Computer and Network Security (2023-2024)

Part 3: Cryptographic Hash Functions

Jeroen Famaey
jeroen.famaey@uantwerpen.be

University of Antwerp
Faculty of Science

Hash Function Basics

University of Antwerp
Faculty of Science

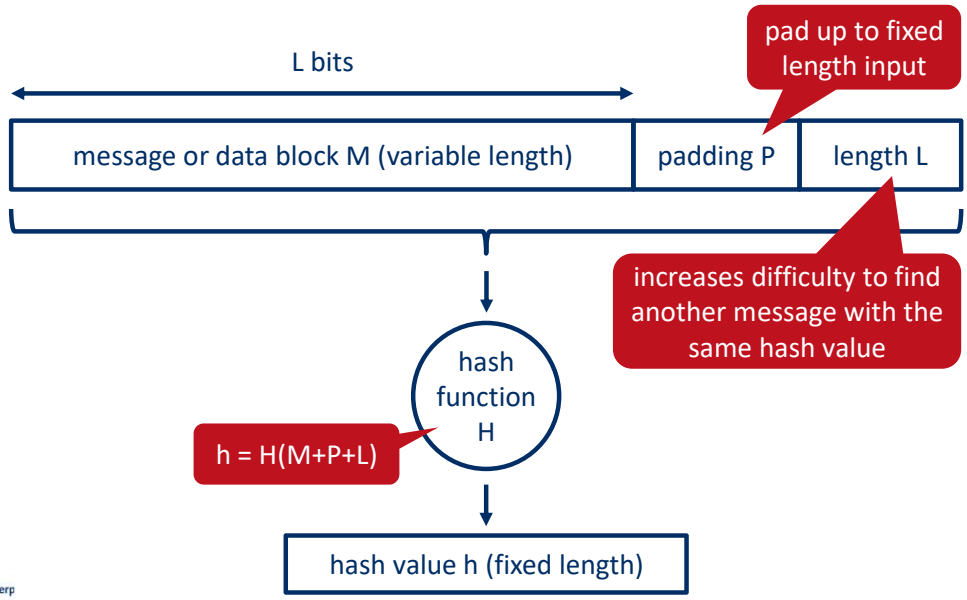
117 million LinkedIn password hashes leaked in 2016

- The passwords were hashed using unsalted SHA-1
- It took hackers a mere 2 hours on a computing cluster to hack 65%

Frequency	Hash	Plaintext
1135936	7c4a8d09ca3762af61e59520943dc26494f8941b	123456
207488	7728240c80b6bfd450849405e8500d6d207783b6	linkedin
188380	5baa61e4c9b93f3f0682250b6cf8331b7ee68fd8	password
149916	f7c3bc1d808e04732adf679965ccc34ca7ae3441	123456789
95854	7c222fb2927d828af22f592134e8932480637c0d	12345678

Source: *https://blog.korelogic.com/blog/2016/05/19/linkedin_passwords_2016*

What is a hash function?



University of Antwerp
Faculty of Science

4

A **hash function** H accepts a variable-length block of data *M* as input and produces a fixed-size hash value *h* = H(*M*). A "good" hash function has the property that the results of applying the function to a large set of inputs will produce outputs that are evenly distributed and apparently random. In general terms, the principal object of a hash function is data integrity. A change to any bit or bits in *M* results, with high probability, in a change to the hash value.

The kind of hash function needed for security applications is referred to as a **cryptographic hash function**. A cryptographic hash function is an algorithm for which it is computationally infeasible (because no attack is significantly more efficient than brute force) to find either (a) a data object that maps to a pre-specified hash result (the one-way property) or (b) two data objects that map to the same hash result (the collision-free property). Because of these characteristics, hash functions are often used to determine whether or not data has changed.

The figure depicts the general operation of a cryptographic hash function. Typically, the input is padded out to an integer multiple of some fixed length (e.g., 1024 bits), and the padding includes the value of the length of the original message in bits. The length field is a security measure to increase the difficulty for an attacker to produce an alternative message with the same hash value, as explained subsequently.

Applications of cryptographic hash functions

message authentication	Assures that the data received was not altered or replayed
digital signatures	Assures both the authenticity of the message and identity of the sender
one-way password file	Stores a hash value of the password rather than the password itself
intrusion/virus detection	Store H(F) for each file to determine if files have been modified
pseudorandom function	Use the hash function to generate pseudorandom private keys

University of Antwerp | Faculty of Science

5

Message authentication is a mechanism or service used to verify the integrity of a message. Message authentication assures that data received are exactly as sent (i.e., there is no modification, insertion, deletion, or replay). In many cases, there is a requirement that the authentication mechanism assures that purported identity of the sender is valid. When a hash function is used to provide message authentication, the hash function value is often referred to as a **message digest**. The essence of the use of a hash function for message integrity is as follows. The sender computes a hash value as a function of the bits in the message and transmits both the hash value and the message. The receiver performs the same hash calculation on the message bits and compares this value with the incoming hash value. If there is a mismatch, the receiver knows that the message (or possibly the hash value) has been altered. The hash value must be transmitted in a secure fashion. That is, the hash value must be protected so that if an adversary alters or replaces the message, it is not feasible for adversary to also alter the hash value to fool the receiver.

Another important application, which is similar to the message authentication application, is the **digital signature**. The operation of the digital signature is similar to that of the MAC. In the case of the digital signature, the hash value of a message is encrypted with a user's private key. Anyone who knows the user's public key can verify the integrity of the message that is associated with the digital signature. In this case, an attacker who wishes to alter the message would need to know the user's private key.

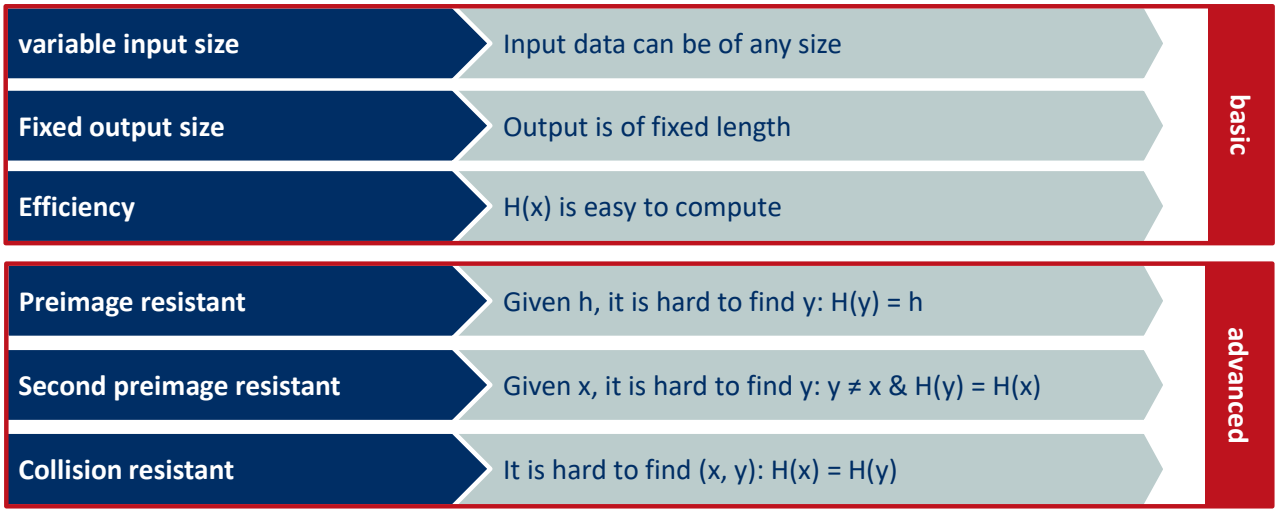
Hash functions are commonly used to create a **one-way password file**. A hash of a password is stored by an operating system rather than the password itself. Thus, the actual password is not retrievable by a hacker who gains access to the password file. In simple terms, when a

user enters a password, the hash of that password is compared to the stored hash value for verification. This approach to password protection is used by most operating systems.

Hash functions can be used for **intrusion detection** and **virus detection**. Store H(F) for each file on a system and secure the hash values (e.g., on a CD-R that is kept secure). One can later determine if a file has been modified by recomputing H(F). An intruder would need to change F without changing H(F).

A cryptographic hash function can be used to construct a **pseudorandom function (PRF)** or a **pseudorandom number generator (PRNG)**. A common application for a hash-based PRF is for the generation of symmetric keys.

Security requirements



The first three properties are required for any practical hash function
The last three extend security and can exist together or independently

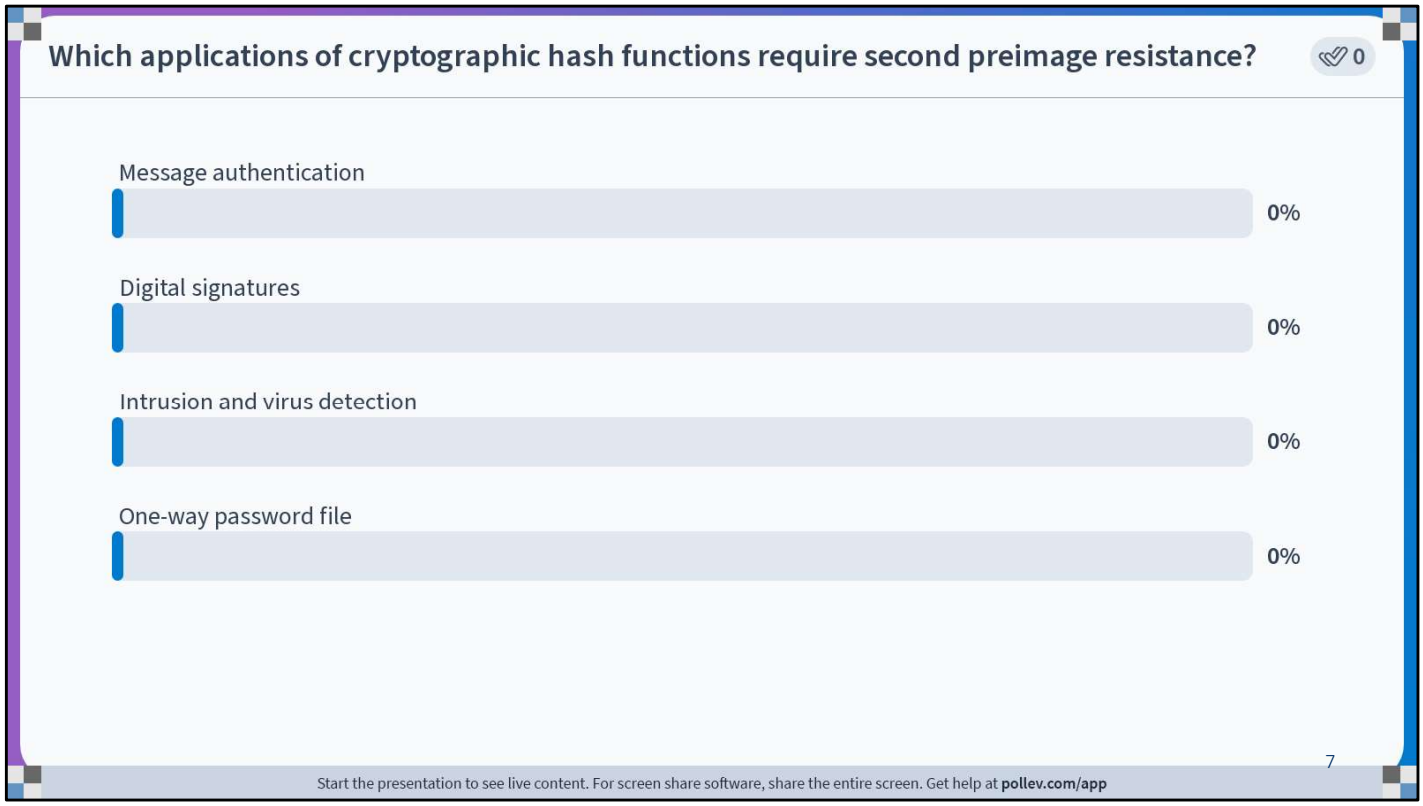
University of Antwerp
Faculty of Science

6

The fourth property, **preimage resistant**, is the one-way property: it is easy to generate a code given a message, but virtually impossible to generate a message given a code. This property is important if the authentication technique involves the use of a secret value. The secret value itself is not sent. However, if the hash function is not one way, an attacker can easily discover the secret value: If the attacker can observe or intercept a transmission, the attacker obtains the message M, and the hash code $h = H(S||M)$. The attacker then inverts the hash function to obtain $S||M = H^{-1}(MD_M)$. Because the attacker now has both M and S_{AB} II M, it is a trivial matter to recover S_{AB}.

The fifth property, **second preimage resistant**, guarantees that it is infeasible to find an alternative message with the same hash value as a given message. This prevents forgery when an encrypted hash code is used. If this property were not true, an attacker would be capable of the following sequence: First, observe or intercept a message plus its encrypted hash code; second, generate an unencrypted hash code from the message; third, generate an alternate message with the same hash code.

A hash function that satisfies the first five properties is referred to as a weak hash function. If the sixth property, **collision resistant**, is also satisfied, then it is referred to as a strong hash function. A strong hash function protects against an attack in which one party generates a message for another party to sign. For example, suppose Bob writes an IOU (I Owe You) message, sends it to Alice, and she signs it. Bob finds two messages with the same hash, one of which requires Alice to pay a small amount and one that requires a large payment. Alice signs the first message, and Bob is then able to claim that the second message is authentic.



Poll Title: Do not modify the notes in this section to avoid tampering with the Poll Everywhere activity.
More info at polleverywhere.com/support

Which applications of cryptographic hash functions require second preimage resistance?
https://www.polleverywhere.com/multiple_choice_polls/ZPZaOzmmLQ40LQo?state=opened&flow=Default&onscreen=persist

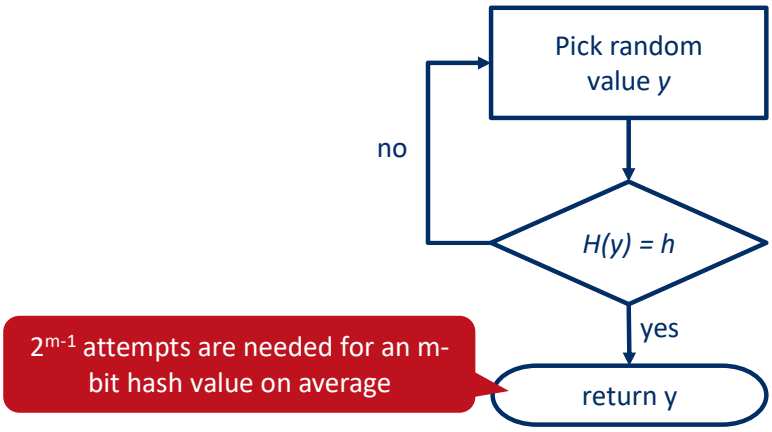
Security requirements of applications

	Preimage resistant	Second pre-image resistant	collision resistant
message authentication	Yes	Yes	Yes*
digital signatures	Yes	Yes	Yes*
intrusion and virus detection		Yes	
one-way password file	Yes		

*Only required against a chosen plaintext message attack

Brute force preimage attacks

- Circumvent the (second) preimage resistance properties
- **Goal:** For a given h, find y such that H(y) = h (or for a given x, find y such that H(y) = H(x))



University of Antwerp | Faculty of Science

9

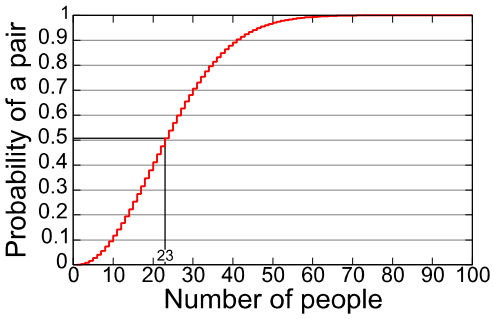
As with encryption algorithms, there are two categories of attacks on hash functions: brute-force attacks and cryptanalysis. A brute-force attack does not depend on the specific algorithm but depends only on bit length. In the case of a hash function, a brute-force attack depends only on the bit length of the hash value. A cryptanalysis, in contrast, is an attack based on weaknesses in a particular cryptographic algorithm. We look at brute-force attacks.

For a preimage or second preimage attack, an adversary wishes to find a value *y* such that H(*y*) is equal to a given hash value *h*. The brute-force method is to pick values of *y* at random and try each value until a collision occurs. For an *m*-bit hash value, the level of effort is proportional to 2^m. Specifically, the adversary would have to try, on average, 2^{m-1} values of y to find one that generates a given hash value *h*.

Brute force collision resistance attacks

- Circumvent the collision resistance properties
- **Goal:** Find x and y such that H(x) = H(y)
- Using the birthday paradox, it can be shown that on average only $\sqrt{2^m} = 2^{m/2}$ attempts are needed (for an m-bit hash value)

- **Birthday paradox**
  - Among 23 people, there is a 50% chance that two share a birthday
  - **Generalization:** Choosing random variables from a uniform distribution in the range of 0 to N – 1, the probability of a repeated element exceeds 0.5 after ~ $\sqrt{N}$ choices have been made

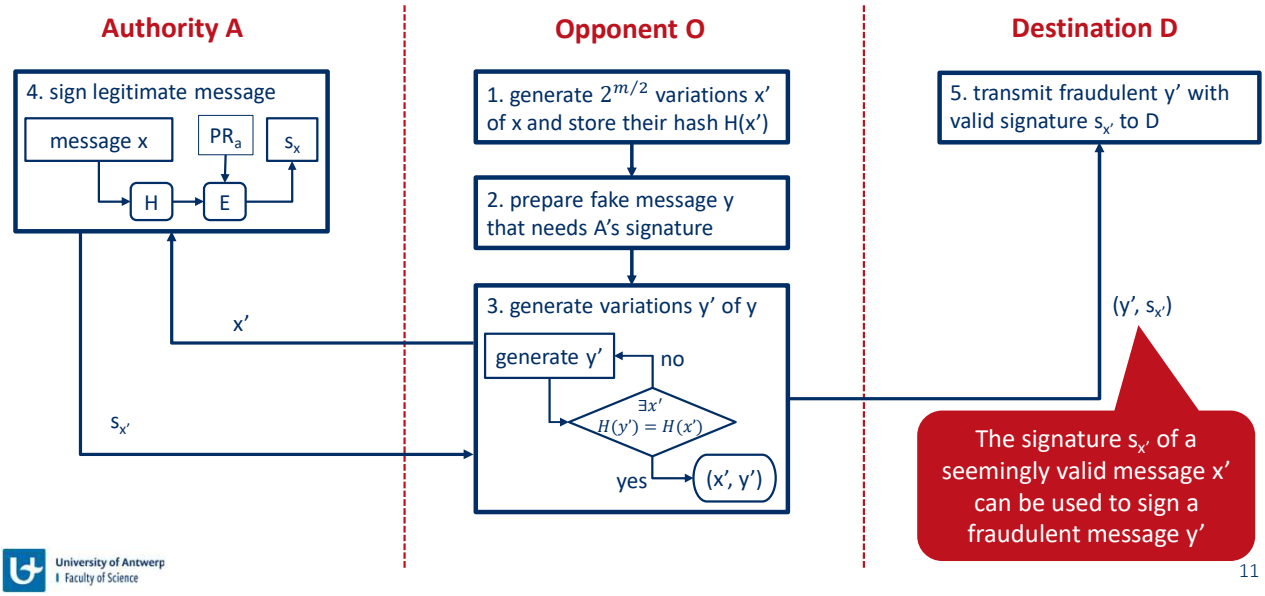


University of Antwerp
Faculty of Science

10

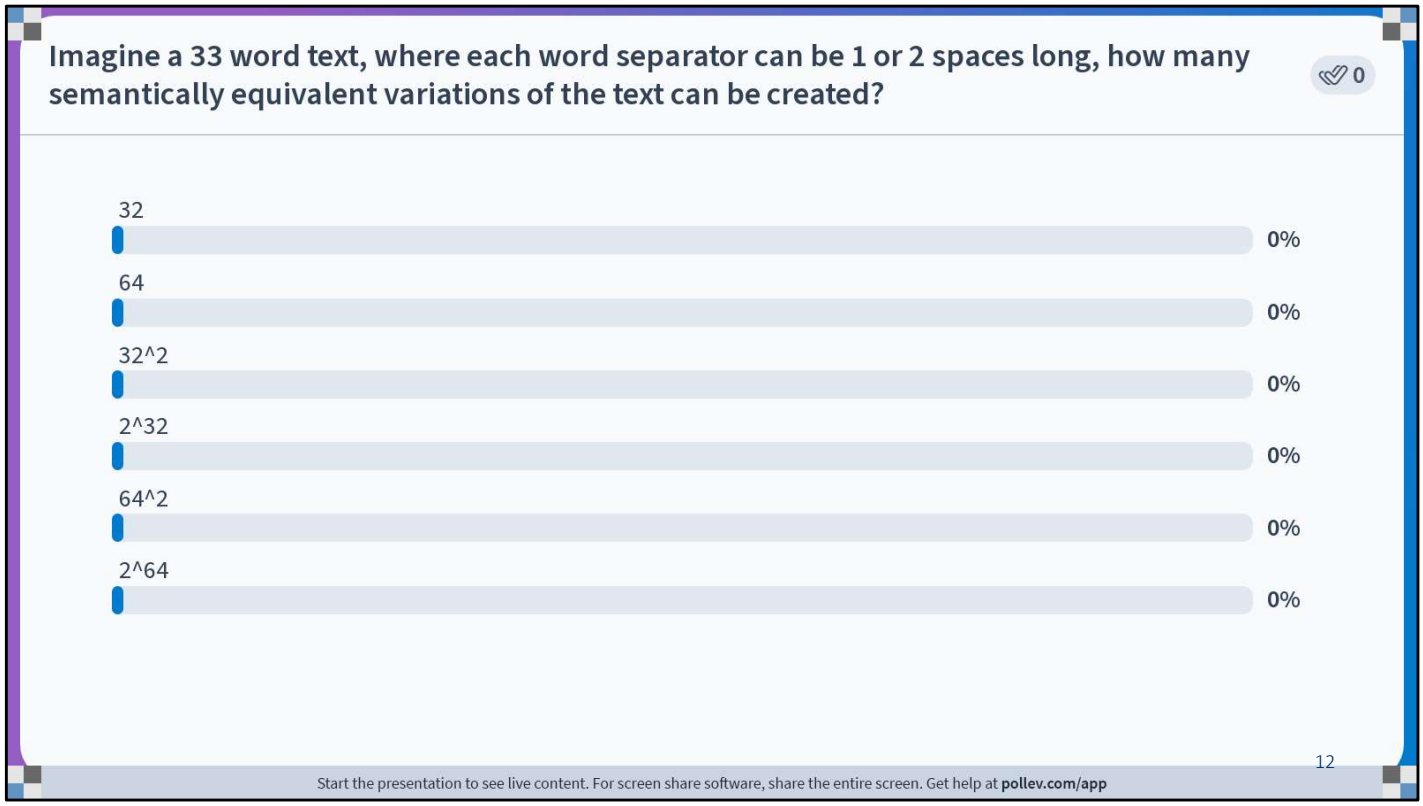
For a collision resistant attack, an adversary wishes to find two messages or data blocks, *x* and *y*, that yield the same hash function: H(*x*) = H(*y*). This turns out to require considerably less effort than a preimage or second preimage attack. The effort required is explained by a mathematical result referred to as the **birthday paradox**. In essence, if we choose random variables from a uniform distribution in the range 0 through *N* - 1, then the probability that a repeated element is encountered exceeds 0.5 after sqrt(*N)* choices have been made. Thus, for an *m*-bit hash value, if we pick data blocks at random, we can expect to find two data blocks with the same hash value within sqrt(2^m) = $2^{m/2}$ attempts.

