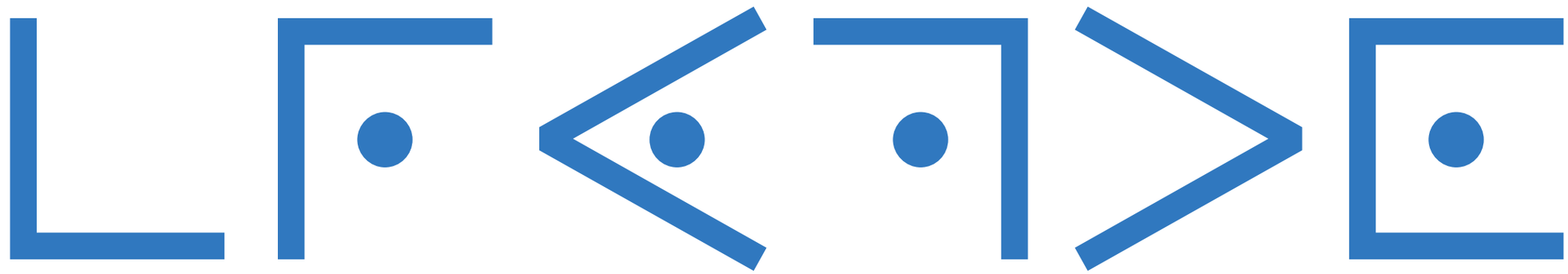


contrôle d'intégrité

Nicolas Ollinger

M1 informatique — 2024/2025

La semaine dernière

Un protocole naïf

Alice et Bob ont échangé une clé secrète de 256 bits et élu l'algorithme AES-256-CBC. Ils souhaitent établir un canal de communication sécurisé entre eux, sur une socket TCP, comme suit. Eve écoute les échanges « sécurisés ».

Pour transmettre un message M, on applique d'abord le padding, puis on choisit un IV aléatoire, on chiffre M avec AES-CBC et on transmet le résultat C préfixé par l'IV. À la réception, les messages sont acquittés par ACK ou NACK.

Identifier les **nombreux** problèmes de ce protocole. Expliquer comment Eve peut déchiffrer facilement les messages transmis !

Security Flaws Induced by CBC Padding Applications to SSL, IPSEC, WTLS...

Serge Vaudenay

Swiss Federal Institute of Technology (EPFL)
Serge.Vaudenay@epfl.ch

Abstract. In many standards, e.g. SSL/TLS, IPSEC, WTLS, messages are first pre-formatted, then encrypted in CBC mode with a block cipher. Decryption needs to check if the format is valid. Validity of the format is easily leaked from communication protocols in a chosen ciphertext attack since the receiver usually sends an acknowledgment or an error message. This is a side channel.
In this paper we show various ways to perform an efficient side channel attack. We discuss potential applications, extensions to other padding schemes and various ways to fix the problem.

Padding oracle attack

```
def alice(msg):
  M=pad(msg)
  IV=genIV()
  C=AES-256-CBC(K,IV,M)
  send(IV·C)
```

```
def bob(msg):
  IV·C=msg
  try:
    MP=AES-256-CBC⁻¹(K,IV,C)
    M=unpad(MP)
    send(ACK)
    return M
  except:
    send(NACK)
```

Padding **oracle** attack

Eve peut déchiffrer facilement les messages transmis en utilisant Bob comme Oracle !

```
def oracle(msg):
  sendto(bob,msg)
  return recvfrom(bob) == ACK
```

En supposant que msg a la bonne taille, l'oracle répond vrai si et seulement si $\texttt{AES-256-CBC}^{-1}\texttt{(K,IV,C)}$ a un padding valide.

Si C est composé d'un unique bloc : $\texttt{oracle(IV·C)}$ est vrai si et seulement si $\texttt{IV}\oplus\texttt{AES-256-CBC}_{\texttt{K}}^{-1}\texttt{(K,IV,C)}$ termine par $\texttt{kkk···k}$ (k fois).

Padding **oracle** attack

Si C est composé d'un unique bloc : $\texttt{oracle(IV·C)}$ est vrai si et seulement si $\texttt{IV}\oplus\texttt{AES-256-CBC}_K^{-1}\texttt{(K,IV,C)}$ termine par $\texttt{kkk}\cdots\texttt{k}$ (k fois).

