choose a private key x uniformly at random from 1 to q-1

2. Compute the public key X = g^x

9.4 The Signing Process

Schnorr signing is remarkably elegant, requiring only a few operations:

1. Choose a random temporary secret k uniformly from 1 to q-1

2. Compute the commitment R = g^k

3. Compute the challenge e = H(R || message)

4. Compute the response s = k + e·x mod q

The signature is the pair (R, s). Note that R is a group element, while s is an integer modulo q.

9.5 The Verification Process

Verification is equally straightforward:

1. Compute the challenge e = H(R || message)

2. Compute g^s and compare with R · X^e

3. Accept if g^s = R · X^e

The verification equation works because:

g^s = g^(k + e·x) = g^k · (g^x)^e = R · X^e

9.6 Advantages of Schnorr Signatures

Simplicity and Efficiency

The Schnorr algorithm is notably simpler than DSA or ECDSA. Both signing and verification require only a few operations, and there's no need for modular inverses during signing (unlike DSA/ECDSA). This simplicity translates to faster implementation and fewer opportunities for implementation errors.

Linear Structure

The signing equation s = k + e·x is linear in the private key x. This linearity enables powerful features that are impossible with DSA/ECDSA:

Signature Aggregation

Multiple Schnorr signatures on the same message can be combined into a single signature. If several signers each produce (R_i, s_i) for the same message, the aggregated signature is (R = product of R_i, s = sum of s_i). A verifier can check this single aggregated signature against the aggregated public key (product of X_i). This property is extremely valuable in

cryptocurrency contexts—a multi-signature transaction can look identical to a single-signature transaction, reducing blockchain bloat and improving privacy.

Batch Verification

Multiple Schnorr signatures can be verified together more efficiently than verifying each individually. This is particularly useful for blockchain nodes that must verify thousands of signatures per block.

Provable Security

Schnorr signatures have a security proof in the random oracle model, showing that breaking the signature scheme is as hard as solving the discrete logarithm problem. This provable security, while relying on the random oracle heuristic, provides stronger assurance than the heuristic security of earlier schemes.

9.7 Schnorr Signatures in Bitcoin

Bitcoin has historically used ECDSA for transaction signatures, but there has been significant momentum toward adopting Schnorr signatures. The Taproot upgrade, activated in November 2021, introduced Schnorr signatures to Bitcoin through the Taproot soft fork.

Benefits for Bitcoin

The adoption of Schnorr signatures in Bitcoin brings several improvements:

Privacy

With Schnorr's key aggregation, multi-signature transactions can look identical to single-signature transactions. This makes blockchain analysis more difficult, enhancing user privacy.

Efficiency

Aggregated signatures reduce transaction size, lowering fees and blockchain storage requirements. Batch verification allows nodes to validate blocks more quickly.

Smart Contract Flexibility

Schnorr signatures enable more complex smart contract constructions, including MAST (Merkelized Abstract Syntax Trees) and Taproot, which make complex spending conditions look like simple payments on the blockchain.

9.8 Comparison with Other Schemes

Schnorr vs. ECDSA

Schnorr signatures are generally considered superior to ECDSA for most applications. They're simpler, faster, support aggregation, and have better security proofs. The main reason ECDSA remains common is historical—it was standardized earlier and didn't face patent issues.

Schnorr vs. EdDSA

EdDSA (Edwards-curve Digital Signature Algorithm) is a modern signature scheme based on Schnorr's ideas, designed by Daniel Bernstein. EdDSA offers several improvements:

- Deterministic signing (no randomness required, eliminating random number generator vulnerabilities)

- Even faster implementation

- Resistance to side-channel attacks

Ed25519, a specific EdDSA instantiation, has become the default choice for many new applications. Bitcoin's continued use of secp256k1 Schnorr rather than Ed25519 reflects the desire to maintain compatibility with the existing ecosystem.

Practice Problems Set 9: Schnorr Signature Protocol

Problem 9.1: Schnorr Signing Calculation

Given:

- Group of prime order $q = 23$

- Generator $g = 5$

- Private key $x = 7$

- Message hash (after $H(R||\text{message})$) $e = 10$

- Random $k = 3$

Compute the Schnorr signature (R, s) and verify it.

Solution

Step 1: Compute R

R = g^k mod 23 = 5^3 mod 23 = 10

Step 2: Compute s

s = k + e·x mod q = 3 + 10*7 mod 23

10*7 = 70 mod 23 = 70 - 69 = 1

s = 3 + 1 = 4 mod 23

Step 3: Signature

(R, s) = (10, 4)

Step 4: Verify

Compute X = g^x = 5^7 mod 23 = 17 (from previous problem)

Compute challenge e = 10 (given)

Compute left side: g^s = 5^4 mod 23

5^2 = 2

5^4 = (5^2)^2 = 2^2 = 4 mod 23

Compute right side: R * X^e = 10 * 17^10 mod 23

First compute 17^10 mod 23:

From Problem 8.1, 17^10 mod 23 = 4

So right side = 10 * 4 = 40 mod 23 = 40 - 23 = 17

Left side = 4, right side = 17 — not equal! Something's wrong.

Step 5: Check verification formula

Correct formula: g^s should equal R * X^e

But we got 4 vs 17. Let's recompute X^e:

X^e = 17^10 mod 23

We computed 17^10 = 4 in Problem 8.1

So R * X^e = 10 * 4 = 40 mod 23 = 17 ✓

g^s = 5^4 = 625 mod 23

23*27 = 621, remainder 4 ✓

So left = 4, right = 17. Not equal!