Collision resistance attack using birthday paradox



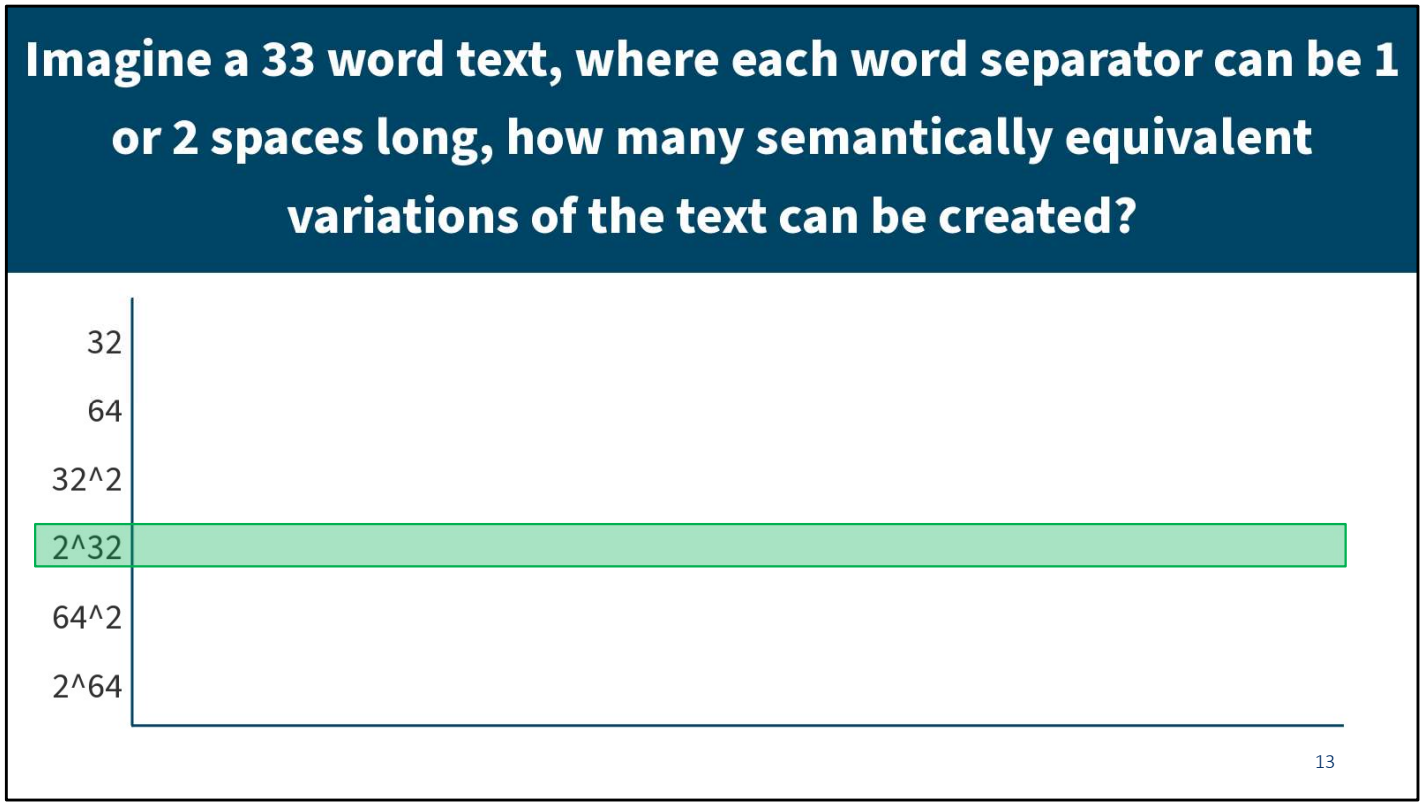
Yuval proposed the following strategy to exploit the birthday paradox in a collision resistant attack:

1. The source, A, is prepared to sign a legitimate message *x* by appending the appropriate *m*-bit hash code and encrypting that hash code with A's private key.
2. The opponent generates $2^{m/2}$ variations *x*′ of *x*, all of which convey essentially the same meaning, and stores the messages and their hash values.
3. The opponent prepares a fraudulent message *y* for which A's signature is desired.
4. The opponent generates minor variations *y*′ of *y*, all of which convey essentially the same meaning. For each *y*′, the opponent computes *H*(*y*′), checks for matches with any of the *H*(*x*′) values, and continues until a match is found. That is, the process continues until a *y*′ is generated with a hash value equal to the hash value of one of the *x*′ values.
5. The opponent offers the valid variation to A for signature. This signature can then be attached to the fraudulent variation for transmission to the intended recipient. Because the two variations have the same hash code, they will pro- duce the same signature; the opponent is assured of success even though the encryption key is not known.



Poll Title: Do not modify the notes in this section to avoid tampering with the Poll Everywhere activity.
More info at polleverywhere.com/support

Imagine a 33 word text, where each word separator can be 1 or 2 spaces long, how many semantically equivalent variations of the text can be created?
https://www.polleverywhere.com/multiple_choice_polls/9gZsTOXNp013XXH?state=opened&flow=Default&onscreen=persist



Poll Title: Imagine a 33 word text, where each word separator can be 1 or 2 spaces long, how many semantically equivalent variations of the text can be created?

For a 64-bit hash only 2^{32} variations are needed

As {the / —} Dean of Blakewell College, I have {had the pleasure of knowing / known} Cherise Rosetti for the {last / past} four years. She {has been / was} {a tremendous / an outstanding} {asset to / role model in} {our / the} school. I {would like to take this opportunity to / wholeheartedly} recommend Cherise for your {school's / —} graduate program. I {am / feel} {confident / certain} {that / —} {she / Cherise} will {continue to / —} succeed in her studies. {She / Cherise} is a dedicated student and {thus far her grades / her grades thus far} {have been / are} {exemplary / excellent}. In class, {she / Cherise} {has proven to be / has been} a take-charge {person / individual} {who is / —} able to successfully develop plans and implement them.

{She / Cherise} has also assisted {us / —} in our admissions office. {She / Cherise} has {successfully / —} demonstrated leadership ability by counseling new and prospective students. {Her / Cherise's} advice has been {a great / of considerable} help to these students, many of whom have {taken time to share / shared} their comments with me regarding her pleasant and {encouraging / reassuring} attitude. {For these reasons / It is for these reasons that} I {highly recommend / offer high recommendations for} Cherise {without reservation / unreservedly}. Her {ambition / drive} and {abilities / potential} will {truly / surely} be an {asset to / plus for} your {establishment / school}.

These are 2^{38} variations of the same message already

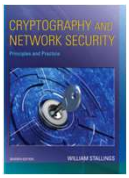
University of Antwerp | Faculty of Science

14

The generation of many variations that convey the same meaning is not difficult. For example, the opponent could insert a number of "space-space-backspace" character pairs between words throughout the document. Variations could then be generated by substituting "space-backspace-space" in selected instances. Alternatively, the opponent could simply reword the message but retain the meaning.

Summary: Hash Function Basics

- **Cryptographic hash functions** transform variable length input into fixed length hash code
- Many **applications** in computer security
  - Message authentication
  - Digital signatures
  - One-way password files
  - Intrusion & virus detection
  - Pseudorandom private key generation
- Efficient chosen plaintext attacks using **birthday paradox** are possible



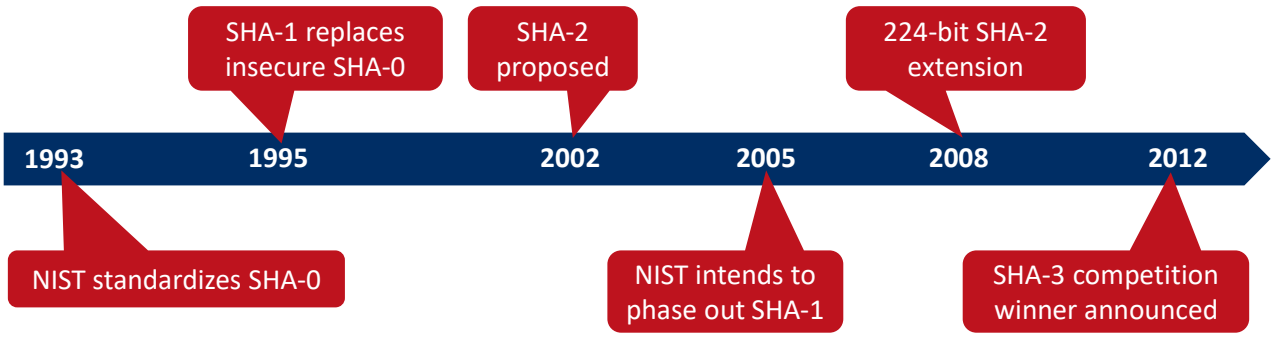
Link with the book
- Chapter 11 (Sections 11.1, 11.3)

Link with the videos
- Week 3 (Collision Resistance 1)

Secure Hash Function (SHA)

University of Antwerp
Faculty of Science

Secure Hash Algorithm (SHA) history



University of Antwerp
Faculty of Science

17

In recent years, the most widely used hash function has been the Secure Hash Algorithm (SHA). Indeed, because virtually every other widely used hash function had been found to have substantial cryptanalytic weaknesses, SHA was more or less the last remaining standardized hash algorithm by 2005. SHA was developed by the National Institute of Standards and Technology (NIST) and published as a federal information processing standard (FIPS 180) in 1993. When weaknesses were discovered in SHA, now known as **SHA-0**, a revised version was issued as FIPS 180-1 in 1995 and is referred to as **SHA-1**. The actual standards document is entitled "Secure Hash Standard." SHA is based on the hash function MD4, and its design closely models MD4.

1. SHA-1 and SHA-2
2. Length Extension Attack
3. SHA-3

University of Antwerp
Faculty of Science

The original SHA hash function family (sizes in bits)

- SHA-1 and SHA-2 use the same structure as SHA-0 and MD4
- SHA-224 is generated using SHA-256, truncated to the 224 leftmost bits
- SHA-384, SHA-512/224 and SHA-512/256 use t leftmost bits of SHA-512
- SHA-1 and SHA-2 are **susceptible to length extension attacks**

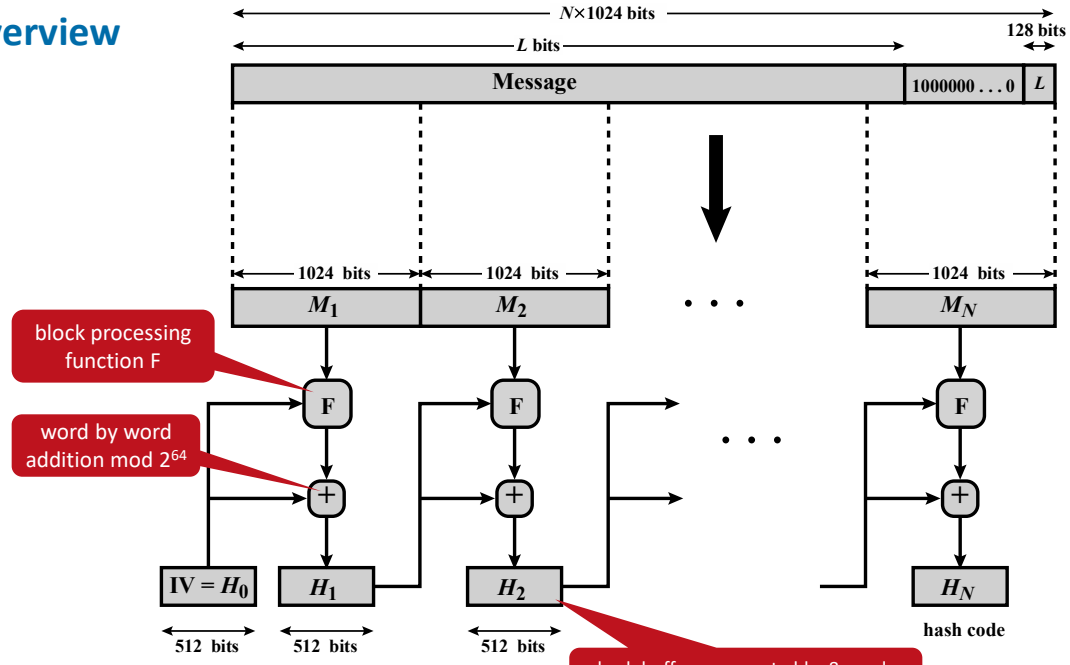
	SHA-1	SHA-2					
		SHA-224	SHA-256	SHA-384	SHA-512	SHA-512/224	SHA-512/256
Hash size	160	224	256	384	512	224	256
Message size	$< 2^{64}$	$< 2^{64}$	$< 2^{64}$	$< 2^{128}$	$< 2^{128}$	$< 2^{128}$	$< 2^{128}$
Block size	512	512	512	1024	1024	1024	1024
Word size	32	32	32	64	64	64	64
Rounds	80	64	64	80	80	80	80

University of Antwerp | Faculty of Science

19

SHA-1 produces a hash value of 160 bits. In 2002, NIST produced a revised version of the standard, FIPS 180-2, that defined three new versions of SHA, with hash value lengths of 256, 384, and 512 bits, known as SHA-256, SHA-384, and SHA-512, respectively. Collectively, these hash algorithms are known as **SHA-2**. These new versions have the same underlying structure and use the same types of modular arithmetic and logical binary operations as SHA-1. A revised document was issued as FIP PUB 180-3 in 2008, which added a 224-bit version (Table 11.3). In 2015, NIST issued FIPS 180-4, which added two additional algorithms: SHA-512/224 and SHA-512/256. SHA-1 and SHA-2 are also specified in RFC 6234, which essentially duplicates the material in FIPS 180-3 but adds a C code implementation. In 2005, NIST announced the intention to phase out approval of SHA-1 and move to a reliance on SHA-2 by 2010. Shortly thereafter, a research team described an attack in which two separate messages could be found that deliver the same SHA-1 hash using 2^{69} operations, far fewer than the 2^{80} operations previously thought needed to find a collision with an SHA-1 hash.

SHA-512 overview



University of Antwerp | Faculty of Science

20

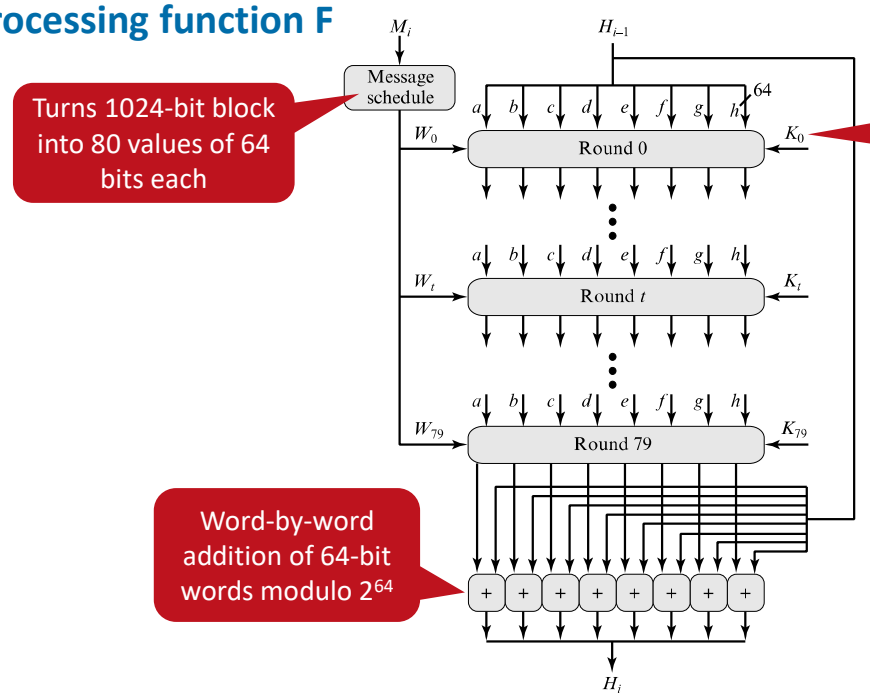
The algorithm takes as input a message with a maximum length of less than 2^{128} bits and produces as output a 512-bit message digest. The input is processed in 1024-bit blocks. The processing consists of the following steps:

1. **Append padding bits.** The message is padded so that its length is congruent to 896 modulo 1024 [length = 896 (mod 1024)]. Padding is always added, even if the message is already of the desired length. Thus, the number of padding bits is in the range of 1 to 1024. The padding consists of a single 1 bit followed by the necessary number of 0 bits.
2. **Append length.** A block of 128 bits is appended to the message. This block is treated as an unsigned 128-bit integer (most significant byte first) and contains the length of the original message in bits (before the padding). The outcome of the first two steps yields a message that is an integer multiple of 1024 bits in length. The expanded message is represented as the sequence of 1024-bit blocks M_1, M_2,… , M_N, so that the total length of the expanded message is N * 1024 bits.
3. **Initialize hash buffer.** A 512-bit buffer is used to hold intermediate and final results of the hash function. The buffer can be represented as eight 64-bit registers (a, b, c, d, e, f, g, h). These registers are initialized to the following 64-bit integers (hexadecimal values):
    - a = 6A09E667F3BCC908
    - b = BB67AE8584CAA73B
    - c = 3C6EF372FE94F82B
    - d = A54FF53A5F1D36F1
    - e = 510E527FADE682D1
    - f = 9B05688C2B3E6C1F
    - g = 1F83D9ABFB41BD6B
    - h = 5BE0CD19137E2179

- These values are stored in **big-endian** format, which is the most significant byte of a word in the low-address (leftmost) byte position. These words were obtained by taking the first sixty-four bits of the fractional parts of the square roots of the first eight prime numbers.

1. **Process message in 1024-bit (128-byte) blocks.** The heart of the algorithm is a module that consists of 80 rounds; this module is labeled F.
2. **Output.** After all *N* 1024-bit blocks have been processed, the output from the *N*th stage is the 512-bit message digest.

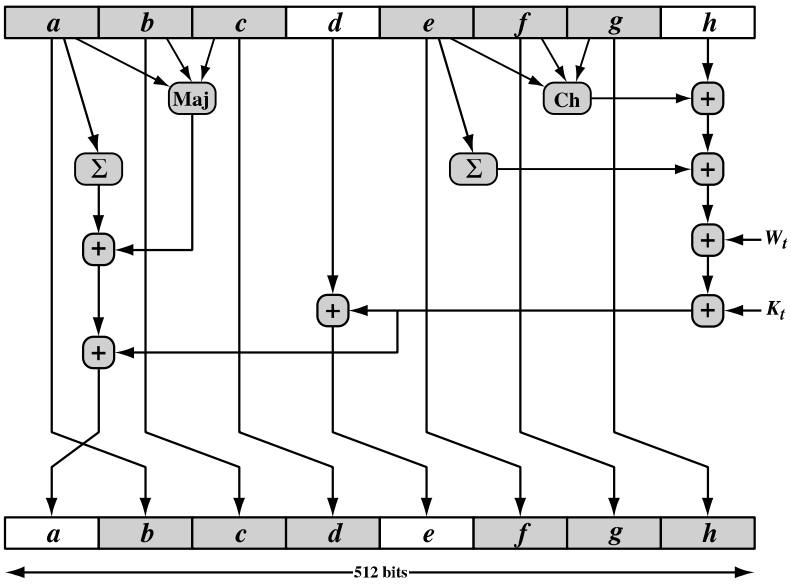
SHA-512 block processing function F



Each round takes as input the 512-bit buffer value, abcdefgh, and updates the contents of the buffer. As input to the first round, the buffer has the value of the intermediate hash value, H_{i-1}. Each round t makes use of a 64-bit value W_t, derived from the current 1024-bit block being processed (M_i). These values are derived using a message schedule. Each round also makes use of an additive constant K_t, where $0 \leq t \leq 79$ indicates one of the 80 rounds. These words represent the first 64 bits of the fractional parts of the cube roots of the first 80 prime numbers. The constants provide a "randomized" set of 64-bit patterns, which should eliminate any regularities in the input data.

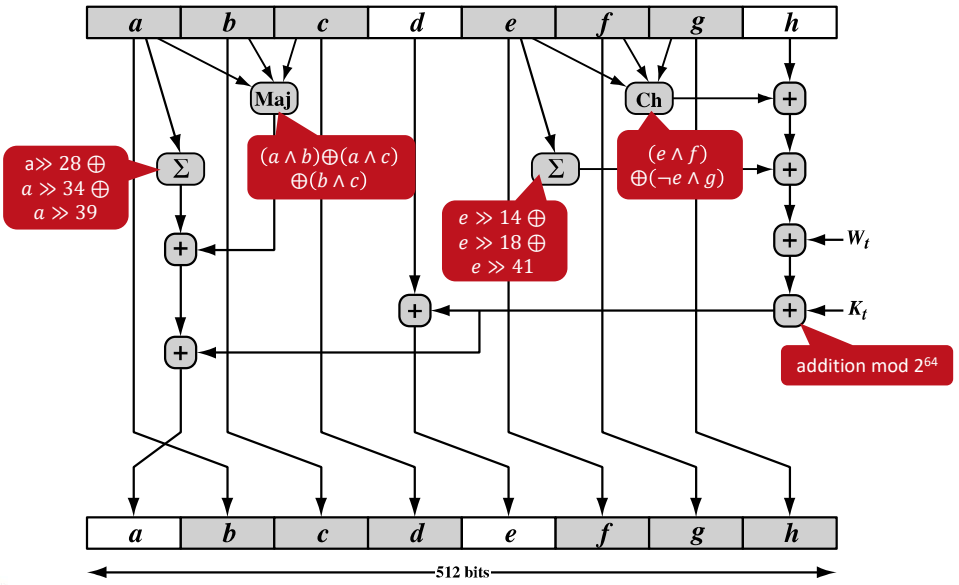
The output of the eightieth round is added to the input to the first round (H_{i-1}) to produce H_i. The addition is done independently for each of the eight words in the buffer with each of the corresponding words in H_{i-1}, using addition modulo 2^{64}.