L'astuce consiste à calculer $\texttt{oracle((IV}\oplus\Delta\texttt{)·C)}$ en faisant varier Δ pour déchiffrer $\texttt{M}$ octet par octet, en faisant parcourir au padding les valeurs successives 1, 22, 333, 4444, 55555, ...

Astuce : remarquer que lorsque Δ parcourt les 256 valeurs de la forme $\texttt{0000...00x}$, l'oracle retourne vrai au plus 2 fois.

Protocole cryptographique

Un protocole cryptographique est construit à partir de briques, les primitives cryptographiques, dans le but d'assurer un certain nombre de propriétés, typiquement :

- confidentialité ;
- intégrité ;
- authenticité ;
- non répudiabilité.

Confidentialité

- Assurer que seules les deux parties ont accès aux données échangées.

- Empêche l'**écoute** des données en transit.

- Selon le contexte cette confidentialité peut être **persistante** dans le temps.

Intégrité

- Assurer la correction et la consistance des données transmises.
- Empêche de **modifier** les données en transit.
- Empêche de **forger** des nouvelles données.
- Selon le contexte ce contrôle d'intégrité peut se faire avec ou sans **répudiabilité**.

Authenticité

- Permettre aux deux parties en présence de **valider l'identité** de l'autre partie.

- Empêche les accès non autorisés mais aussi...

- Empêche les attaques de type **homme du milieu**.

- Affaibli parfois dans le cadre client/serveur par une authentification du serveur uniquement.

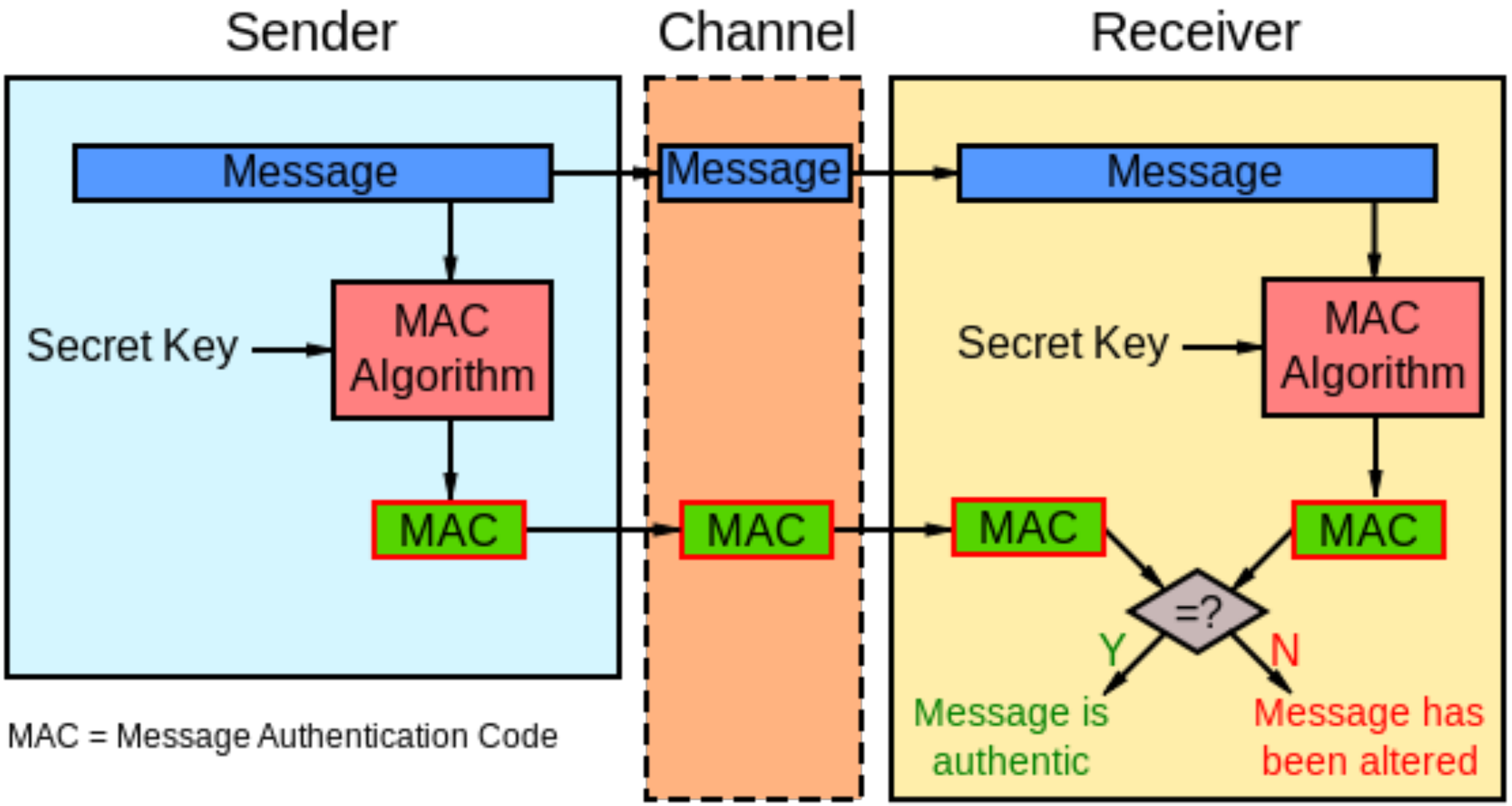
Attention à la notion d'identité (IP, DNS, ... ?)