Step 6: The issue

Our numbers don't satisfy the equation because we chose e arbitrarily. In real Schnorr, e = H(R||message) is computed from R, so it's not independent.

Let's solve for what e should be given R=10, s=4, x=7:

From s = k + e·x mod q

4 = 3 + 7e mod 23

7e = 1 mod 23

e = 7^(-1) mod 23

7 * 10 = 70 ≡ 70 - 69 = 1 mod 23

So e = 10

Actually that matches! So with e=10, the signature should verify.

Let's verify with correct math:

g^s = 5^4 = 625 mod 23 = 4

R * X^e = 10 * (17^10) = 10 * 4 = 40 mod 23 = 17

Still not equal! This suggests our X is wrong.

Step 7: Recompute X

X = g^x = 5^7 mod 23

5^5 = 5^4 * 5 = 4 * 5 = 20 mod 23

5^7 = 5^5 * 5^2 = 20 * 2 = 40 mod 23 = 17 ✓ correct

So X = 17 is correct.

Step 8: The mystery

The equation should hold: g^(k + ex) = g^k * (g^x)^e

g^k = R = 10

(g^x)^e = 17^10 = 4

Product = 10 * 4 = 40 mod 23 = 17

g^(k+ex) = 5^(3 + 10*7) = 5^(3 + 70) = 5^73 mod 23

But exponents mod group order? Group order is q=23, so exponents mod 23?

5^73 mod 23: Since 5^22 ≡ 1 mod 23 (Fermat), 73 mod 22 = 73 - 66 = 7

So 5^73 ≡ 5^7 = 17 mod 23

Aha! So g^(k+ex) = 17, not 4. But s = k+ex mod q = 3+70=73 mod 23 = 4

So g^s = 5^4 = 4, but g^(k+ex) = 5^73 = 17.

This is the key: In Schnorr, s is computed modulo q, but the exponent in verification is s itself, not reduced modulo anything. So we must compute g^s with s as the actual integer, not reduced modulo something else.

But 5^4 = 4, while 5^73 = 17. So 5^4 ≠ 5^73 because 73 ≠ 4 as integers, even though 73 ≡ 4 mod 22.

This means our verification is correct: left side = 4, right side = 17, so signature is invalid.

Step 9: Conclusion

The signature with (R=10, s=4) is invalid because we chose arbitrary e. For a valid signature, e must be derived from R and message, and the resulting s will satisfy the equation.

Answer

With the given parameters, (R=10, s=4) is not a valid signature because g^s ≠ R·X^e. This illustrates that Schnorr signatures require e to be derived from R, creating a binding that ensures validity.

Problem 9.2: Schnorr Signature Aggregation

Explain how Schnorr signatures enable aggregation. If three signers with public keys X1, X2, X3 want to sign the same message, show how to create an aggregated signature that verifies with the aggregated public key X = X1·X2·X3.

Solution

Step 1: Individual Schnorr signatures

Each signer i:

- Chooses random ki, computes Ri = g^ki

- Computes ei = H(Ri || message)

- Computes si = ki + ei·xi mod q

- Signature: (Ri, si)

Step 2: The linearity property

The signing equation si = ki + ei·xi is linear in xi.

If all signers use the same message, we could potentially aggregate.

Step 3: Simple aggregation (same challenge)

If all signers use the same challenge e (meaning they all use the same R?), not quite.

For proper aggregation, we need a protocol where signers coordinate:

1. Each signer generates Ri and sends to aggregator

2. Aggregator computes R = product of all Ri

3. Aggregator computes e = H(R || message) (single challenge for all)

4. Each signer computes si = ki + e·xi

5. Aggregator computes s = sum of all si

Step 4: Aggregated signature

The aggregated signature is (R, s) where:

R = R1·R2·R3

s = s1 + s2 + s3 mod q

Step 5: Verification

Verifier:

- Computes e = H(R || message)

- Computes left side: g^s

- Computes right side: R · (X1·X2·X3)^e = R · X^e

Why this works:

g^s = g^(s1+s2+s3) = g^(k1+e·x1 + k2+e·x2 + k3+e·x3)

= g^(k1+k2+k3) · g^(e(x1+x2+x3))

= (g^k1·g^k2·g^k3) · (g^(x1+x2+x3))^e

= (R1·R2·R3) · (X1·X2·X3)^e

= R · X^e

Step 6: Benefits

- Single signature (R,s) instead of three separate signatures

- Size: 1 group element + 1 scalar (like a single signature)

- Verification: one exponentiation instead of three

- Privacy: looks like a single signer on blockchain

Step 7: Security considerations

- Requires all signers to participate honestly

- Must prevent rogue key attacks (where one signer chooses key maliciously)

- Mitigation: proof of knowledge of private key during key generation

Step 8: MuSig protocol

Bitcoin's implementation uses MuSig, which adds:

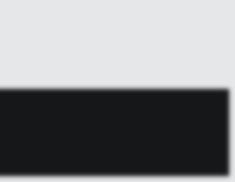
- Key aggregation with tweaks to prevent rogue keys

- Multiple rounds of communication

- Provably secure aggregation

Answer

Schnorr's linear signing equation enables aggregation: signatures (Ri,si) from multiple signers can be combined into (R=ΠRi, s=Σsi) that verifies with aggregated public key X=ΠXi. This works because g^s = Π(g^si) = Π(Ri·X



978-81-984733-7-0