Round function: substitution and permutation



University of Antwerp
Faculty of Science

Round function: substitution and permutation



Let us look in more detail at the logic in each of the 80 steps of the processing of one 512-bit block. Each round is defined by the following set of equations:

$T_1 = h + Ch(e,f,g) + \left(\sum_1^{512} e\right) + W_t + K_t$

$T_2 = \left(\sum_0^{512} a\right) + Maj(a,b,c)$

$h = g$

$g = f$

$f = e$

$e = d + T_1$

$d = c$

$c = b$

$b = a$

$a = T_1 + T_2$

where

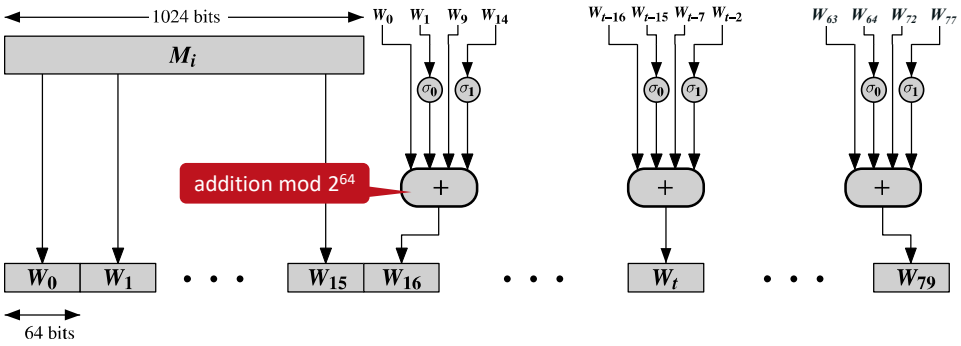
t	= step number; $0 \le t \le 79$
Ch(e, f, g)	= (e AND f) ⊕ (NOT e AND g) *the conditional function: If e then f else g*
Maj(a, b, c)	= (a AND b) ⊕(a AND c) ⊕(b AND c) *the function is true only of the majority (two or three) of the arguments are true*
$\left(\sum_0^{512} a\right)$	= ROTR^{28}(a) ⊕ ROTR^{34}(a) ⊕ ROTR^{39}(a)
$\left(\sum_1^{512} e\right)$	= ROTR^{14}(e) ⊕ROTR^{18}(e) ⊕ROTR^{41}(e)
ROTR^n(x)	= circular right shift (rotation) of the 64-bit argument x by n bits
W_t	= a 64-bit word derived from the current 1024-bit input block

K_t = a 64-bit additive constant
\+ = addition modulo 2^{64}

Two observations can be made about the round function:

1. Six of the eight words of the output of the round function involve simply permutation (*b, c, d, f, g, h*) by means of rotation. This is indicated by shading.
2. Only two of the output words (*a, e*) are generated by substitution. Word *e* is a function of input variables (*d, e, f, g, h*), as well as the round word W_t and the constant K_t. Word *a* is a function of all of the input variables except *d*, as well as the round word W_t and the constant K_t.

Transforming a block M into 64-bit words W_0-W_{79}



$\sigma_0(x) = \text{ROTR}^1(x) \oplus \text{ROTR}^8(x) \oplus \text{SHR}^7(x)$
$\sigma_1(x) = \text{ROTR}^{19}(x) \oplus \text{ROTR}^{61}(x) \oplus \text{SHR}^6(x)$

$\text{ROTR}^n(x)$ = circular right bit-shift of the 64-bit argument x by n bits
$\text{SHR}^n(x)$ = right bit-shift of the 64-bit argument x by n bits with padding by n zeros on the left
$\oplus$ = bitwise XOR

University of Antwerp
Faculty of Science

24

The first 16 values of W_t are taken directly from the 16 words of the current block. The remaining values are defined as

$$W_t = \sigma_1^{512}(W_{t-2}) + W_{t-7} + \sigma_0^{512}(W_{t-15}) + W_{t-16}$$

where

$$\sigma_0^{512}(x) = ROTR^1(x) \oplus ROTR^8(x) \oplus SHR^7(x)$$
$$\sigma_1^{512}(x) = ROTR^{19}(x) \oplus ROTR^{61}(x) \oplus SHR^6(x)$$

$\text{ROTR}^n(x)$ = circular right shift (rotation) of the 64-bit argument x by n bits
$\text{SHR}^n(x)$ = right shift of the 64-bit argument x by n bits with padding by zeros on the left
$+$ = addition modulo 2^{64}

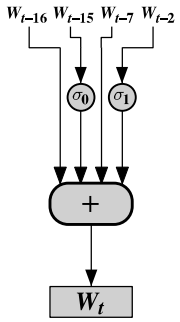
Thus, in the first 16 steps of processing, the value of W_t is equal to the corresponding word in the message block. For the remaining 64 steps, the value of W_t consists of the circular left shift by one bit of the XOR of four of the preceding values of W_t, with two of those values subjected to shift and rotate operations. This introduces a great deal of redundancy and interdependence into the message blocks that are compressed, which complicates the task of finding a different message block that maps to the same compression function output.

The SHA-512 algorithm has the property that every bit of the hash code is a function of every bit of the input. The complex repetition of the basic function F produces results that are well mixed; that is, it is unlikely that two messages chosen at random, even if they exhibit similar regularities, will have the same hash code.

Exercise: Calculating SHA round words

Exercise: Given a simplified version of SHA with a 128-bit block size and 8-bit word size, calculate round word W_{16}.

M = {5C 84 C2 03 06 9A AF A7 5A CA 8D 4B C9 2A 06 30}



$$\sigma_0(x) = \mathrm{ROTR}^1(x) \oplus \mathrm{ROTR}^2(x) \oplus \mathrm{SHR}^7(x)$$
$$\sigma_1(x) = \mathrm{ROTR}^3(x) \oplus \mathrm{ROTR}^5(x) \oplus \mathrm{SHR}^6(x)$$

$\mathrm{ROTR}^n(x)$ = circular right bit-shift of the 64-bit argument x by n bits
$\mathrm{SHR}^n(x)$ = right bit-shift of the 64-bit argument x by n bits with padding by n zeros on the left

Solution: Calculating SHA round words (1/2)

$$W_{16} = W_0 + \sigma_0(W_1) + W_9 + \sigma_1(W_{14}) \; mod \; 2^8$$
$$W_{16} = 5C + \left(\text{ROTR}^1(84) \oplus \text{ROTR}^2(84) \oplus \text{SHR}^7(84)\right) + CA + \left(\text{ROTR}^3(06) \oplus \text{ROTR}^5(06) \oplus \text{SHR}^6(06)\right) \; mod \; 2^8$$

84	= 1000 0100	06	= 0000 0110
$\text{ROTR}^1(84)$	= 0100 0010	$\text{ROTR}^3(06)$	= 1100 0000
$\text{ROTR}^2(84)$	= 0010 0001	$\text{ROTR}^5(06)$	= 0011 0000
$\text{SHR}^7(84)$	= 0000 0001	$\text{SHR}^6(06)$	= 0000 0000
$\oplus$	= 0110 0010	$\oplus$	= 1111 0000
	= 62		= F0

Solution: Calculating SHA round words (2/2)

$W_{16} = W_0 + \sigma_0(W_1) + W_9 + \sigma_1(W_{14})\ mod\ 2^8$

W_0	0101 1100	=	{5C}	=	92
$\sigma_0(W_1)$	0110 0010	=	{62}	=	98
W_9	1100 1010	=	{CA}	=	202
$\sigma_1(W_{14})$	1111 0000	=	{F0}	=	240
+				=	632
$mod\ 2^8$					120

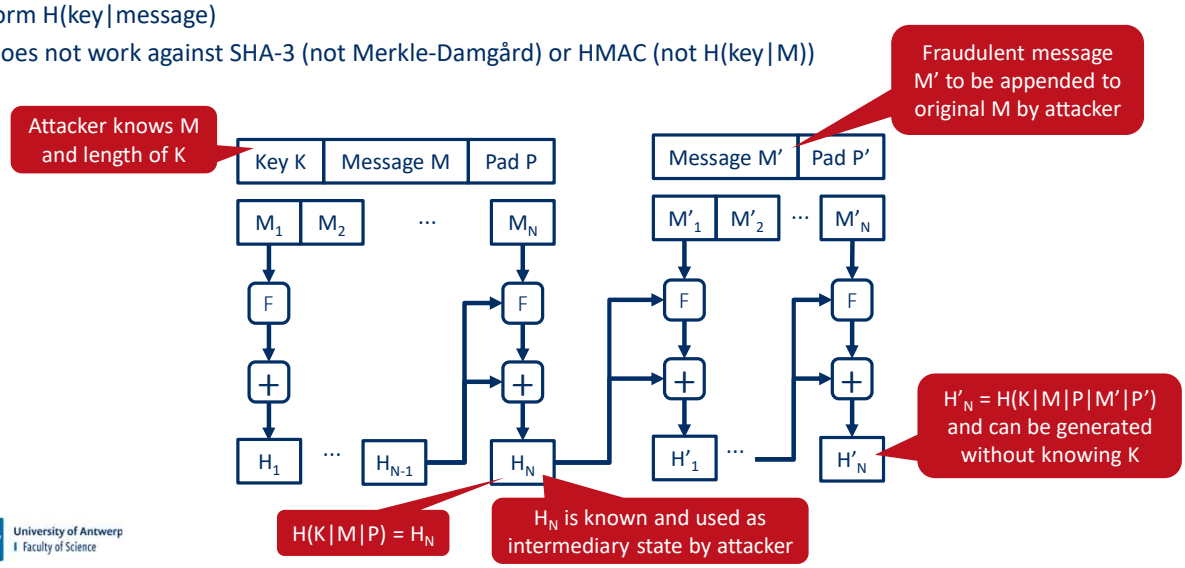
$W_{16} = 120 = \{78\}$

1. SHA-1 and SHA-2
2. Length Extension Attack
3. SHA-3

University of Antwerp
Faculty of Science

Length extension attack

- Works against any Merkle-Damgård hash (e.g., SHA-1, SHA-2), used as a message authentication code (MAC) of the form H(key|message)
- Does not work against SHA-3 (not Merkle-Damgård) or HMAC (not H(key|M))



University of Antwerp | Faculty of Science

29

A length extension attack is a type of attack where an attacker can use Hash(message1) and the length of message1 to calculate Hash(message1 ‖ message2) for an attacker-controlled message2. Algorithms like MD5, SHA-1, and SHA-2 that are based on the Merkle–Damgård construction are susceptible to this kind of attack. The SHA-3 algorithm is not susceptible.

When a Merkle–Damgård based hash is misused as a message authentication code with construction H(secret ‖ message), and message and the length of secret is known, a length extension attack allows anyone to include extra information at the end of the message and produce a valid hash without knowing the secret. Note that since HMAC doesn't use this construction, HMAC hashes are not prone to length extension attacks.

The vulnerable hashing functions work by taking the input message, and using it to transform an internal state. After all of the input has been processed, the hash digest is generated by outputting the internal state of the function. It is possible to reconstruct the internal state from the hash digest, which can then be used to process the new data. In this way one may extend the message and compute the hash that is a valid signature for the new message.

Length extension attack: HTTP example

- Consider a website for ordering waffles online
- The user orders a waffle of type eggo online using the following request URL:
  - Original Data: count=10&lat=37.351&user_id=1&long=-119.827&waffle=eggo
  - Original Signature: 6d5f807e23db210bc254a28be2d6759a0f5f5d99
- An attacker can overwrite choice of waffle by overwriting the URL parameter:
  - Desired New Data: count=10&lat=37.351&user_id=1&long=-119.827&waffle=eggo**&waffle=liege**
- If the attacker knows the length of key K, and the algorithm used, they can generate the **correct padding** from the original request and append **new data**:
  - New Data: count=10&lat=37.351&user_id=1&long=-119.827&waffle=eggo
    \x80\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00
    \x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00
    \x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00
    \x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x02\x**&waffle=liege**
  - New Signature: 0e41270260895979317fff3898ab85668953aaa2

University of Antwerp | Faculty of Science

Source: *https://en.wikipedia.org/wiki/Length_extension_attack*

30

A server for delivering waffles of a specified type to a specific user at a location could be implemented to handle requests of the given format:

> *Original Data: count=10&lat=37.351&user_id=1&long=-119.827&waffle=eggo*
> *Original Signature: 6d5f807e23db210bc254a28be2d6759a0f5f5d99*

The server would perform the request given (to deliver ten waffles of type eggo to the given location for user 1) only if the signature is valid for the user. The signature used here is a MAC, signed with a key not known to the attacker.

It is possible for an attacker to modify the request, in this example switching the requested waffle from "eggo" to "liege." This can be done by taking advantage of a flexibility in the message format if duplicate content in the query string gives preference to the latter value. This flexibility does not indicate an exploit in the message format, because the message format was never designed to be cryptographically secure in the first place, without the signature algorithm to help it.

Desired New Data: count=10&lat=37.351&user_id=1&long=-119.827&waffle=eggo&waffle=liege

In order to sign this new message, typically the attacker would need to know the key the message was signed with, and generate a new signature by generating a new MAC. However, with a length extension attack, it is possible to feed the hash (the signature given above) into the state of the hashing function, and continue where the original request had left off, so

long as you know the length of the original request. In this request, the original key's length was 14 bytes, which could be determined by trying forged requests with various assumed lengths, and checking which length results in a request that the server accepts as valid.

The message as fed into the hashing function is often padded, as many algorithms can only work on input messages whose lengths are a multiple of some given size. The content of this padding is always specified by the hash function used. The attacker must include all of these padding bits in their forged message before the internal states of their message and the original will line up. Thus, the attacker constructs a slightly different message using these padding rules:

New Data: count=10&lat=37.351&user_id=1&long=-119.827&waffle=eggo\x80\x00\x00
\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00
\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00
\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00
\x00\x00\x02\x28&waffle=liege

This message includes all of the padding that was appended to the original message inside of the hash function before their payload (in this case, a 0x80 followed by a number of 0x00s and a message length, 0x228 = 552 = (14+55)*8, which is the length of the key plus the original message, appended at the end). The attacker knows that the state behind the hashed key/message pair for the original message is identical to that of new message up to the final "&." The attacker also knows the hash digest at this point, which means they know the internal state of the hashing function at that point. It is then trivial to initialize a hashing algorithm at that point, input the last few characters, and generate a new digest which can sign their new message without the original key.

New Signature: 0e41270260895979317fff3898ab85668953aaa2

By combining the new signature and new data into a new request, the server will see the forged request as a valid request due to the signature being the same as it would have been generated if the password was known.

1. SHA-1 and SHA-2
2. Length Extension Attack
3. SHA-3

University of Antwerp
Faculty of Science

SHA-3 parameters

	SHA-3			
Hash code size (bits)	224	256	384	512
Message size	--	--	--	--
Block size r (bits)	1152	1088	832	576
Word size (bits)	64	64	64	64
Number of rounds	24	24	24	24
Capacity c	448	512	768	1024
Collision resistance	2^{112}	2^{128}	2^{192}	2^{256}
Second preimage resistance	2^{224}	2^{256}	2^{384}	2^{512}

$r + c = 1600$ bits

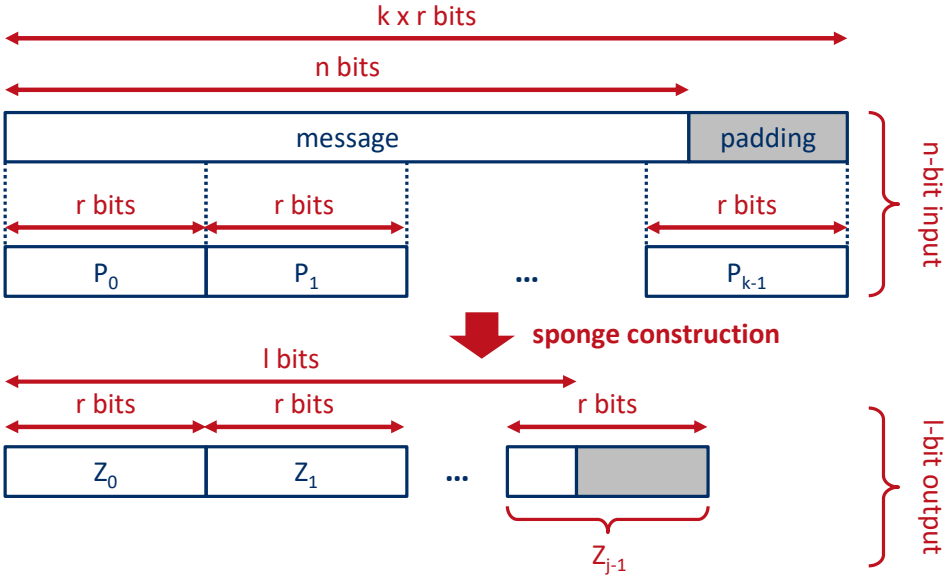
University of Antwerp | Faculty of Science

32

Accordingly, NIST announced in 2007 a competition to produce the next generation NIST hash function, to be called SHA-3. The winning design for SHA-3 was announced by NIST in October 2012 and published as FIP 102 in August 2015. SHA-3 is a cryptographic hash function that is intended to complement SHA-2 as the approved standard for a wide range of applications.

SHA-3 makes use of the iteration function *f*, labeled Keccak-*f*. The overall SHA-3 function is a sponge function expressed as Keccak[*r*, *c*] to reflect that SHA-3 has two operational parameters, *r*, the message block size, and *c*, the capacity, with the default of $r + c = 1600$ bits. The table shows the supported values of *r* and *c*. The hash function security associated with the sponge construction is a function of the capacity *c*.

SHA-3 uses variable input/output sponges



University of Antwerp
Faculty of Science

33

The underlying structure of SHA-3 is a scheme referred to by its designers as a **sponge construction**. The sponge construction has the same general structure as other iterated hash functions. The sponge function takes an input message and partitions it into fixed-size blocks. Each block is processed in turn with the output of each iteration fed into the next iteration, finally producing an output block.
The sponge function is defined by three parameters:

f = the internal function used to process each input block
r = the size in bits of the input blocks, called the **bitrate**
padding = the padding algorithm

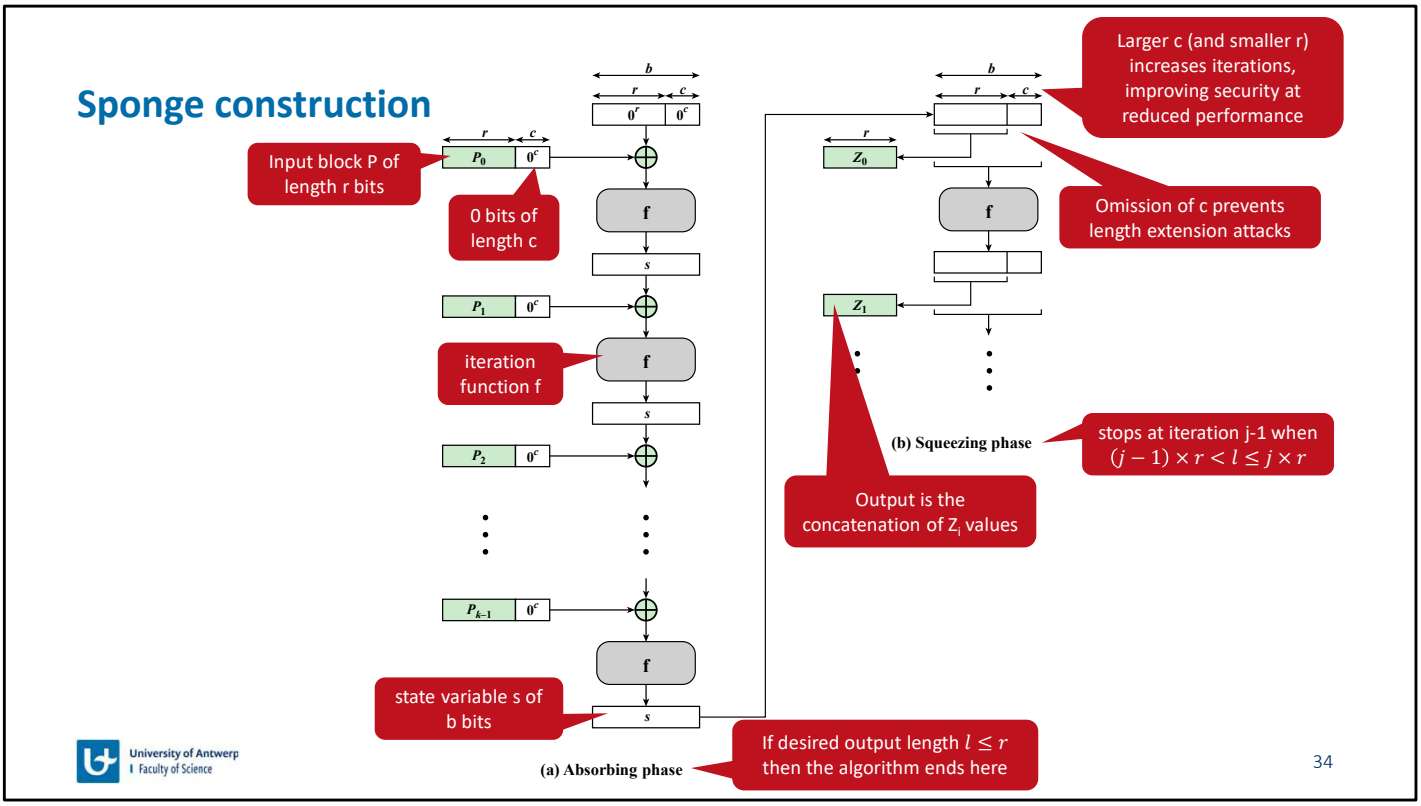
A sponge function allows both variable length input and output, making it a flexible structure that can be used for a hash function (fixed-length output), a pseudorandom number generator (fixed-length input), and other cryptographic functions.