Quelques primitives cryptographiques

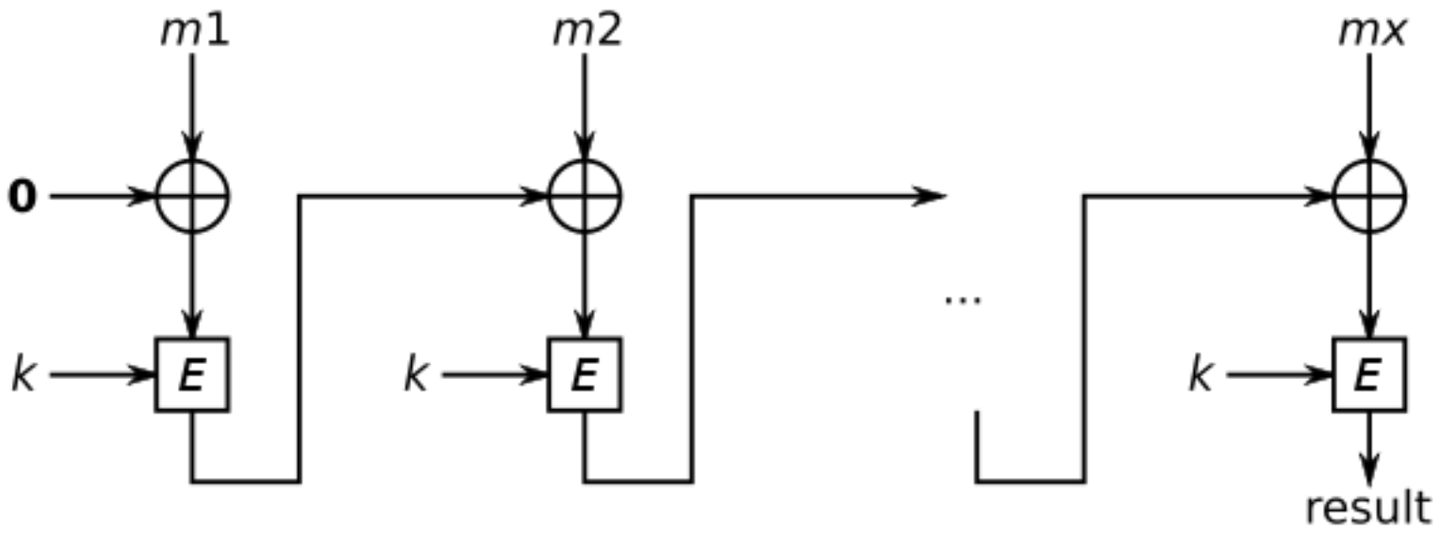
- **Cryptographie symétrique** (à clé secrète) ;
- Cryptographie asymétrique (à clé publique) ;
- Générateurs de nombres pseudo-aléatoires de qualité cryptographique ;
- Fonctions de hachage de qualité cryptographique ;
- *Codes d'authentification de messages (MAC)* ;
- Algorithmes de signature numérique.