An input message of *n* bits is partitioned into *k* fixed-size blocks of *r* bits each. The message is padded to achieve a length that is an integer multiple of *r* bits. The resulting partition is the sequence of blocks $P_0, P_1, \ldots, P_{k-1}$, with length $k * r$. For uniformity, padding is always added, so that if $n \bmod r = 0$, a padding block of *r* bits is added. The actual padding algorithm is a parameter of the function. The sponge specification proposes two padding schemes:

- **Simple padding:** Denoted by pad10*, appends a single bit 1 followed by the minimum number of bits 0 such that the length of the result is a multiple of the block length.
- **Multirate padding:** Denoted by pad10*1, appends a single bit 1 followed by the minimum number of bits 0 followed by a single bit 1 such that the length of the result is a multiple of the block length. This is the simplest padding scheme that allows secure use of the same *f*

with different rates *r*. FIPS 202 uses multirate padding.

After processing all of the blocks, the sponge function generates a sequence of output blocks $Z_0, Z_1, \ldots, Z_{j-1}$. The number of output blocks generated is determined by the number of output bits desired. If the desired output is / bits, then *j* blocks are produced, such that (*j* - 1) * *r* < l <= *j* * *r*.



This slide shows the iterated structure of the sponge function. The sponge construction operates on a state variable *s* of *b* = *r* + *c* bits, which is initialized to all zeros and modified at each iteration. The value *r* is called the bitrate. This value is the block size used to partition the input message. The term *bitrate* reflects the fact that *r* is the number of bits processed at each iteration: the larger the value of *r*, the greater the rate at which message bits are processed by the sponge construction. The value *c* is referred to as the **capacity**. A discussion of the security implications of the capacity is beyond our scope. In essence, the capacity is a measure of the achievable complexity of the sponge construction and therefore the achievable level of security. A given implementation can trade claimed security for speed by increasing the capacity *c* and decreasing the bitrate *r* accordingly, or vice versa. The default values for Keccak are *c* = 1024 bits, *r* = 576 bits, and therefore *b* = 1600 bits.

The sponge construction consists of two phases. The **absorbing phase** proceeds as follows: For each iteration, the input block to be processed is padded with zeroes to extend its length from *r* bits to *b* bits. Then, the bitwise XOR of the extended message block and *s* is formed to create a *b*-bit input to the iteration function *f*. The output of *f* is the value of *s* for the next iteration.

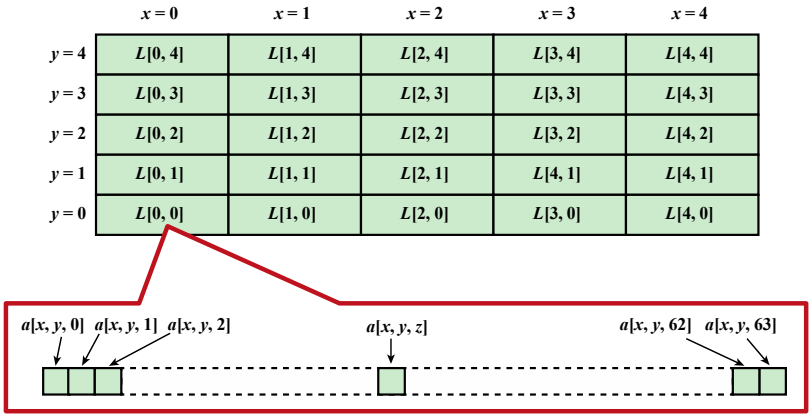
If the desired output length *l* satisfies *l* <= *r*, then at the completion of the absorbing phase, the first *l* bits of *s* are returned and the sponge construction terminates. Otherwise, the sponge construction enters the **squeezing phase**. To begin, the first *r* bits of *s* are retained as block Z_0. Then, the value of *s* is updated with repeated executions of *f*, and at each iteration, the first *l* bits of *s* are retained as block Z_i and concatenated with previously generated blocks. The process continues through (*j* - 1) iterations until we have (*j* - 1) * *r* ≤ *l* ≤ *j* * *r*. At this point

the first *l* bits of the concatenated block *Z* are returned.

Note that the absorbing phase has the structure of a typical hash function. A common case will be one in which the desired hash length is less than or equal to the input block length; that is, *l* <= *r*. In that case, the sponge construction terminates after the absorbing phase. If a longer output than *r* bits is required, then the squeezing phase is employed. Thus the sponge construction is quite flexible. For example, a short message with a length *r* could be used as a seed and the sponge construction would function as a pseudorandom number generator.

SHA-3 uses Keccak[r, c] as iteration function f

- By default, SHA-3 uses variables r and c, with $r + c = 1600$ bits
- Internal state is represented by 5x5 matrix of 64-bit "lanes" or "words"
  - a[x, y, z] refers to bit z of the lane at column x and row y
  - L[x, y] refers to the lane at column x and row y



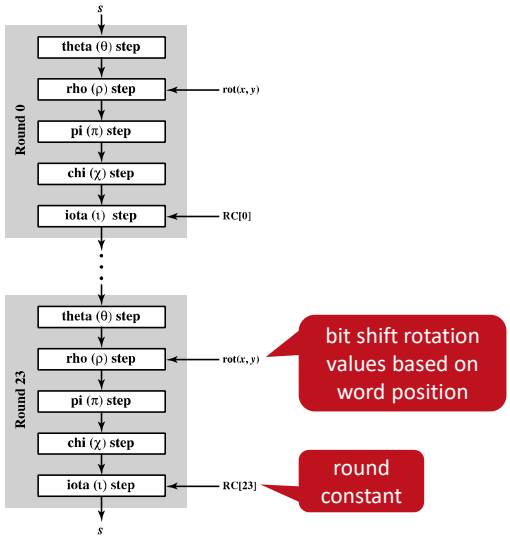
University of Antwerp | Faculty of Science

35

We now examine the iteration function Keccak-*f* used to process each successive block of the input message. Recall that *f* takes as input a 1600-bit variable *s* consisting of *r* bits, corresponding to the message block size followed by *c* bits, referred to as the capacity. For internal processing within *f*, the input state variable *s* is organized as a 5 * 5 * 64 array *a*. The 64-bit units are referred to as lanes. For our purposes, we generally use the notation *a*[*x*, *y*, *z*] to refer to an individual bit with the state array. When we are more concerned with operations that affect entire lanes, we designate the 5 * 5 matrix as *L*[*x*, *y*], where each entry in *L* is a 64-bit lane.

When treating the state as a matrix of lanes, the first lane in the lower left corner, *L*[0, 0], corresponds to the first 64 bits of *s*. The lane in the second column, lowest row, *L*[1, 0], corresponds to the next 64 bits of *s*. Thus, the array *a* is filled with the bits of *s* starting with row *y* = 0 and proceeding row by row.

Iteration function f has 24 rounds



Function	Type	Description
θ	Substitution	New value of each bit in each word depends on its current value and on one bit in each word of preceding column and one bit of each word in succeeding column.
ρ	Permutation	The bits of each word are permuted using a circular bit shift. $W[0, 0]$ is not affected.
π	Permutation	Words are permuted in the 5×5 matrix. $W[0, 0]$ is not affected.
χ	Substitution	New value of each bit in each word depends on its current value and on one bit in next word in the same row and one bit in the second next word in the same row.
ι	Substitution	$W[0, 0]$ is updated by XOR with a round constant.

rot(x, y):

	$x = 0$	$x = 1$	$x = 2$	$x = 3$	$x = 4$
$y = 4$	18	2	61	56	14
$y = 3$	41	45	15	21	8
$y = 2$	3	10	43	25	39
$y = 1$	36	44	6	55	20
$y = 0$	0	1	62	28	27

University of Antwerp | Faculty of Science

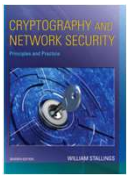
36

The function *f* is executed once for each input block of the message to be hashed. The function takes as input the 1600-bit state variable and converts it into a 5 * 5 matrix of 64-bit lanes. This matrix then passes through 24 rounds of processing. Each round consists of five steps, and each step updates the state matrix by permutation or substitution operations. As shown, the rounds are identical with the exception of the final step in each round, which is modified by a round constant that differs for each round.

The steps have a simple description leading to a specification that is compact and in which no trapdoor can be hidden. The operations on lanes in the specification are limited to bitwise Boolean operations (XOR, AND, NOT) and rotations. There is no need for table lookups, arithmetic operations, or data-dependent rotations. Thus, SHA-3 is easily and efficiently implemented in either hardware or software.

Summary: SHA

- **SHA-1 and SHA-2** are based on Merkle-Damgård principle, where each output block is used as the initial state for the next input block
- Merkle-Damgård hash functions are vulnerable to **length extension attacks** if used as a message authentication code of the form H(Key|Message)
- **SHA-3** is a more secure successor to SHA-2
  - It uses a capacity parameter to only reuse part of each output block as input for the next
  - The capacity reduces vulnerability to the length extension attack



Link with the book
- Chapter 11 (Sections 11.5, 11.6)

Link with the videos
- Week 3 (Collision Resistance 2)

Message Authentication

University of Antwerp
Faculty of Science

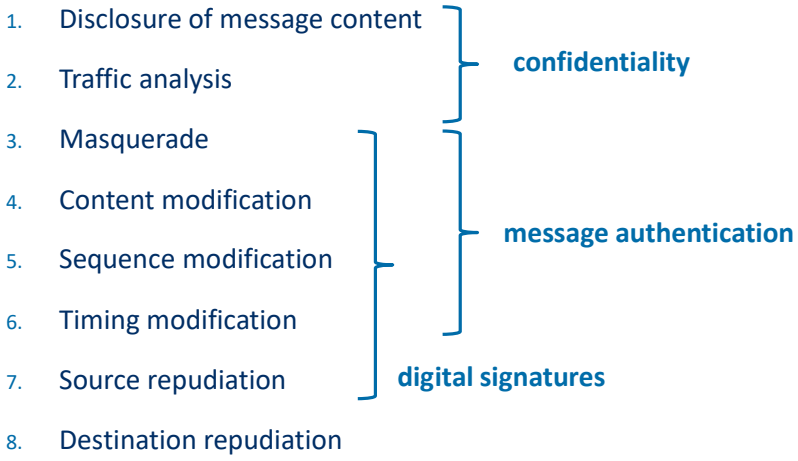
1 Message Authentication Basics

2 Message Authentication Codes

3 Authenticated Encryption

University of Antwerp
Faculty of Science

Types of attacks against transmitted messages



University of Antwerp | Faculty of Science

40

In the context of communications across a network, the following attacks can be identified:

1. **Disclosure:** Release of message contents to any person or process not possessing the appropriate cryptographic key.
2. **Traffic analysis:** Discovery of the pattern of traffic between parties. In a connection-oriented application, the frequency and duration of connections could be determined. In either a connection-oriented or connectionless environment, the number and length of messages between parties could be determined.
3. **Masquerade:** Insertion of messages into the network from a fraudulent source. This includes the creation of messages by an opponent that are purported to come from an authorized entity. Also included are fraudulent acknowledgments of message receipt or nonreceipt by someone other than the message recipient.
4. **Content modification:** Changes to the contents of a message, including insertion, deletion, transposition, and modification.
5. **Sequence modification:** Any modification to a sequence of messages between parties, including insertion, deletion, and reordering.
6. **Timing modification:** Delay or replay of messages. In a connection-oriented application, an entire session or sequence of messages could be a replay of some previous valid session, or individual messages in the sequence could be delayed or replayed. In a connectionless application, an individual message (e.g., datagram) could be delayed or replayed.
7. **Source repudiation:** Denial of transmission of message by source.
8. **Destination repudiation:** Denial of receipt of message by destination.

Measures to deal with the first two attacks are in the realm of message confidentiality and

are dealt with using encryption. Measures to deal with items (3) through (6) in the foregoing list are generally regarded as message authentication. Mechanisms for dealing specifically with item (7) come under the heading of digital signatures. Generally, a digital signature technique will also counter some or all of the attacks listed under items (3) through (6). Dealing with item (8) may require a combination of the use of digital signatures and a protocol designed to counter this attack.

Definitions

Message authentication

Verify that received messages have not been altered (i.e., no modification, insertion, deletion, or replay).

Digital signatures

An authentication technique that also includes measures to counter repudiation by the source.

Message authentication is a mechanism or service used to verify the integrity of a message. Message authentication assures that data received are exactly as sent (i.e., there is no modification, insertion, deletion, or replay). In many cases, there is a requirement that the authentication mechanism assures that purported identity of the sender is valid. A **digital signature** is an authentication technique that also includes measures to counter repudiation by the source.

Message authentication functions

= function that produces an authenticator: a value to be used to authenticate a message.

3 classes

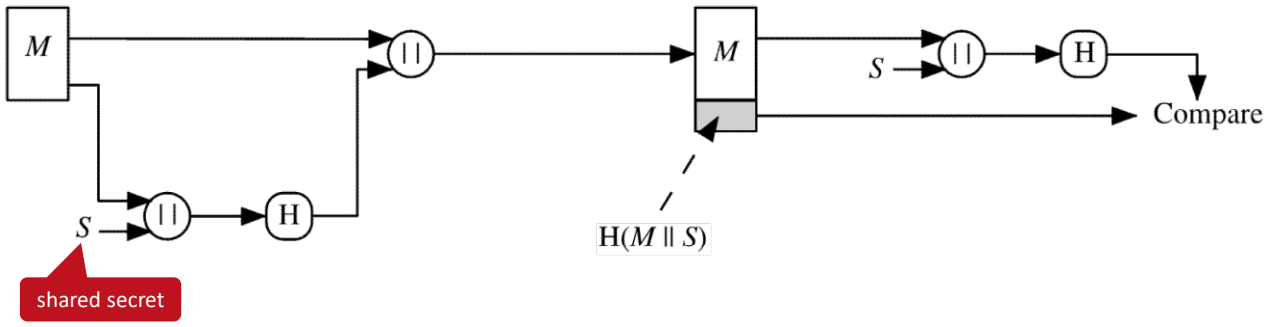
- **Hash function:** Fixed length hash value serves as authenticator
- **Message encryption:** Ciphertext serves as authenticator
- **Message Authentication Code (MAC):** Fixed-length value resulting from message and secret key serves as authenticator

University of Antwerp
Faculty of Science

Any message authentication or digital signature mechanism has two levels of functionality. At the lower level, there must be some sort of function that produces an authenticator: a value to be used to authenticate a message. This lower-level function is then used as a primitive in a higher-level authentication protocol that enables a receiver to verify the authenticity of a message. These may be grouped into three classes:

- **Hash function:** A function that maps a message of any length into a fixed-length hash value, which serves as the authenticator
- **Message encryption:** The ciphertext of the entire message serves as its authenticator
- **Message authentication code (MAC):** A function of the message and a secret key that produces a fixed-length value that serves as the authenticator

Using hash functions for message authentication



University of Antwerp
Faculty of Science

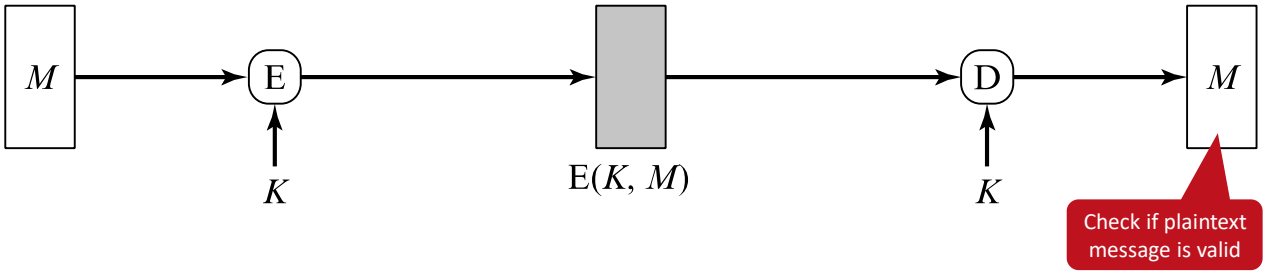
43

When a hash function is used to provide message authentication, the hash function value is often referred to as a **message digest**. The essence of the use of a hash function for message integrity is as follows. The sender computes a hash value as a function of the bits in the message and transmits both the hash value and the message. The receiver performs the same hash calculation on the message bits and compares this value with the incoming hash value. If there is a mismatch, the receiver knows that the message (or possibly the hash value) has been altered. The hash value must be transmitted in a secure fashion. That is, the hash value must be protected so that if an adversary alters or replaces the message, it is not feasible for adversary to also alter the hash value to fool the receiver.

It is possible to use a hash function but no encryption for message authentication. The technique assumes that the two communicating parties share a common secret value *S*. A computes the hash value over the concatenation of *M* and *S* and appends the resulting hash value to *M*. Because B possesses *S*, it can recompute the hash value to verify. Because the secret value itself is not sent, an opponent cannot modify an intercepted message and cannot generate a false message.

Using symmetric encryption for authentication

- Provides both confidentiality and authentication if K is secret
- If the decrypted message M is not meaningful (e.g., random bits) then the message can be considered altered
- Some automated means to determine legitimate plaintext may be needed
- Cannot be used if encryption algorithm is **malleable** (e.g., stream ciphers, or block operation modes such as CBC (partially) and CTR)



University of Antwerp | Faculty of Science

44

Consider the straightforward use of symmetric encryption. A message *M* transmitted from source A to destination B is encrypted using a secret key *K* shared by A and B. If no other party knows the key, then confidentiality is provided: No other party can recover the plaintext of the message. In addition, B is assured that the message was generated by A. Why? The message must have come from A, because A is the only other party that possesses *K* and therefore the only other party with the information necessary to construct ciphertext that can be decrypted with *K*. Furthermore, if *M* is recovered, B knows that none of the bits of *M* have been altered, because an opponent that does not know *K* would not know how to alter bits in the ciphertext to produce the desired changes in the plaintext.

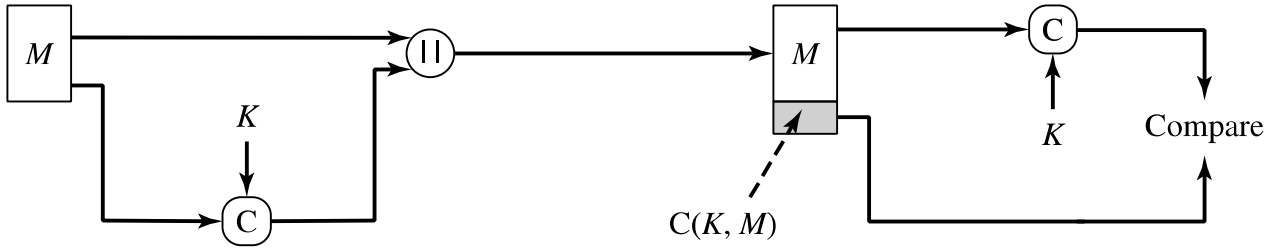
So we may say that symmetric encryption provides authentication as well as confidentiality. However, this flat statement needs to be qualified. Consider exactly what is happening at B. Given a decryption function D and a secret key *K*, the destination will accept *any* input *X* and produce output $Y = D(K, X)$. If *X* is the ciphertext of a legitimate message *M* produced by the corresponding encryption function, then *Y* is some plaintext message *M*. Otherwise, *Y* will likely be a meaningless sequence of bits. There may need to be some automated means of determining at B whether *Y* is legitimate plaintext and therefore must have come from A.

It may be difficult to determine *automatically* if incoming ciphertext decrypts to intelligible plaintext. If the plaintext is, say, a binary object file or digitized X-rays, determination of properly formed and therefore authentic plaintext may be difficult. Thus, an opponent could achieve a certain level of disruption simply by issuing messages with random content purporting to come from a legitimate user. One solution to this problem is to force the plaintext to have some structure that is easily recognized but that cannot be replicated

without recourse to the encryption function. We could, for example, append an error-detecting code, also known as a frame check sequence (FCS) or checksum, to each message before encryption.

Message authentication code (MAC)

- Alternative method to provide message authentication
- Generates a fixed-size **cryptographic checksum** based on the message and a secret key
- Does **not** need to be **reversible**, in contrast to encryption



University of Antwerp
Faculty of Science

45

An alternative authentication technique involves the use of a secret key to generate a small fixed-size block of data, known as a cryptographic checksum or MAC, that is appended to the message. This technique assumes that two communicating parties, say A and B, share a common secret key *K*. When A has a message to send to B, it calculates the MAC as a function of the message and the key:

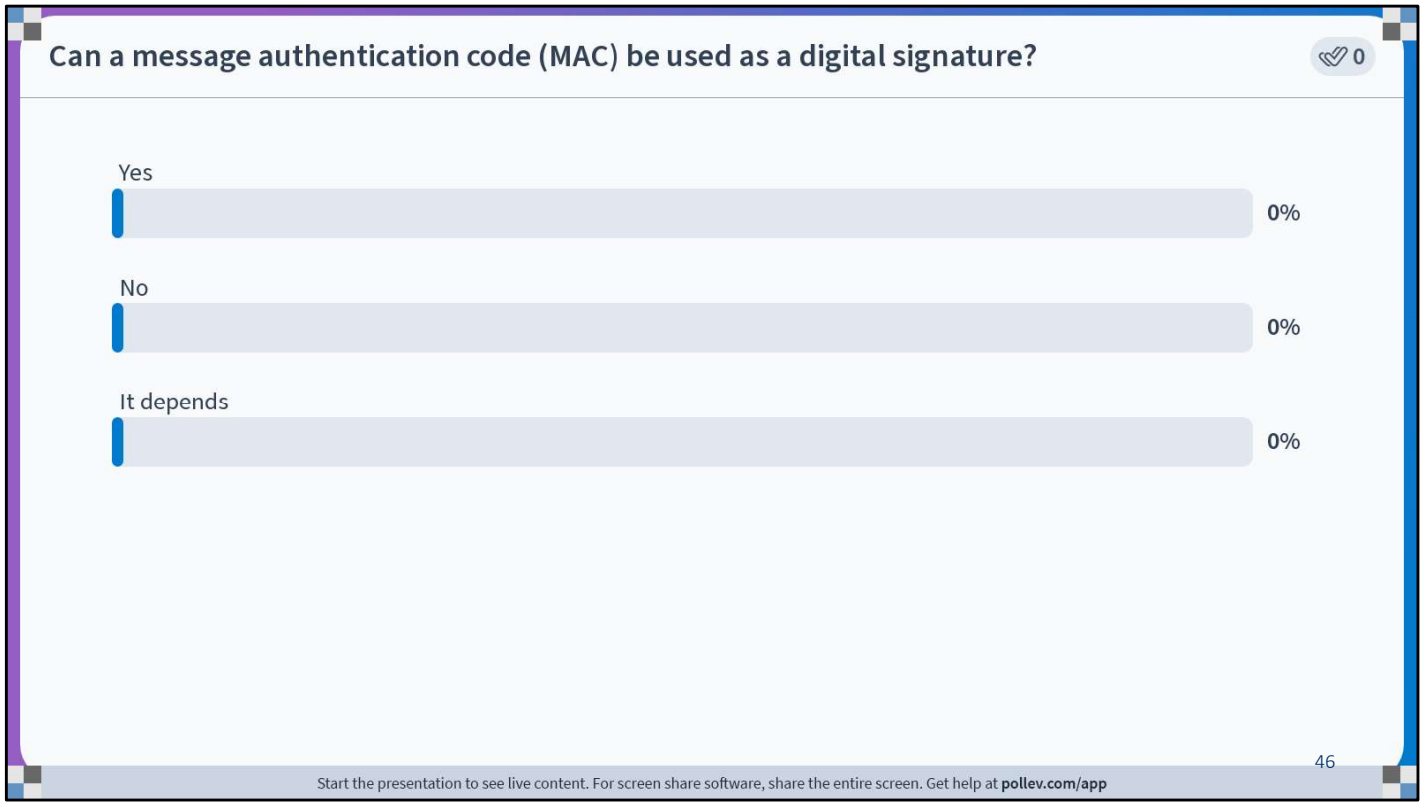
MAC = C(*K*, *M*)

The message plus MAC are transmitted to the intended recipient. The recipient performs the same calculation on the received message, using the same secret key, to generate a new MAC. The received MAC is compared to the calculated MAC. If we assume that only the receiver and the sender know the identity of the secret key, and if the received MAC matches the calculated MAC, then:

1. The receiver is assured that the message has not been altered. If an attacker alters the message but does not alter the MAC, then the receiver's calculation of the MAC will differ from the received MAC. Because the attacker is assumed not to know the secret key, the attacker cannot alter the MAC to correspond to the alterations in the message.
2. The receiver is assured that the message is from the alleged sender. Because no one else knows the secret key, no one else could prepare a message with a proper MAC.
3. If the message includes a sequence number (such as is used with HDLC, X.25, and TCP), then the receiver can be assured of the proper sequence because an attacker cannot successfully alter the sequence number.

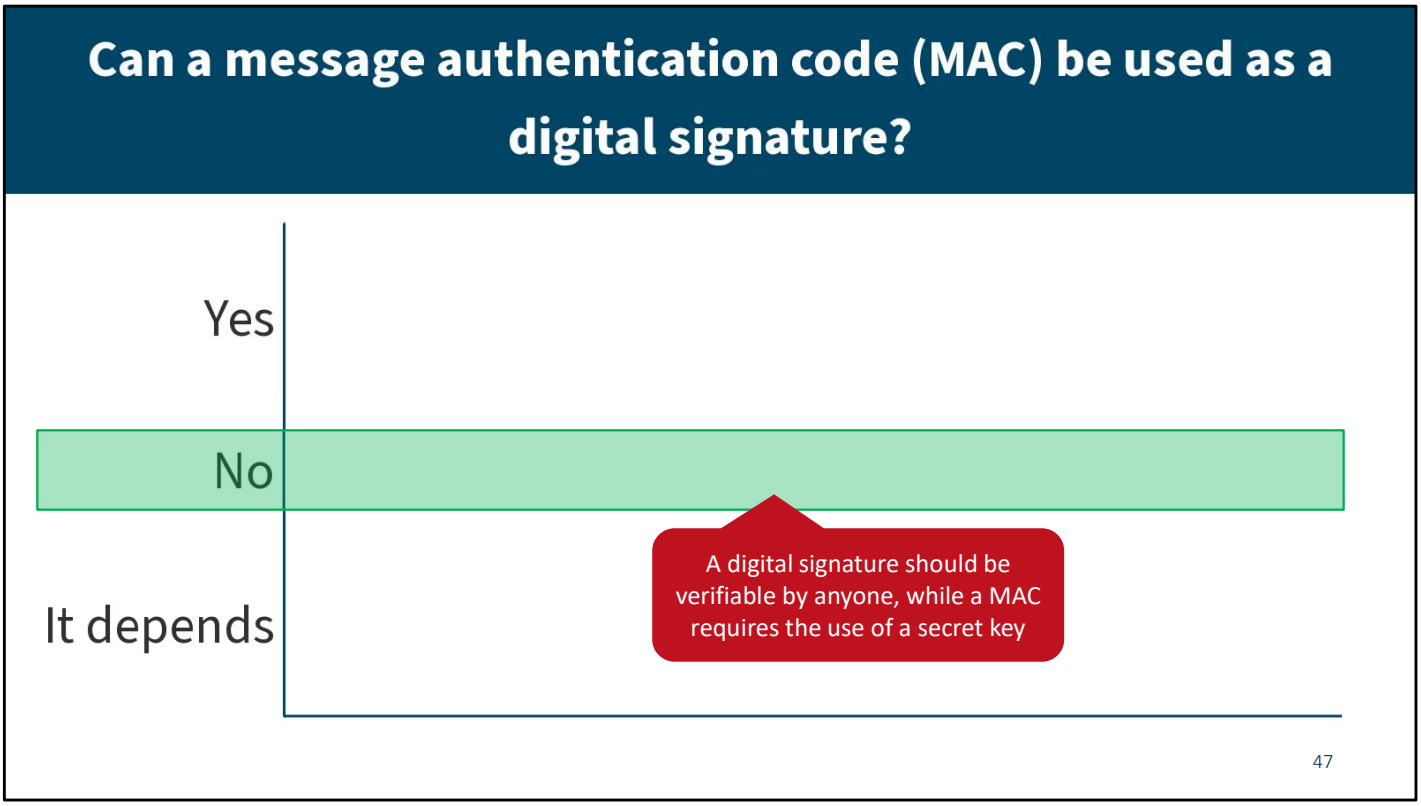
A MAC function is similar to encryption. One difference is that the MAC algorithm need not be reversible, as it must be for decryption. In general, the MAC function is a many-to-one function. The domain of the function consists of messages of some arbitrary length, whereas the range consists of all possible MACs and all possible keys. If an n-bit MAC is used, then there are 2^n possible MACs, whereas there are N possible messages with $N >> 2^n$. Furthermore, with a k-bit key, there are 2^k possible keys.

It turns out that, because of the mathematical properties of the authentication function, it is less vulnerable to being broken than encryption.



Poll Title: Do not modify the notes in this section to avoid tampering with the Poll Everywhere activity.
More info at polleverywhere.com/support

Can a message authentication code (MAC) be used as a digital signature?
https://www.polleverywhere.com/multiple_choice_polls/d3EysrCzQ0wqzvP



Poll Title: Can a message authentication code (MAC) be used as a digital signature?

Security requirements for MAC functions

Given a MAC function, and an opponent who knows the MAC, but does not know the secret key K, the following requirements should be satisfied:

1. If an opponent observes M and $MAC(K, M)$ it should be infeasible to construct M', such that $MAC(K, M') = MAC(K, M)$
2. Given two randomly chosen messages M and M', the probability that $MAC(K, M) = MAC(K, M')$ should be 2^{-n} (for an n-bit code)
3. If M' is a known transformation of M (i.e., $M' = f(M)$), then $Pr[MAC(K, M) = MAC(K, M')] = 2^{-n}$

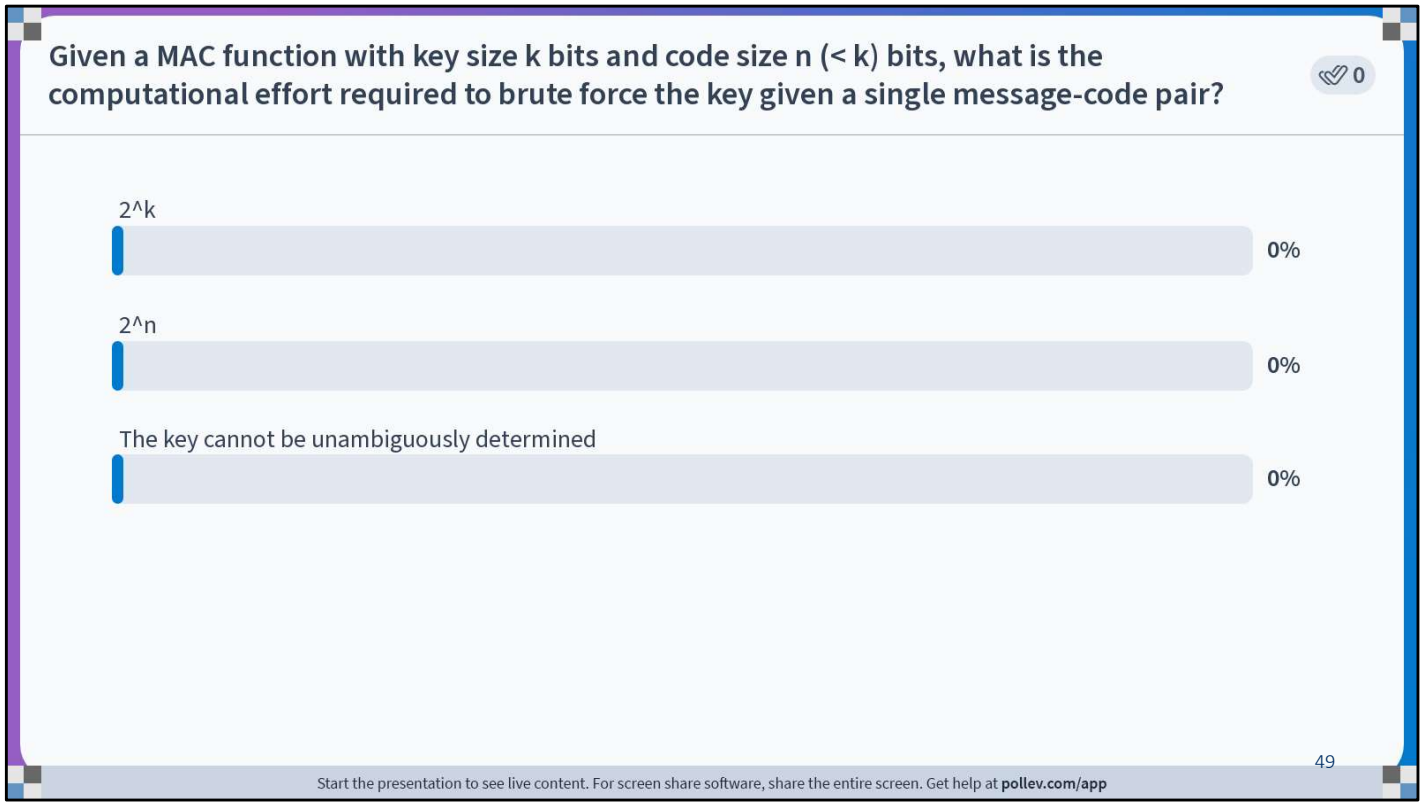
University of Antwerp | Faculty of Science

48

The **first requirement** avoids attacks in which an opponent is able to construct a new message to match a given tag, even though the opponent does not know and does not learn the key.

The **second requirement** deals with the need to thwart a brute-force attack based on chosen plaintext. That is, if we assume that the opponent does not know *K* but does have access to the MAC function and can present messages for MAC generation, then the opponent could try various messages until finding one that matches a given tag. If the MAC function exhibits uniform distribution, then a brute-force method would require, on average, $2^{(n-1)}$ attempts before finding a message that fits a given tag.

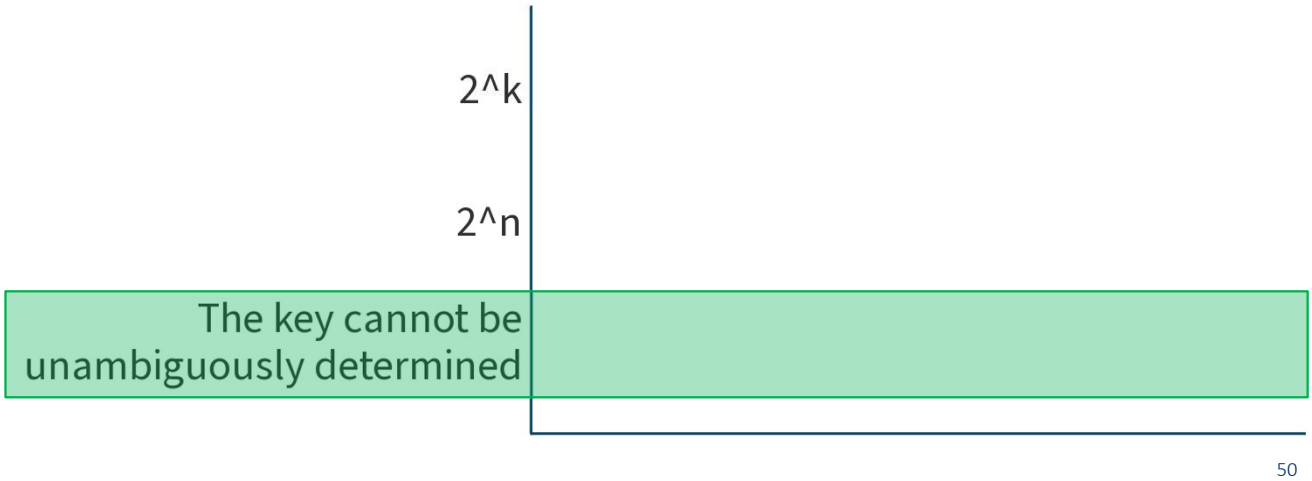
The **final requirement** dictates that the authentication algorithm should not be weaker with respect to certain parts or bits of the message than others. If this were not the case, then an opponent who had *M* and MAC(*K, M)* could attempt variations on *M* at the known "weak spots" with a likelihood of early success at producing a new message that matched the old tags.



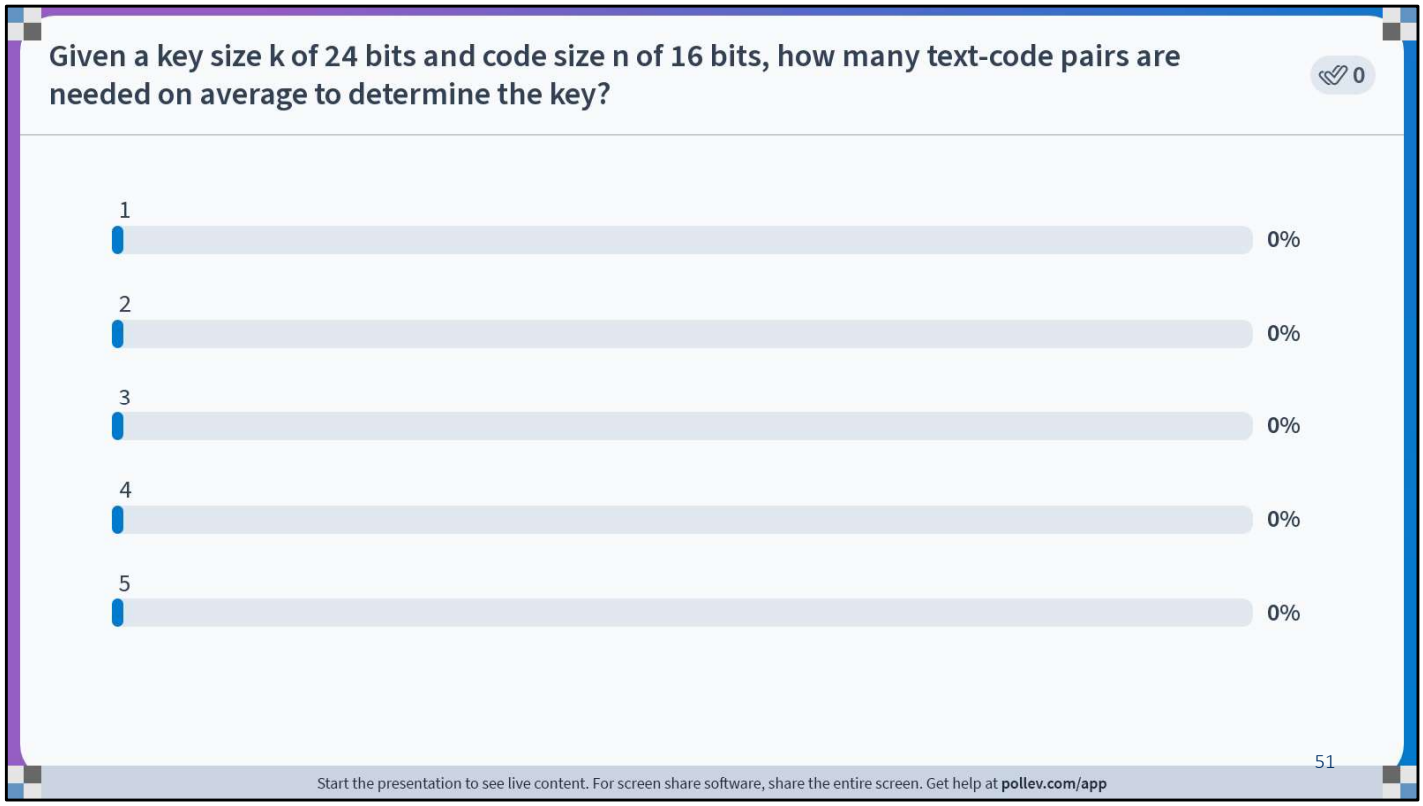
Poll Title: Do not modify the notes in this section to avoid tampering with the Poll Everywhere activity.
More info at polleverywhere.com/support

Given a MAC function with key size k bits and code size n (< k) bits, what is the computational effort required to brute force the key given a single message-code pair?
https://www.polleverywhere.com/multiple_choice_polls/nbcKlWo4hbdCahZ

Given a MAC function with key size k bits and code size n (< k) bits, what is the computational effort required to brute force the key given a single message-code pair?



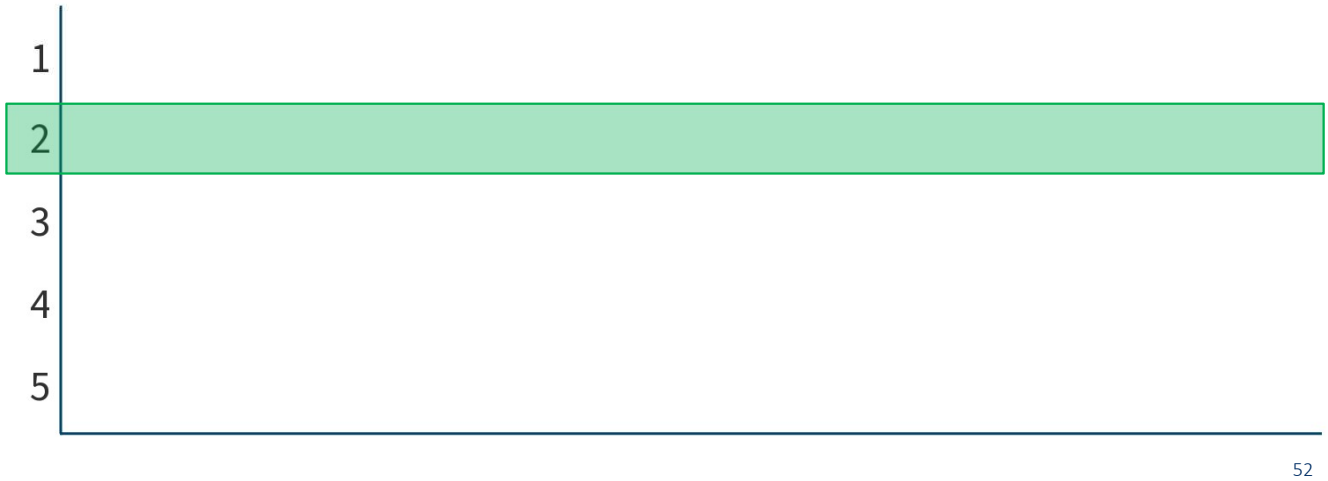
Poll Title: Given a MAC function with key size k bits and code size n (< k) bits, what is the computational effort required to brute force the key given a single message-code pair?



Poll Title: Do not modify the notes in this section to avoid tampering with the Poll Everywhere activity.
More info at polleverywhere.com/support

Given a key size k of 24 bits and code size n of 16 bits, how many text-code pairs are needed on average to determine the key?
https://www.polleverywhere.com/multiple_choice_polls/VNCuvZLfqYxCros

Given a key size k of 24 bits and code size n of 16 bits, how many text-code pairs are needed on average to determine the key?



Poll Title: Given a key size k of 24 bits and code size n of 16 bits, how many text-code pairs are needed on average to determine the key?

Solution: Brute force attacks on MACs

Round 1

Given M_1, $T_1 = MAC(K, M_1)$

Compute $T_i = MAC(K_i, M_1)$ for all possible 2^k keys

Number of average matching keys $\approx 2^{k-n} = 2^8$

Round 2

Given M_2, $T_2 = MAC(K, M_2)$

Compute $T_i = MAC(K_i, M_2)$ for the 2^8 remaining keys

Number of average matching keys $\approx 2^{8-n} = 2^{-8} \approx 1$

On average, 2 text-code pairs are needed

If k < n, then on average only 1 pair is needed

If an attacker can determine the MAC key, then it is possible to generate a valid MAC value for any input *x*. Suppose the key size is *k* bits and that the attacker has one known text-tag pair. Then the attacker can compute the *n*-bit tag on the known text for all possible keys. At least one key is guaranteed to produce the correct tag, namely, the valid key that was initially used to produce the known text-tag pair. This phase of the attack takes a level of effort proportional to 2^k (that is, one operation for each of the 2^k possible key values). However, as was described earlier, because the MAC is a many-to-one mapping, there may be other keys that produce the correct value. Thus, if more than one key is found to produce the correct value, additional text-tag pairs must be tested. It can be shown that the level of effort drops off rapidly with each additional text-MAC pair and that the overall level of effort is roughly 2^k.