Codes MAC

Code d’authentification MAC



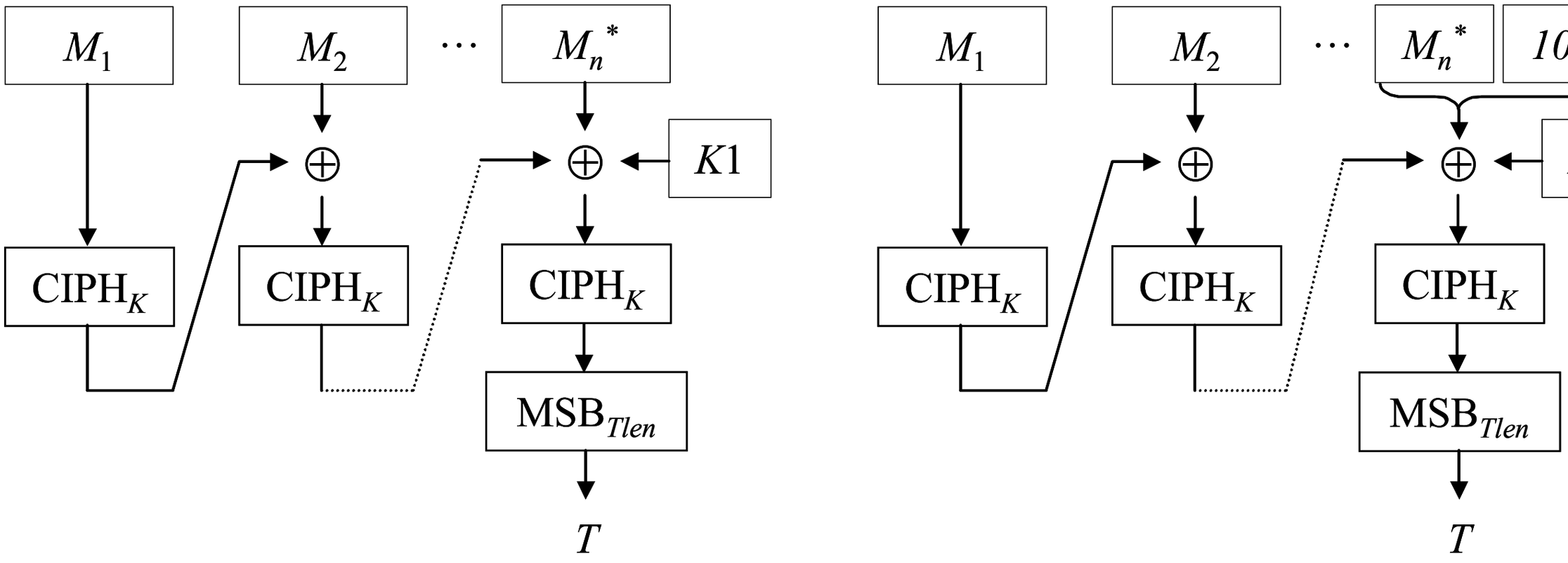
Exemple : CBC-MAC



Utiliser le dernier bloc d’un chiffrement CBC (avec une autre clé !!!)

Possibilité de forger des MAC valides sur des messages de taille variable : CBC-MAC n'est pas sûr pour des messages de taille variable.

CMAC (**NIST 800-38B**)



Amélioration de CBC-MAC.

Pause exercice

Alice a la brillante idée de combiner AES-256-CBC avec AES-CBC-MAC. Munie d'une clé secrète K de 256 bits, elle chiffre un message M comme suit :

$$E_K(M) \,\|\, H_K(M)$$

Quelle est l'énorme faille dans sa proposition ?

Et en remplaçant AES-CBC-MAC par AES-CMAC ?

Exercice à emporter

Protocole pour un canal sécurisé

Alice et Bob ont échangé des clés secrètes de 256 bits et élu les algorithmes **AES-256-CTR** et **AES-CMAC**. Ils souhaitent établir un canal de communication sécurisé entre eux, sur une socket TCP. Eve écoute les échanges « sécurisés ».

Proposer un protocole cryptographique qui permet d’assurer la confidentialité et l’intégrité des échanges.

Hachage

Fonction de hachage

$$H : \{0,1\}^* \rightarrow \{0,1\}^t$$

Fonction de compression qui associe une **empreinte** de taille fixe à un message de taille arbitraire.

Application : table de hachage

Propriété : les valeurs prises par *H* doivent être **uniformément réparties** ; faible probabilité de **collision** *H*(x)=*H*(y).

Hachage cryptographique

Un ingrédient essentiel dans les protocoles modernes : permet d'agencer des primitives (par ex. on signe l'empreinte des messages).

Propriétés recherchées pour H :

1. **one-way** : difficile de retrouver x à partir de $H(x)$.
2. **résistance aux collisions** : difficile de trouver x et y tels que $H(x) = H(y)$.
3. indiscernable d'une fonction aléatoire.

Construction de Merkle–Damgård

Transforme une fonction de compression one-way résistante aux collisions en une fonction de hachage de qualité cryptographique.

$$m = m_1 \| m_2 \| \cdots \| m_k$$

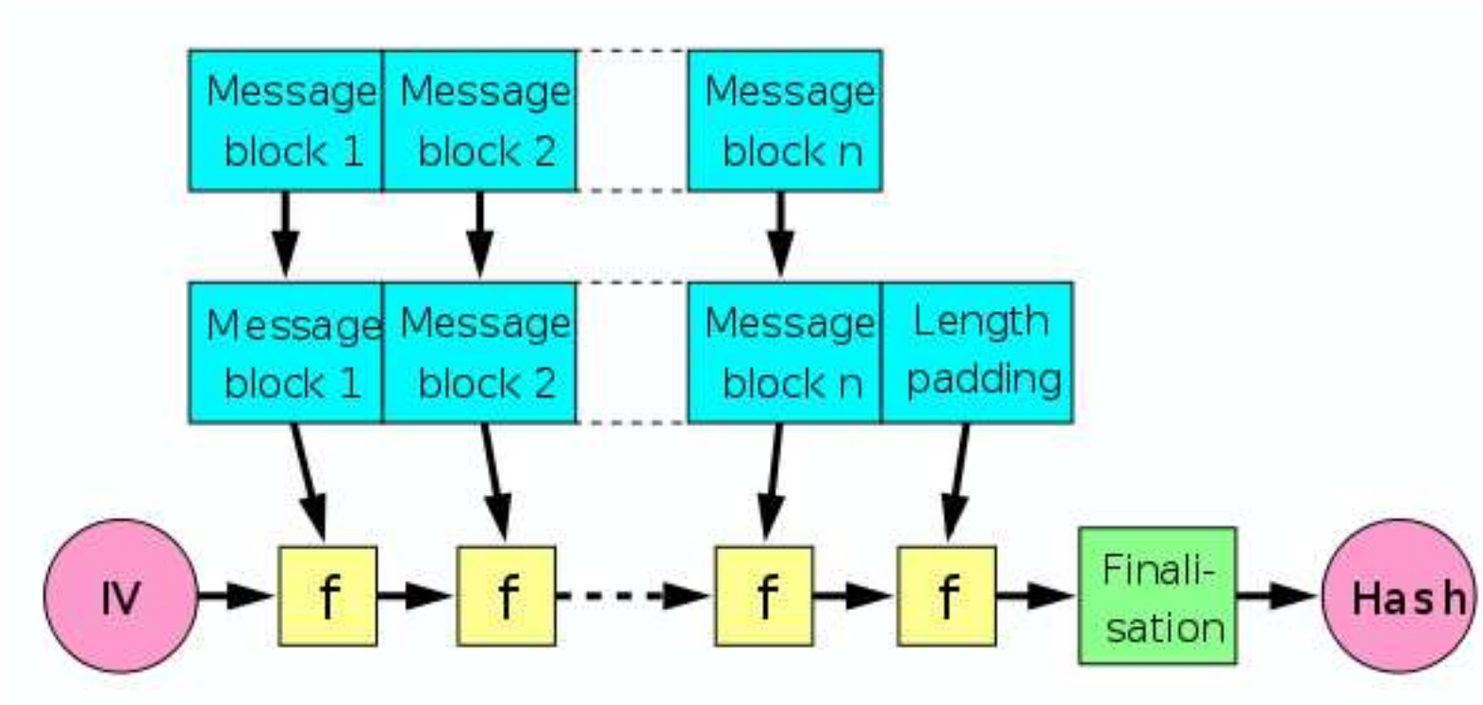
$$H_0 = \mathrm{IV}$$

$$H_i = F(H_{i-1}, m_i)$$

$$H(m) = G(H_k)$$

$$F : \{0,1\}^s \times \{0,1\}^b \to \{0,1\}^s$$

$$G : \{0,1\}^s \to \{0,1\}^t$$



Pause exercice

Charlie trouve que la construction MD ressemble beaucoup au chaînage CBC de la semaine dernière. Il propose de prendre IV=0 et comme fonction de compression le chaînage d'AES-256 avec la clé 0.

Montrer que cette fonction n'est pas du tout résistante aux collisions ! Expliquez comment construire une deuxième préimage pour tout message de deux blocs.

Padding MD-compatible

Conditions suffisantes sur le padding pour que les collisions MD correspondent à des collisions de la fonction de compression :

x est un préfixe de Pad(x).