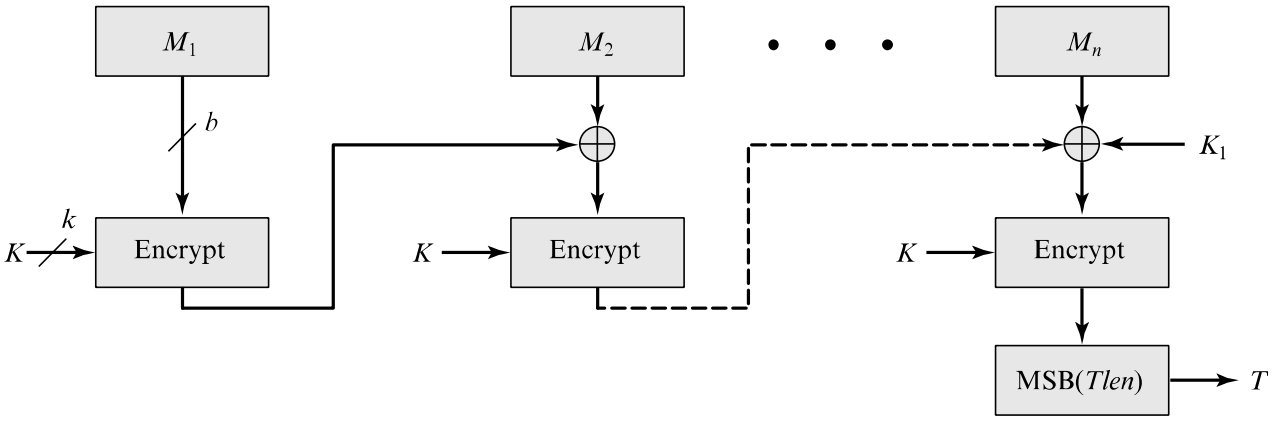
1 Message Authentication Basics

2 Message Authentication Codes

3 Authenticated Encryption

University of Antwerp
Faculty of Science

Cipher-Based Message Authentication Code (CMAC)



University of Antwerp
Faculty of Science

55

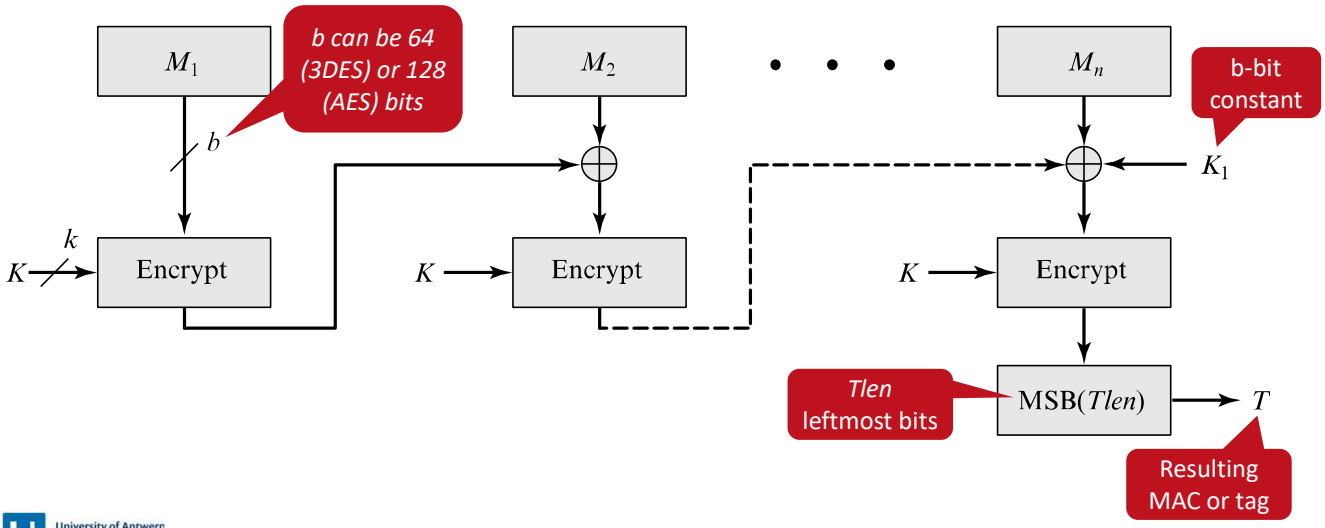
First, let us define the operation of CMAC when the message is an integer multiple n of the cipher block length b. For AES, b = 128, and for triple DES, b = 64. The message is divided into n blocks (M_1, M_2,… , M_n). The algorithm makes use of a k-bit encryption key K and a b-bit constant, K_1. For AES, the key size k is 128, 192, or 256 bits; for triple DES, the key size is 112 or 168 bits. CMAC is calculated as follows:

$C1 = E(K, M1)$
$C2 = E(K, [M2 \oplus C1])$
$C3 = E(K, [M3 \oplus C2])$
…
$C_n = E(K, [M_n \oplus C_{n-1} \oplus K_1])$ $T = MSB_{Tlen}(C_n)$

where

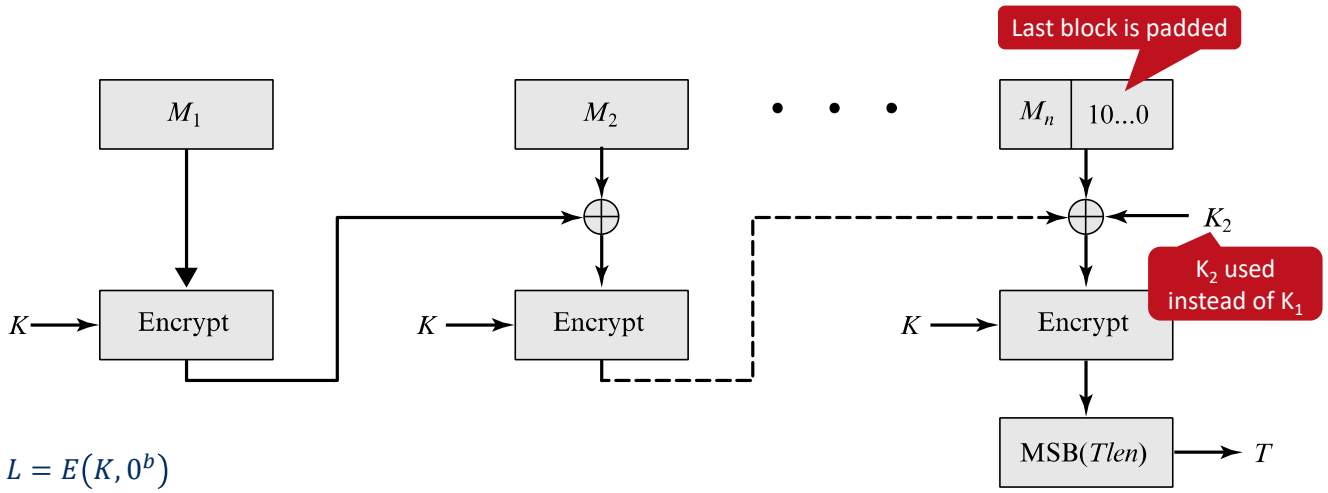
T = message authentication code, also referred to as the tag
T_{len} = bit length of T
$MSB_s(X)$ = the s leftmost bits of the bit string X

Cipher-Based Message Authentication Code (CMAC)



University of Antwerp | Faculty of Science

CMAC if message length is not an integer multiple of b



$L = E(K, 0^b)$

$K_1 = L \cdot x$ (Multiplication in GF(2^b))

$K_2 = L \cdot x^2$

University of Antwerp | Faculty of Science

57

If the message is not an integer multiple of the cipher block length, then the final block is padded to the right (least significant bits) with a 1 and as many 0s as necessary so that the final block is also of length b. The CMAC operation then proceeds as before, except that a different b-bit key K_2 is used instead of K_1. The two b-bit keys are derived from the k-bit encryption key as follows:

$L = E(K, 0^b)$

$K_1 = L * x$

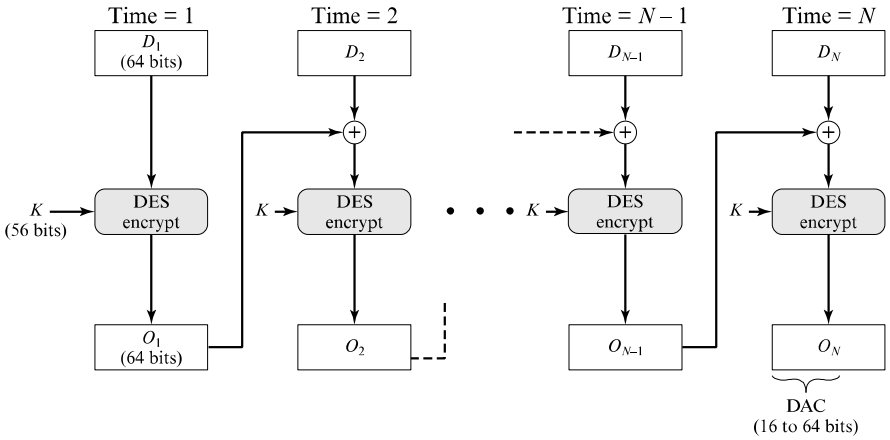
$K_2 = L * x^2 = (L * x) * x$

where multiplication (*) is done in the finite field GF(2^b) and x and x^2 are first- and second-order polynomials that are elements of GF(2^b). Thus, the binary representation of x consists of b - 2 zeros followed by 10; the binary representation of x^2 consists of b - 3 zeros followed by 100. The finite field is defined with respect to an irreducible polynomial that is lexicographically first among all such polynomials with the minimum possible number of nonzero terms. For the two approved block sizes, the polynomials are $x^{64} + x^4 + x^3 + x + 1$ and $x^{128} + x^7 + x^2 + x + 1$.

To generate K_1 and K_2, the block cipher is applied to the block that consists entirely of 0 bits. The first subkey is derived from the resulting ciphertext by a left shift of one bit and, conditionally, by XORing a constant that depends on the block size. The second subkey is derived in the same manner from the first subkey.

Why the need for two different constants K_1 and K_2?

- The Data Authentication Algorithm (DAA), CMAC's predecessor, does not use these constants, which makes it insecure for messages that are not a multiple of the block size
- As a simple example, for a single block message X, the following equation holds when using DAA:
  $MAC(\mathrm{K}, X) = T = MAC(\mathrm{K}, X||X \oplus T)$



It has been demonstrated that DAA is secure under a reasonable set of security criteria, with the following restriction. Only messages of one fixed length of *mn* bits are processed, where *n* is the cipher block size and *m* is a fixed positive integer. As a simple example, notice that given the CBC MAC of a one-block message *X*, say *T* = MAC(*K*, *X*), the adversary immediately knows the CBC MAC for the two-block message *X* || (*X* ⊕ *T*) since this is once again *T*.

Black and Rogaway demonstrated that this limitation could be overcome using three keys: one key *K* of length *k* to be used at each step of the cipher block chaining and two keys of length *b*, where *b* is the cipher block length. This proposed construction was refined by Iwata and Kurosawa so that the two *n*-bit keys could be derived from the encryption key, rather than being provided separately. This refinement, adopted by NIST, is the **Cipher-based Message Authentication Code (CMAC)** mode of operation for use with AES and triple DES. It is specified in NIST Special Publication 800-38B.

Hash-function based MAC: HMAC

- Several **advantages** compared to traditional MACs, based on symmetric block ciphers:
  1. Cryptographic hash functions, e.g., SHA, generally execute faster
  2. Library code for hash functions is widely available
- However, there are some issues to overcome as hash functions are not designed to **rely on a secret key**
- HMAC was first published in 1996 and is used for IP security (IPSec), TLS and SSL authentication

MAC based on the use of a symmetric block cipher has traditionally been the most common approach to constructing a MAC. In recent years, there has been increased interest in developing a MAC derived from a cryptographic hash function. The motivations for this interest are:

1. Cryptographic hash functions such as MD5 and SHA generally execute faster in software than symmetric block ciphers such as DES.
2. Library code for cryptographic hash functions is widely available.

With the development of AES and the more widespread availability of code for encryption algorithms, these considerations are less significant, but hash-based MACs continue to be widely used.

A hash function such as SHA was not designed for use as a MAC and cannot be used directly for that purpose, because it does not rely on a secret key. There have been a number of proposals for the incorporation of a secret key into an existing hash algorithm. The approach that has received the most support is HMAC. HMAC has been issued as RFC 2104, has been chosen as the mandatory-to-implement MAC for IP security, and is used in other Internet protocols, such as SSL. HMAC has also been issued as a NIST standard (FIPS 198).

HMAC design objectives

RFC 2104 lists 5 objectives for HMAC:

1. To use, without modification, available hash functions
2. To allow for easy replaceability of the hash function
3. To preserve the hash function's original performance
4. To use and handle keys in a simple way
5. To have a well understood cryptographic analysis

University of Antwerp | Faculty of Science

60

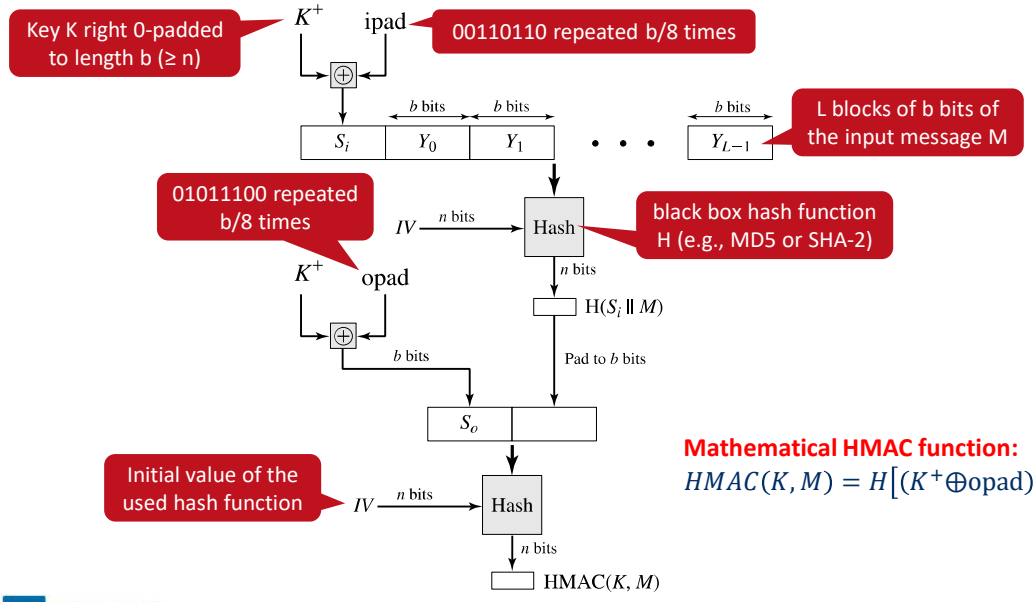
RFC 2104 lists the following design objectives for HMAC:

- To use, without modifications, available hash functions. In particular, to use hash functions that perform well in software and for which code is freely and widely available.
- To allow for easy replaceability of the embedded hash function in case faster or more secure hash functions are found or required.
- To preserve the original performance of the hash function without incurring a significant degradation.
- To use and handle keys in a simple way.
- To have a well understood cryptographic analysis of the strength of the authentication mechanism based on reasonable assumptions about the embedded hash function.

The first two objectives are important to the acceptability of HMAC. HMAC treats the hash function as a "black box." This has two benefits. First, an existing implementation of a hash function can be used as a module in implementing HMAC. In this way, the bulk of the HMAC code is prepackaged and ready to use without modification. Second, if it is ever desired to replace a given hash function in an HMAC implementation, all that is required is to remove the existing hash function module and drop in the new module. This could be done if a faster hash function were desired. More important, if the security of the embedded hash function were compromised, the security of HMAC could be retained simply by replacing the embedded hash function with a more secure one (e.g., replacing SHA-2 with SHA-3).

The last design objective in the preceding list is, in fact, the main advantage of HMAC over other proposed hash-based schemes. HMAC can be proven secure provided that the embedded hash function has some reasonable cryptographic strengths.

HMAC can be used with any hash function



Mathematical HMAC function:

$$HMAC(K, M) = H\left[(K^+ \oplus \text{opad}) || H[(K^+ \oplus \text{ipad}) || M]\right]$$

University of Antwerp | Faculty of Science

61

Define the following terms:

- H = embedded hash function (e.g., MD5, SHA-1, RIPEMD-160)
- IV = initial value input to hash function
- M = message input to HMAC (including the padding specified in the embedded hash function)
- Y_i = i-th block of M, 0 <= i <= (L - 1)
- L = number of blocks in M
- b = number of bits in a block
- n = length of hash code produced by embedded hash function
- K = secret key; recommended length is >= n; if key length is greater than b, the key is input to the hash function to produce an n-bit key
- K^+ = K padded with zeros on the right so that the result is b bits in length
- ipad = 00110110 (36 in hexadecimal) repeated b/8 times
- opad = 01011100 (5C in hexadecimal) repeated b/8 times

Then HMAC can be expressed as:

$$\text{HMAC}(K, M) = \text{H}[(K+ \oplus \text{opad}) || \text{H}[(K+ \oplus \text{ipad}) || M]]$$

We can describe the algorithm as follows:

1. Append zeros to the left end of K to create a b-bit string $K+$ (e.g., if K is of length 160 bits and b = 512, then K will be appended with 44 zeroes).
2. XOR (bitwise exclusive-OR) K^+ with ipad to produce the b-bit block S_i.
3. Append M to S_i.
4. Apply H to the stream generated in step 3.

5. XOR $K+$ with opad to produce the b-bit block S_o.
6. Append the hash result from step 4 to S_o.
7. Apply H to the stream generated in step 6 and output the result.

Note that the XOR with ipad results in flipping one-half of the bits of K. Similarly, the XOR with opad results in flipping one-half of the bits of K, using a different set of bits. In effect, by passing S_i and S_o through the compression function of the hash algorithm, we have pseudorandomly generated two keys from K.

HMAC should execute in approximately the same time as the embedded hash function for long messages. HMAC adds three executions of the hash compression function (for S_i, S_o, and the block produced from the inner hash).

Cryptographic strength of HMAC

- HMAC can be proven secure provided that the embedded hash function has some reasonable cryptographic strengths
- A successful attack on HMAC is equivalent to one of the following attacks on the hash function:
  - The attacker is able to compute an output of the compression function (with secret IV)
  - The attacker finds collisions in the hash function
- The second attack requires $2^{n/2}$ attempts (i.e., birthday attack)
  - Not trivial as message-code pairs cannot be generated offline (as attacker does not know K)
  - On a 1 Gbps link, using MD5 (using a 128-bit hash code) this would take ± 150.000 years without a change in key

The security of any MAC function based on an embedded hash function depends in some way on the cryptographic strength of the underlying hash function. The appeal of HMAC is that its designers have been able to prove an exact relationship between the strength of the embedded hash function and the strength of HMAC.

The security of a MAC function is generally expressed in terms of the probability of successful forgery with a given amount of time spent by the forger and a given number of message-tag pairs created with the same key. In essence, it is proved in that for a given level of effort (time, message–tag pairs) on messages generated by a legitimate user and seen by the attacker, the probability of successful attack on HMAC is equivalent to one of the following attacks on the embedded hash function:

1. The attacker is able to compute an output of the compression function even with an *IV* that is random, secret, and unknown to the attacker.
2. The attacker finds collisions in the hash function even when the *IV* is random and secret.

In the first attack, we can view the compression function as equivalent to the hash function applied to a message consisting of a single *b*-bit block. For this attack, the *IV* of the hash function is replaced by a secret, random value of *n* bits. An attack on this hash function requires either a brute-force attack on the key, which is a level of effort on the order of 2^n, or a birthday attack, which is a special case of the second attack, discussed next.

In the second attack, the attacker is looking for two messages *M* and *M*' that produce the same hash: H(*M*) = H(*M*'). This is the birthday attack discussed before. We have shown that this requires a level of effort of $2^{n/2}$ for a hash length of *n*. On this basis, the security of MD5

is called into question, because a level of effort of 2^{64} looks feasible with today's technology. Does this mean that a 128-bit hash function such as MD5 is unsuitable for HMAC? The answer is no, because of the following argument. To attack MD5, the attacker can choose any set of messages and work on these off line on a dedicated computing facility to find a collision. Because the attacker knows the hash algorithm and the default *IV*, the attacker can generate the hash code for each of the messages that the attacker generates. However, when attacking HMAC, the attacker cannot generate message/code pairs off line because the attacker does not know *K*. Therefore, the attacker must observe a sequence of messages generated by HMAC under the same key and perform the attack on these known messages. For a hash code length of 128 bits, this requires 264 observed blocks (272 bits) generated using the same key. On a 1-Gbps link, one would need to observe a continuous stream of messages with no change in key for about 150,000 years in order to succeed. Thus, if speed is a concern, it is fully acceptable to use MD5 rather than SHA-1 as the embedded hash function for HMAC.

1 Message Authentication Basics

2 Message Authentication Codes

3 Authenticated Encryption

University of Antwerp
Faculty of Science

Authenticated Encryption (AE)

- **Definition:** Encryption systems that simultaneously protect confidentiality and authenticity (integrity)
- Four main methods
  - Hashing followed by encryption (H → E): $C = E(K, M||H(M))$
    - Used by Wi-Fi WEP → **Was shown to have fundamental flaws!**
  - Authentication followed by encryption (A → E): $C = E(K_2, M||MAC(K_1, M))$
    - Used by SSL/TLS
  - Encryption followed by authentication (E → A): $C = E(K_2, M), T = MAC(K_1, C)$
    - Used by IPSec
  - Independently encrypt and authenticate (E + A) : $C = E(K_2, M), T = MAC(K_1, M)$
    - Used by SSH
- Two approaches for AE have been standardized by NIST
  - Counter with Cipher Block Chaining-Message Authentication Code (CCM)
  - Galois/Counter Mode (GCM)

University of Antwerp | Faculty of Science

64

Authenticated encryption (AE) is a term used to describe encryption systems that simultaneously protect confidentiality and authenticity (integrity) of communications. Many applications and protocols require both forms of security, but until recently the two services have been designed separately. There are four common approaches to providing both confidentiality and encryption for a message *M*.

- **Hashing followed by encryption (H → E):** First compute the cryptographic hash function over *M* as *h* = *H*(*M*). Then encrypt the message plus hash function: E(*K*, (*M* || *h*)).
- **Authentication followed by encryption (A → E):** Use two keys. First authenticate the plaintext by computing the MAC value as *T* = MAC(K_1, *M*). Then encrypt the message plus tag: E(K_2, [*M* || *T*]). This approach is taken by the SSL/TLS protocols.
- **Encryption followed by authentication (E → A):** Use two keys. First encrypt the message to yield the ciphertext *C* = E(K_2, *M*). Then authenticate the ciphertext with *T* = MAC(K_1, *C*) to yield the pair (*C, T*). This approach is used in the IPSec protocol.
- **Independently encrypt and authenticate (E + A).** Use two keys. Encrypt the message to yield the ciphertext *C* = E(*K*2, *M*). Authenticate the plaintext with *T* = MAC(*K*1, *M*) to yield the pair (*C, T*). These operations can be performed in either order. This approach is used by the SSH protocol.

Both decryption and verification are straightforward for each approach. For H → E, A → E, and E + A, decrypt first, then verify. For E → A, verify first, then decrypt. There are security vulnerabilities with all of these approaches. The H → E approach is used in the Wired Equivalent Privacy (WEP) protocol to protect WiFi networks. This approach had fundamental weaknesses and led to the replacement of the WEP protocol. [BLAC05] and [BELL00] point out that there are security concerns in each of the three encryption/MAC approaches listed

above. Nevertheless, with proper design, any of these approaches can provide a high level of security. This is the goal of the two approaches discussed in this section, both of which have been standardized by NIST.

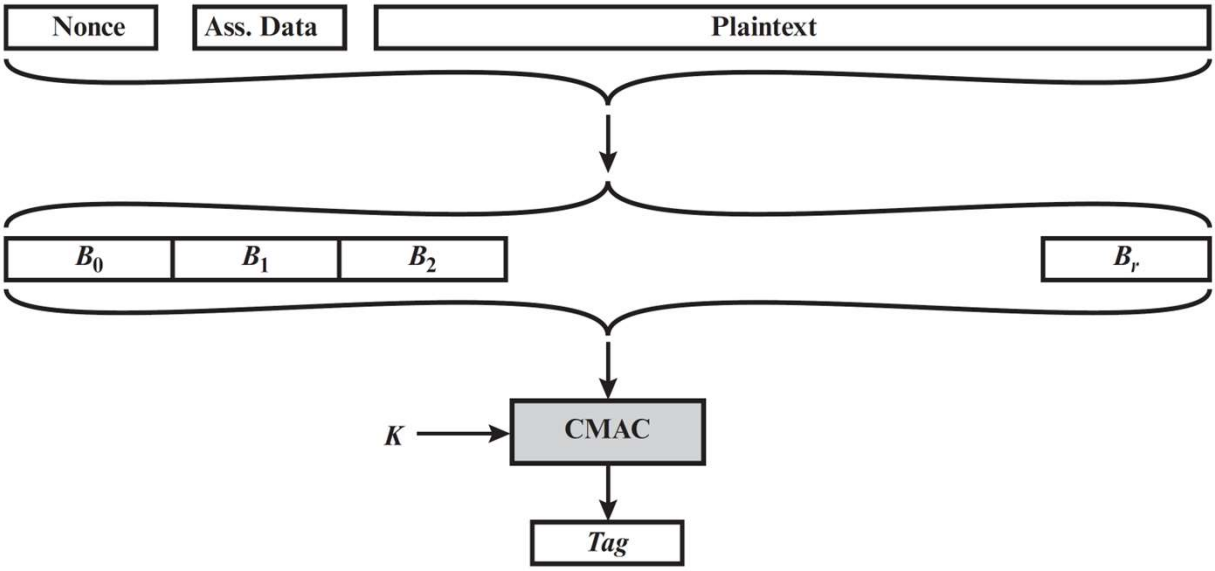
Counter with Cipher Block Chaining-Message Authentication Code (CCM)

- Variation of the encrypt-and-MAC approach (E+A)
- Originally developed for IEEE 802.11 Wi-Fi WPA security
- Standardized as NIST SP-800-38C
- Combines AES, CTR mode, and CMAC authentication
- 4 inputs
  - A single key K for both encryption and MAC
  - The plaintext message P (to be encrypted and authenticated)
  - Associated data A (only to be authenticated, e.g., protocol headers)
  - A unique nonce N that is associated to P and A (against replay attacks)

The CCM mode of operation was standardized by NIST specifically to support the security requirements of IEEE 802.11 WiFi wireless local area networks, but can be used in any networking application requiring authenticated encryption. CCM is a variation of the encrypt-and-MAC approach to authenticated encryption. It is defined in NIST SP 800-38C. The key algorithmic ingredients of CCM are the AES encryption algorithm, the CTR mode of operation (Chapter 7), and the CMAC authentication algorithm. A single key *K* is used for both encryption and MAC algorithms. The input to the CCM encryption process consists of three elements.

1. Data that will be both authenticated and encrypted. This is the plaintext message *P* of data block.
2. Associated data *A* that will be authenticated but not encrypted. An example is a protocol header that must be transmitted in the clear for proper protocol operation but which needs to be authenticated.
3. A nonce *N* that is assigned to the payload and the associated data. This is a unique value that is different for every instance during the lifetime of a protocol association and is intended to prevent replay attacks and certain other types of attacks.

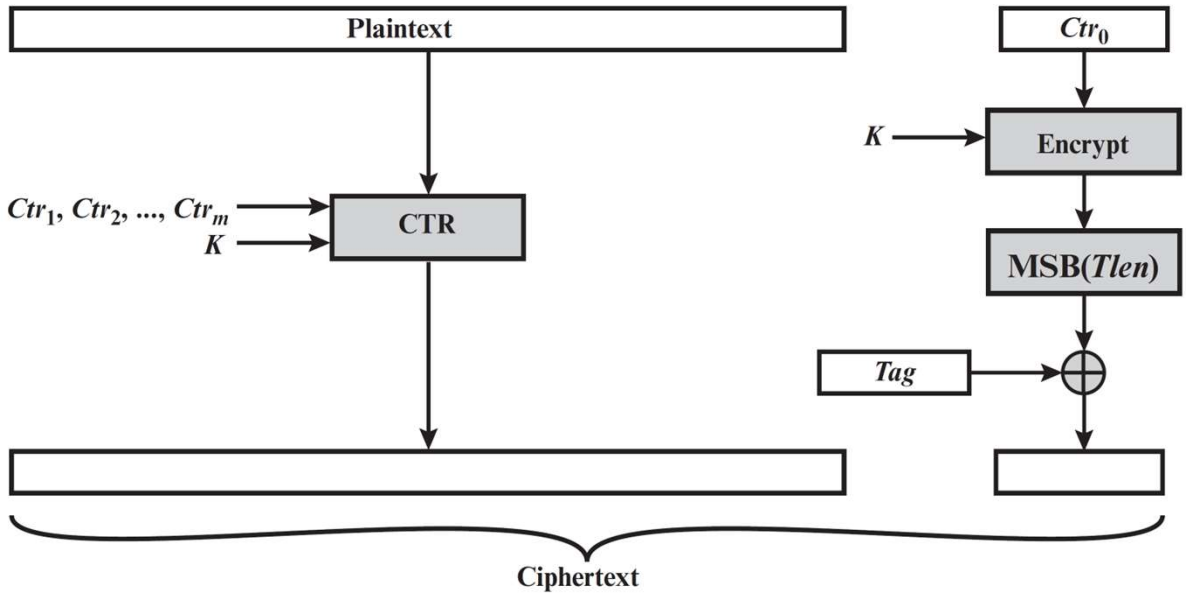
CCM authentication



University of Antwerp
Faculty of Science

For authentication, the input includes the nonce, the associated data, and the plaintext. This input is formatted as a sequence of blocks B_0 through B_r. The first block contains the nonce plus some formatting bits that indicate the lengths of the *N, A,* and *P* elements. This is followed by zero or more blocks that contain *A*, followed by zero of more blocks that contain *P*. The resulting sequence of blocks serves as input to the CMAC algorithm, which produces a MAC value with length *Tlen,* which is less than or equal to the block length.

CCM encryption



University of Antwerp
Faculty of Science

67

For encryption, a sequence of counters is generated that must be independent of the nonce. The authentication tag is encrypted in CTR mode using the single counter Ctr_0. The *Tlen* most significant bits of the output are XORed with the tag to produce an encrypted tag. The remaining counters are used for the CTR mode encryption of the plaintext. The encrypted plaintext is concatenated with the encrypted tag to form the ciphertext output.

SP 800-38C defines the authentication/encryption process as follows:

1. Apply the formatting function to (N, A, P) to produce the blocks $B_0, B_1, ..., B_r$.
2. Set $Y_0 = E(K, B_0)$.
3. For $i = 1$ to r, do $Y_i = E(K, (B_i \oplus Y_{i-1}))$.
4. Set $T = MSB_{Tlen}(Y_r)$.
5. Apply the counter generation function to generate the counter blocks $Ctr_0, Ctr_1, ..., Ctr_m$, where $m = ceil(P_{len}/128)$.
6. For $j = 0$ to m, do $S_j = E(K, Ctr_j)$.
7. Set $S = S_1 \,||\, S_2 \,||\, ... \,||\, S_m$.
8. Return $C = (P \oplus MSB_{Plen}(S)) \,||\, (T \oplus MSB_{Tlen}(S_0))$.

For decryption and verification, the recipient requires the following input: the ciphertext C, the nonce N, the associated data A, the key K, and the initial counter Ctr_0. The steps are as follows:

1. If $C_{len} \leq T_{len}$, then return INVALID.
2. Apply the counter generation function to generate the counter blocks $Ctr_0, Ctr_1, ..., Ctr_m$, where $m = ceil(C_{len}/128)$.
3. For $j = 0$ to m, do $S_j = E(K, Ctr_j)$.

4. Set $S = S_1 \,||\, S_2 \,||\, ... \,||\, S_m$.
5. Set $P = \text{MSB}_{Clen\text{-}Tlen}(C) \oplus \text{MSB}_{Clen\text{-}Tlen}(S)$.
6. Set $T = \text{LSB}_{Tlen}(C) \oplus \text{MSB}_{Tlen}(S_0)$.
7. Apply the formatting function to *(N, A, P)* to produce the blocks $B_0, B_1, ..., B_r$.
8. Set $Y_0 = \text{E}(K, B_0)$.
9. For $i = 1$ to r do $Y_i = \text{E}(K, (B_i \oplus Y_{i-1}))$.
10. If $T \neq \text{MSB}_{Tlen}(Y_r)$, then return INVALID, else return P.

CCM is a relatively complex algorithm. Note that it requires two complete passes through the plaintext, once to generate the MAC value, and once for encryption. Further, the details of the specification require a tradeoff between the length of the nonce and the length of the tag, which is an unnecessary restriction. Also note that the encryption key is used twice with the CTR encryption mode: once to generate the tag and once to encrypt the plaintext plus tag. Whether these complexities add to the security of the algorithm is not clear.

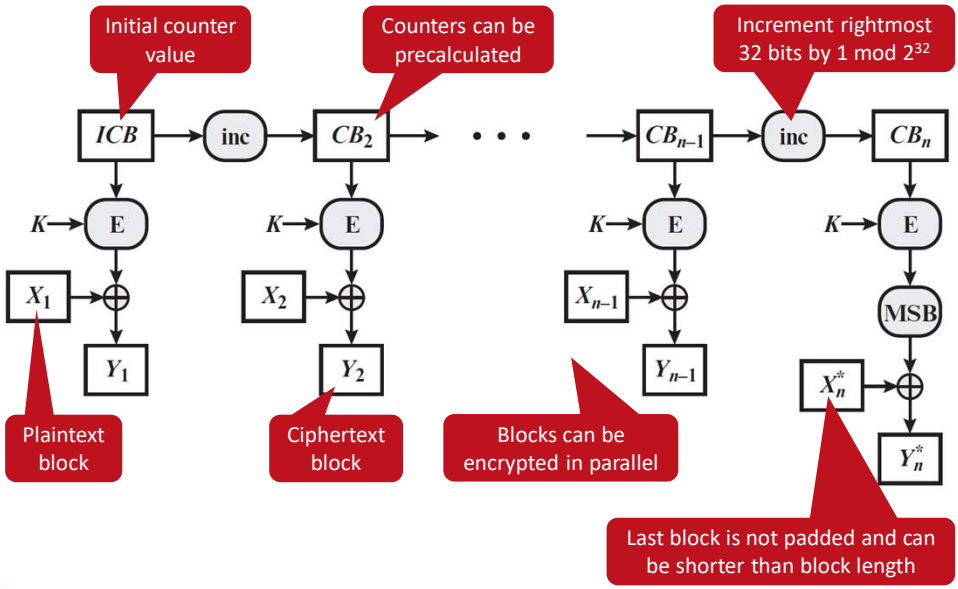
Galois/Counter Mode (GCM)

- Variation of the encrypt-then-MAC approach (E → A)
- Standardized by NIST in NIST SP 800-38D
- Designed for **efficiency**: high throughput with low cost and low latency
- Uses a variant of **CTR mode** that includes an authenticator tag (MAC)
  - GHASH: A keyed hash function to generate MAC
  - GCTR: CTR mode with counters determined by increment
- Authentication-only mode is also available, known as **GMAC**

The GCM mode of operation, standardized by NIST in NIST SP 800-38D, is designed to be parallelizable so that it can provide high throughput with low cost and low latency. In essence, the message is encrypted in variant of CTR mode. The resulting ciphertext is multiplied with key material and message length information over GF(2^{128}) to generate the authenticator tag. The standard also specifies a mode of operation that supplies the MAC only, known as GMAC.

The GCM mode makes use of two functions: GHASH, which is a keyed hash function, and GCTR, which is essentially the CTR mode with the counters determined by a simple increment by one operation.

GCM – GCTR



University of Antwerp | Faculty of Science

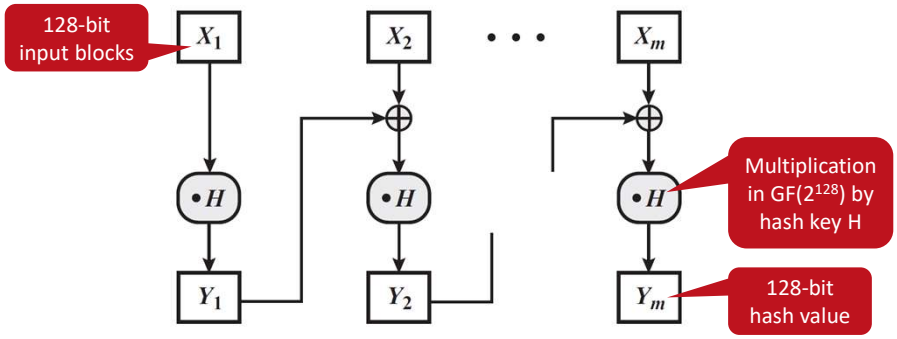
69

$GCTR_K(ICB, X)$ takes as input a secret key K and a bit string X of arbitrary length and returns a ciphertext Y of bit length (X). The function may be specified as follows:

1. If X is the empty string, then return the empty string as Y.
2. Let n = ceil(len(X)/128). That is, n is the smallest integer greater than or equal to (X)/128.
3. Let $X_1, X_2, ..., X_{n-1}, X_n^*$ denote the unique sequence of bit strings such that $X = X_1 \,||\, X_2 \,||\, ... \,||\, X_{n-1} \,||\, X_n^*$ ($X_1, X_2, ..., X_{n-1}$ are complete 128-bit blocks)
4. Let $CB_1 = ICB$.
5. For, i = 2 to n let $CB_i = inc_{32}(CB_{i-1})$, where the $inc_{32}(S)$ function increments the rightmost 32 bits of S by 1 mod 2^{32}, and the remaining bits are unchanged.
6. For i = 1 to n - 1, do $Y_i = X_i \oplus E(K, CB_i)$.
7. Let $Y_n^* = X_n^* \oplus MSB_{len(Xn^*)}(E(K, CB_n))$.
8. Let $Y = Y_1 \,||\, Y_2 \,||\, ... \,||\, Y_{n-1} \,||\, Y_n^*$
9. Return Y.

Note that the counter values can be quickly generated and that the encryption operations can be performed in parallel.

GCM – GHASH



$$GHASH_H(X) = (X_1 \cdot H^m) \oplus (X_2 \cdot H^{m-1}) \oplus \cdots \oplus (X_{m-1} \cdot H^2) \oplus (X_m \cdot H)$$

University of Antwerp | Faculty of Science

$GHASH_H(X)$ takes as input the hash key H and a bit string X such that len(X) = 128m bits for some positive integer m and produces a 128-bit MAC value. The function may be specified as follows:

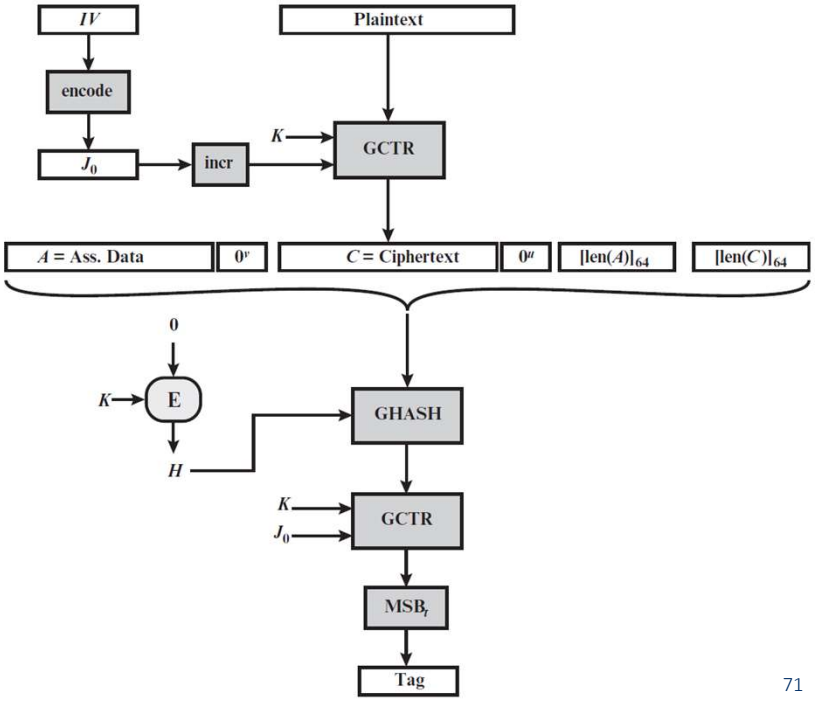
1. Let $X_1, X_2, \ldots, X_{m-1}, X_m$ denote the unique sequence of blocks such that
2. $X = X_1 \,||\, X_2 \,||\, \ldots \,||\, X_{m-1} \,||\, X_m$.
3. Let Y_0 be a block of 128 zeros, designated as 0^{128}.
4. For $i = 1, \ldots, m$, let $Y_i = (Y_{i-1} \oplus X_i) \cdot H$, where $\cdot$ designates multiplication in GF(2^{128}).
5. Return Y_m.

The GHASH(X) function can be expressed as

$$(X_1 \cdot H^m) \oplus (X_2 \cdot H^{m-1}) \oplus \cdots \oplus (X_{m-1} \cdot H^2) \oplus (X_m \cdot H)$$

This formulation has desirable performance implications. If the same hash key is to be used to authenticate multiple messages, then the values H^2, H^3, ... can be precalculated one time for use with each message to be authenticated. Then, the blocks of the data to be authenticated ($X_1, X_2, \ldots, X_m$) can be processed in parallel, because the computations are independent of one another.

Overview of GCM mode



University of Antwerp
Faculty of Science

71

We can now define the overall authenticated encryption function. The input consists of a secret key K, an initialization vector IV, a plaintext P, and additional authenticated data A. The notation $[x]_s$ means the s-bit binary representation of the nonnegative integer x. The steps are as follows.

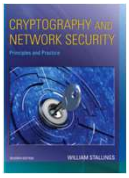
1. Let $H = E(K, 0^{128})$.
2. Define a block, J_0, as
   If len(IV) = 96, then let $J_0 = IV \,||\, 0^{31} \,||\, 1$.
   If len (IV) ≠ 96, then let s = 128 * ceil(len(IV)/128) - len(IV), and let $J_0 = \text{GHASH}_H(IV \,||\, 0^{s+64} \,||\, [\text{len}(IV)]_{64})$.
3. Let $C = \text{GCTR}_K(\text{inc}_{32}(J_0), P)$.
4. Let u = 128 * ceil(len(C)/128) - len(C) and let v = 128 * ceil(len(A)/128) - len(A).
5. Define a block, S, as $S = \text{GHASH}_H(A \,||\, 0^v \,||\, C \,||\, 0^u \,||\, [\text{len}(A)]_{64} \,||\, [\text{len}(C)]_{64})$
6. Let $T = MSB_t(GCTR_K(J_0, S))$, where t is the supported tag length.
7. Return (C, T).

In step 1, the hash key is generated by encrypting a block of all zeros with the secret key K. In step 2, the pre-counter block (J_0) is generated from the IV. In particular, when the length of the IV is 96 bits, then the padding string $0^{31} \,||\, 1$ is appended to the IV to form the pre-counter block. Otherwise, the IV is padded with the minimum number of 0 bits, possibly none, so that the length of the resulting string is a multiple of 128 bits (the block size); this string in turn is appended with 64 additional 0 bits, followed by the 64-bit representation of the length of the IV, and the GHASH function is applied to the resulting string to form the pre-counter block.

Thus, GCM is based on the CTR mode of operation and adds a MAC that authenticates both the message and additional data that requires only authentication. The function that computes the hash uses only multiplication in a Galois field. This choice was made because the operation of multiplication is easy to perform within a Galois field and is easily implemented in hardware.

Summary of message authentication

- Ensures integrity of transmitted data
- MACs combine a checksum, with the use of a secret key
  - Merkle-Damgård MAC of the form H(S|M) is vulnerable to length extension attack
- CMAC turns 3DES or AES into a MAC
- HMAC uses hash functions to provide a key-based cryptographic checksum
  - Generally more efficient than CMAC
- Authenticated encryption ensures confidentiality and integrity
  - CCM performs encrypt-and-MAC using AES, CTR mode, and CMAC
  - GCM performs encrypt-then-MAC using GHASH and GCTR mode



Link with the book

- Chapter 12 (Sections 12.1, 12.4—12.7)

Link with the videos

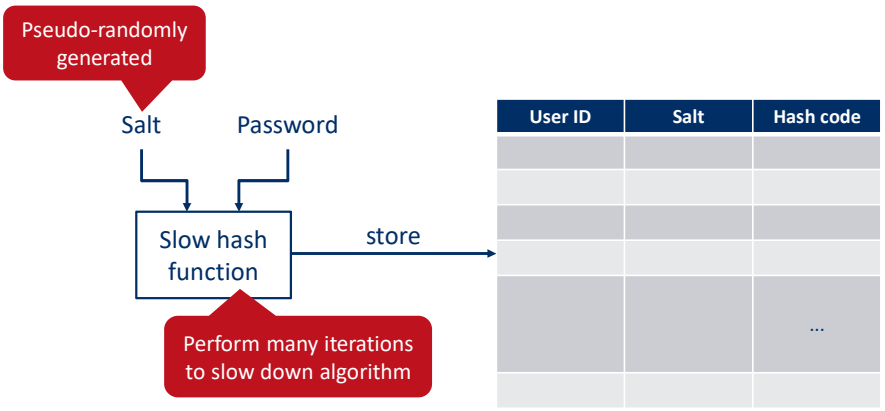
- Week 3 (Message Integrity 1 & 2)
- Week 3 (HMAC)
- Week 4 (Authenticated Encryption 1 & 2)

Secure Password Storage

University of Antwerp
Faculty of Science

Password-based authentication

- Users log into a system using a combination of username (ID) and password
  - ID provides a unique identifier of the users for access control purposes
  - Password provides a way to authenticate the user



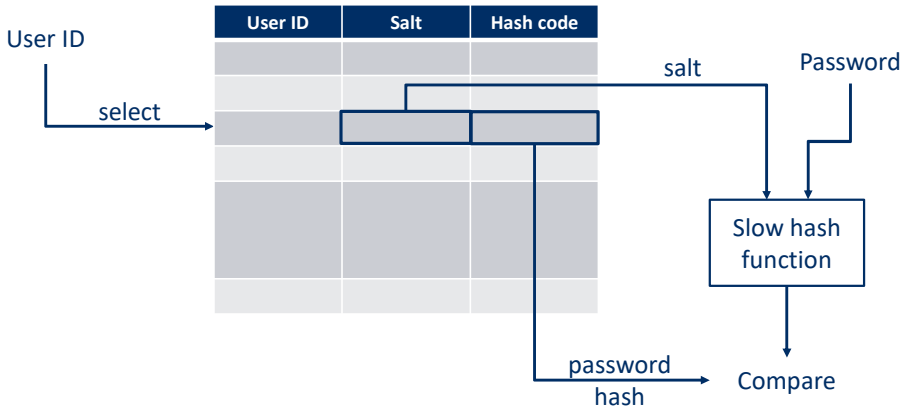
University of Antwerp | Faculty of Science

74

The front line of defense against intruders is the password system. Virtually all multiuser systems require that a user provide not only a name or identifier (ID) but also a password. The password serves to authenticate the ID of the individual logging on to the system.

A widely used password security technique is the use of hashed passwords and a salt value. To store a new password onto the system, the user selects or is assigned a password. This password is combined with a fixed-length salt value. Modern implementations use a pseudorandom or random number. The password and salt serve as inputs to a hashing algorithm to produce a fixed-length hash code. The hash algorithm is designed to be slow to execute to thwart brute force or password guessing attacks. The hashed password is then stored, together with the plaintext copy of the salt, in the password file of the corresponding user ID.

Password verification



University of Antwerp
Faculty of Science

When a user attempts to log into the system, the user provides an ID and a password. The password management system uses the ID to index into the password file or database and retrieve the plaintext salt and the hashed password. The salt and user-supplied password are used as input to the hash function routine. If the result matches the stored value, the password is accepted.

Purpose of a salt value

1. Prevents duplicate passwords from being visible in the password file
2. It greatly increases the difficulty of offline dictionary attacks
3. It becomes nearly impossible to find out if a person uses the same password on multiple systems

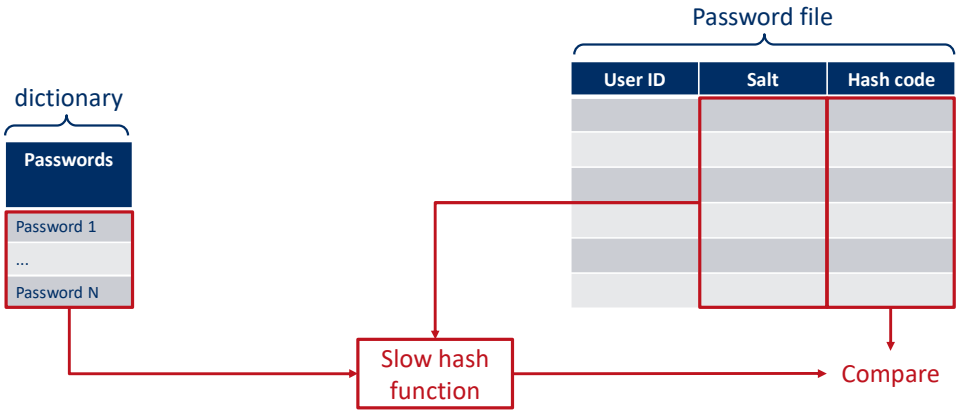
The salt serves three purposes:

1. It prevents duplicate passwords from being visible in the password file or database. Even if two users choose the same password, those passwords will be assigned different salt values. Hence, the hashed passwords of the two users will differ.
2. It greatly increases the difficulty of offline dictionary attacks. For a salt of length b bits, the number of possible passwords is increased by a factor or 2^b, increasing the difficulty of guessing a password in a dictionary attack.
3. It becomes nearly impossible to find out whether a person with passwords on two or more systems has used the same password on all of them.

To see the second point, consider the way that an offline dictionary attack would work. The attacker obtains a copy of the password file. Suppose first that the salt is not used. The attacker's goal is to guess a single password. To that end, the attacker submits a large number of likely passwords to the hashing function. If any of the guesses matches one of the hashes in the file, then the attacker has found a password that is in the file. But faced with salted hashes, the attacker must take each guess and submit it to the hash function once for each salt value in the dictionary file, multiplying the number of guesses that must be checked.

Password cracking: Dictionary attack

- Develop a large dictionary of possible passwords offline
- Try each dictionary entry hash against the password file
- Each possible password must be hashed with each available salt value



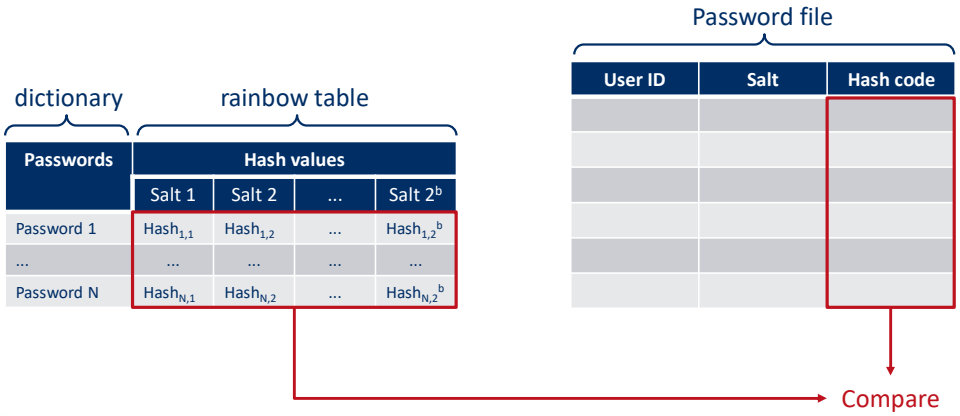
University of Antwerp
Faculty of Science

77

The traditional approach to password guessing, or password cracking as it is called, is to develop a large dictionary of possible passwords and try each of these against the password file. The means that each password must be hashed using each available salt value and then compared to the stored hash values. If no match is found, then the cracking program tries variations on all the words in its dictionary of likely passwords. Such variations include backward spelling of words, additional numbers of special characters, or sequence of characters.

Password cracking: Rainbow table attack

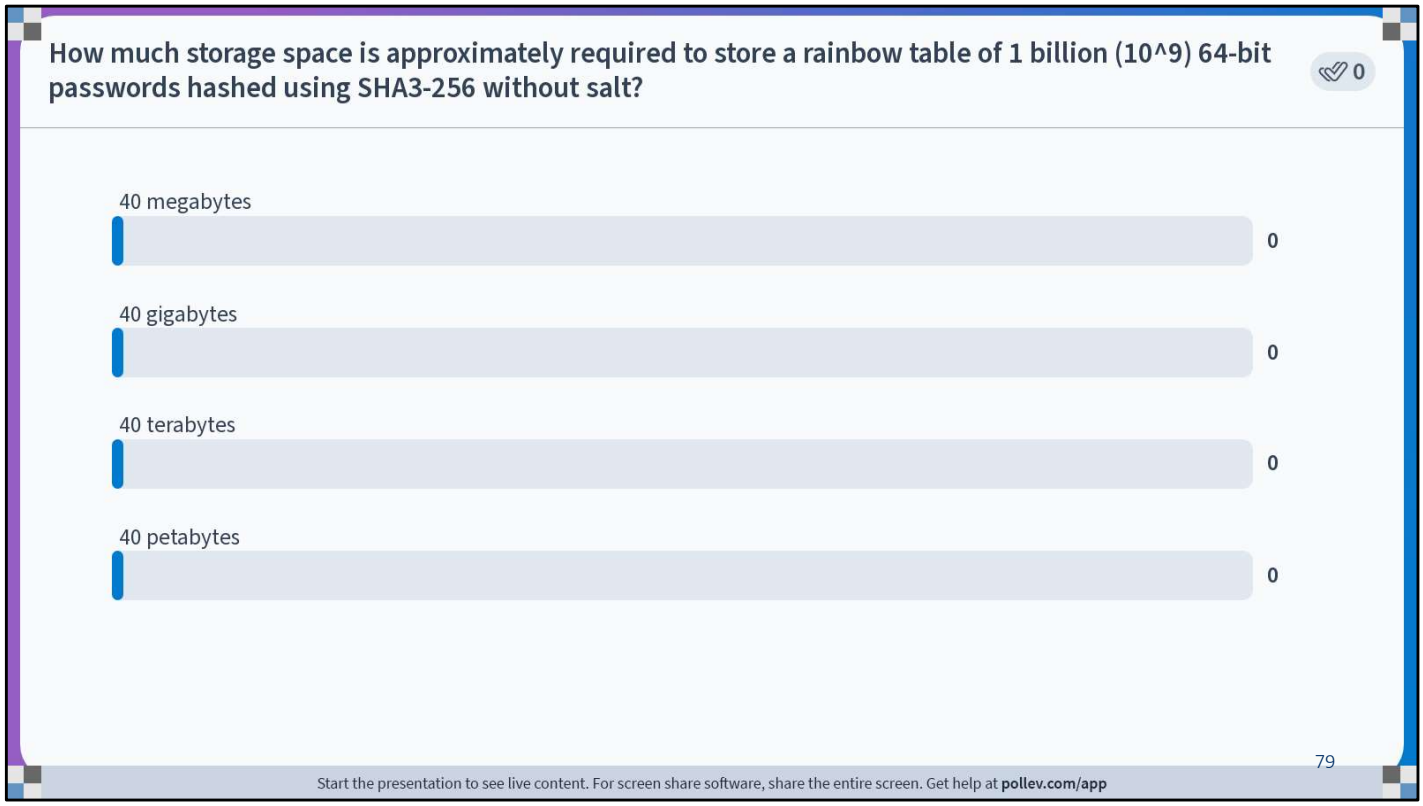
- Trade-off time for space by precomputing potential hash values
- Hash values of each possible password with each possible salt are pre-calculated and stored in a file



University of Antwerp
Faculty of Science

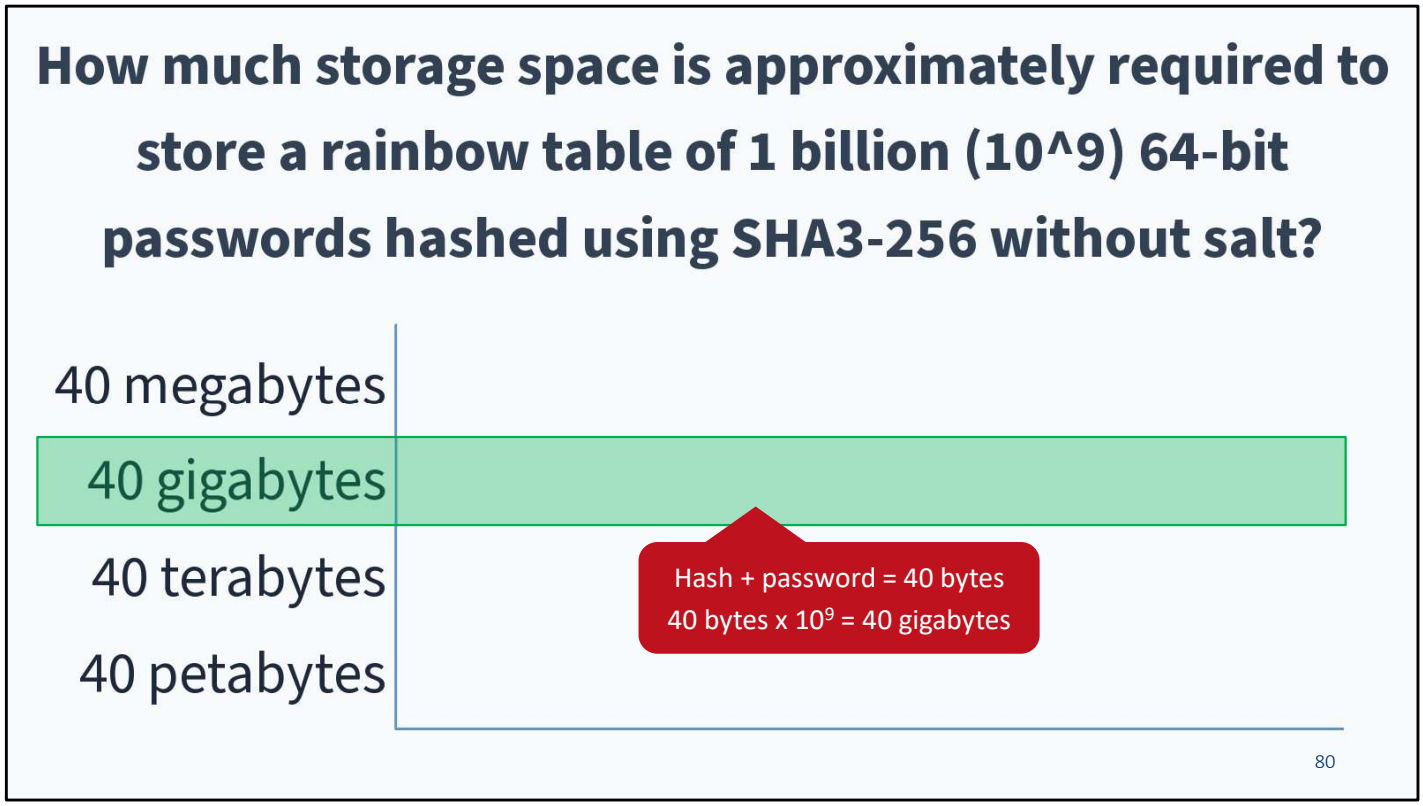
78

An alternative is to trade off space for time by precomputing potential hash values. In this approach the attacker generates a large dictionary of possible passwords. For each password, the attacker generates the hash values associated with each possible salt value. The result is a mammoth table of hash values known as a rainbow table.

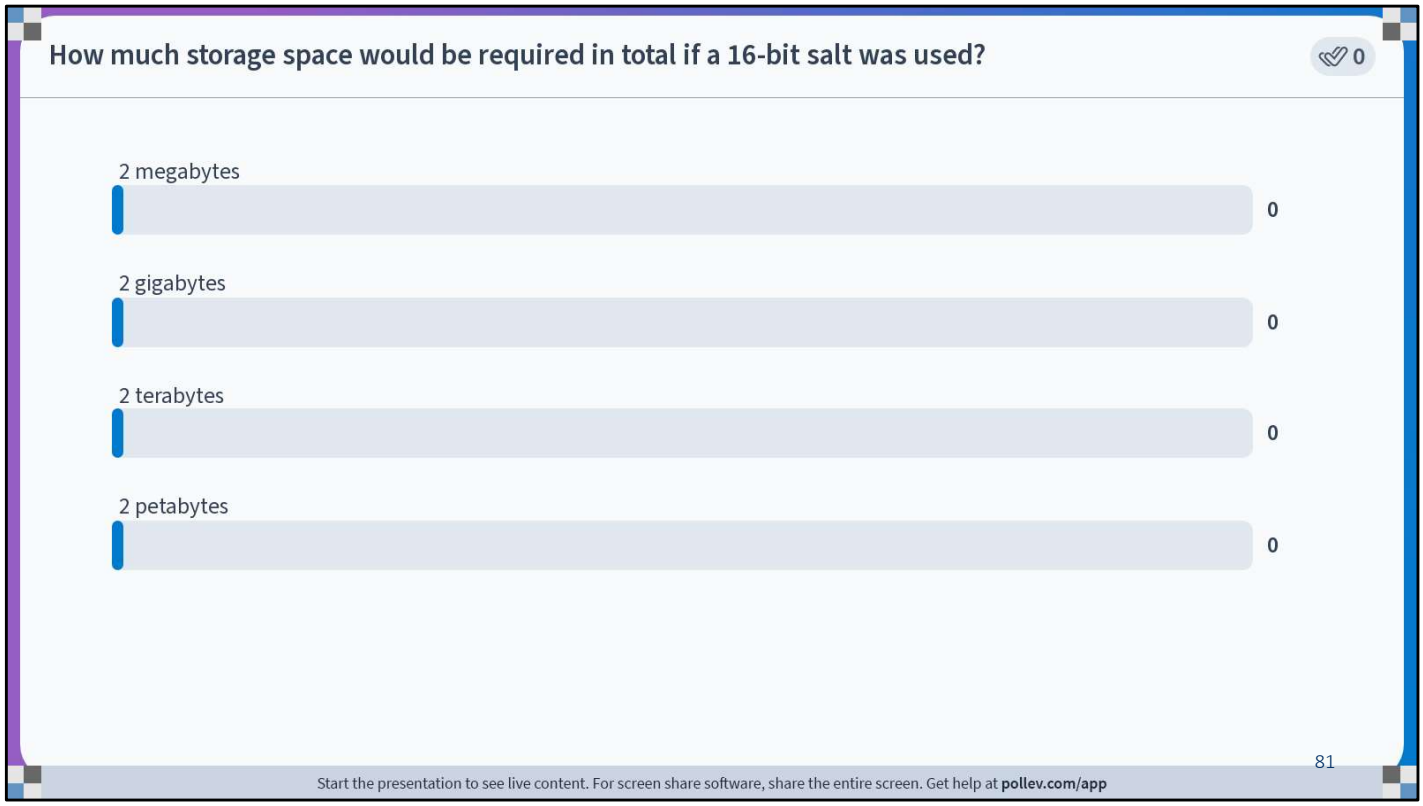


Poll Title: Do not modify the notes in this section to avoid tampering with the Poll Everywhere activity.
More info at polleverywhere.com/support

How much storage space is approximately required to store a rainbow table of 1 billion (10^9) 64-bit passwords hashed using SHA3-256 without salt?
https://www.polleverywhere.com/multiple_choice_polls/QtQGLTzqHmMtRRLQCRYTL

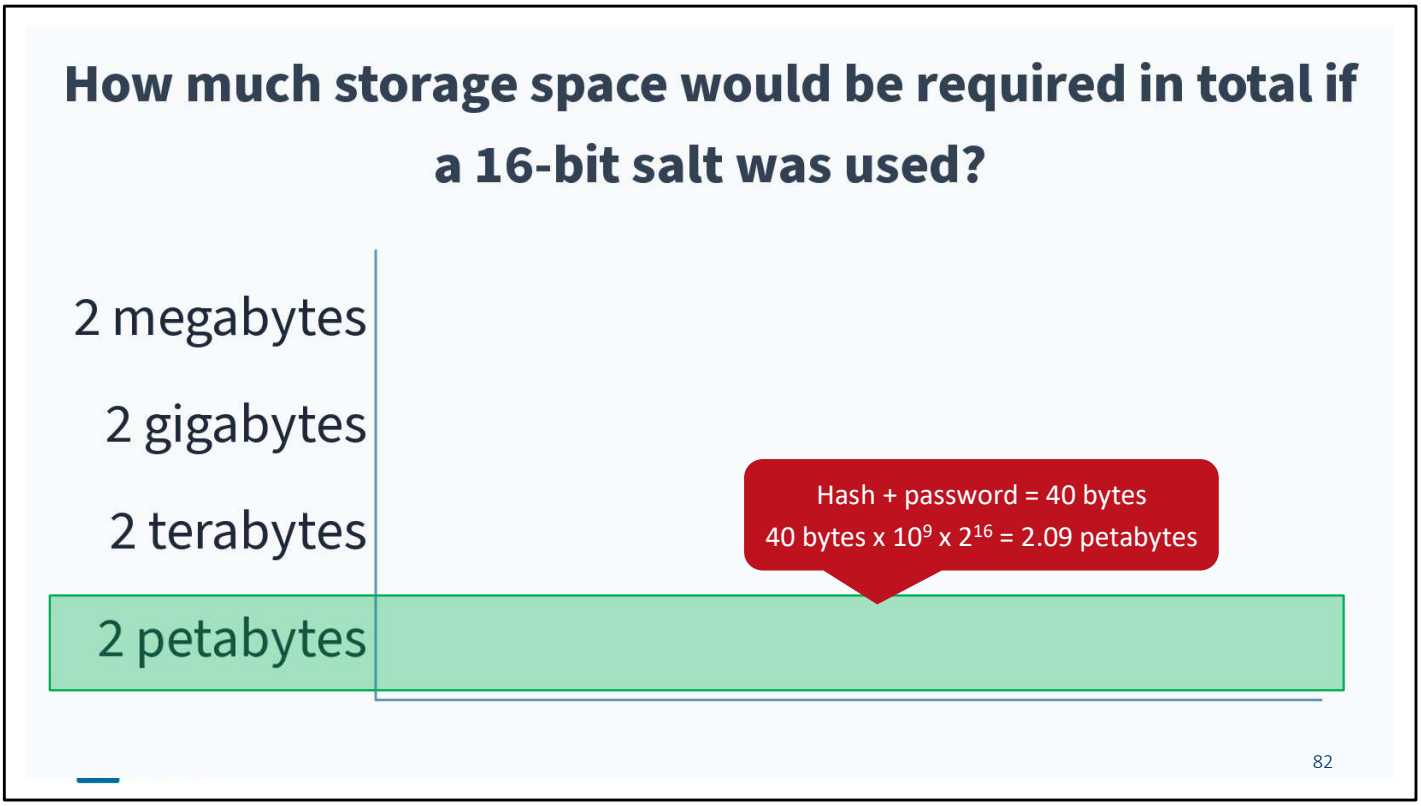


Poll Title: How much storage space is approximately required to store a rainbow table of 1 billion (10^9) 64-bit passwords hashed using SHA3-256 without salt?



Poll Title: Do not modify the notes in this section to avoid tampering with the Poll Everywhere activity.
More info at polleverywhere.com/support

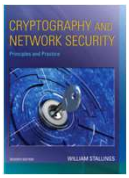
How much storage space would be required in total if a 16-bit salt was used?
https://www.polleverywhere.com/multiple_choice_polls/l8Fe7Nt0Th5snd1P1I95E



Poll Title: How much storage space would be required in total if a 16-bit salt was used?

Summary of secure password storage

- User ID provides unique identification and access control
- Password provides authentication of the user
- Stored password as hashed with a salt to keep them secret
  - A slow hash function is used to slow down brute force password guessing attacks
  - Salt is used to increase difficulty of offline guessing attacks
- Two main types of attacks
  - Dictionary attack: List of common passwords that are tested online
  - Rainbow table attack: Salted hashes are pre-calculated to trade off time for storage space



Link with the book (online chapters)

- Chapter 22 (Section 22.3)

End of Part 3

University of Antwerp
Faculty of Science