si $|x| = |y|$ alors $|\text{Pad}(x)| = |\text{Pad}(y)|$

sinon les derniers blocs de Pad(x) et Pad(y) différent.

MD5 (**RFC 1321**)

Inventé par Rivest en 1992

Très populaire malgré une première faille identifiée dès 1995.

Empreinte de 128 bits (blocs de 512 bits)

Conçu pour être rapide sur archi 32 bits.

Padding : on ajoute un bit à 1 puis des 0 pour obtenir une taille congrue à 448 mod 512 et enfin on ajoute la longueur initiale du message codée sur 64 bits.

```
A,B=0x01234567,0x89abcdef
C,D=0xfedcba98,0x76543210
Pour i de 0 à 63:
  T[i] = int(4294967296*abs(sin(i)))

Pour i de 0 à N/16-1:
Pour j de 0 à 15: X[j]=M[i*16+j]
AA,BB,CC,DD=A,B,C,D
Round1()
Round2()
Round3()
Round4()
A,B,C,D=A+AA,B+BB,C+CC,D+DD
```

```
def Round1():
  F(X,Y,Z) = XY v not(X) Z
  – Let [abcd k s i] denote the operation
  – a=b+((a+F(b,c,d)+X[k]+T[i]) <<< s).
   [ABCD  0  7  1]  [DABC  1 12  2]
   [CDAB  2 17  3]  [BCDA  3 22  4]
   [ABCD  4  7  5]  [DABC  5 12  6]
   [CDAB  6 17  7]  [BCDA  7 22  8]
   [ABCD  8  7  9]  [DABC  9 12 10]
   [CDAB 10 17 11]  [BCDA 11 22 12]
   [ABCD 12  7 13]  [DABC 13 12 14]
   [CDAB 14 17 15]  [BCDA 15 22 16]
```

```
def Round2():
  G(X,Y,Z) = XZ v Y not(Z)
  — Let [abcd k s i] denote the operation
  — a=b+((a+G(b,c,d)+X[k]+T[i]) <<< s).
   [ABCD  1  5 17]  [DABC  6  9 18]
   [CDAB 11 14 19]  [BCDA  0 20 20]
   [ABCD  5  5 21]  [DABC 10  9 22]
   [CDAB 15 14 23]  [BCDA  4 20 24]
   [ABCD  9  5 25]  [DABC 14  9 26]
   [CDAB  3 14 27]  [BCDA  8 20 28]
   [ABCD 13  5 29]  [DABC  2  9 30]
   [CDAB  7 14 31]  [BCDA 12 20 32]
```

```
def Round3():
  H(X,Y,Z) = X xor Y xor Z
  — Let [abcd k s i] denote the operation
  — a=b+((a+H(b,c,d)+X[k]+T[i]) <<< s).
   [ABCD  5  4 33]  [DABC  8 11 34]
   [CDAB 11 16 35]  [BCDA 14 23 36]
   [ABCD  1  4 37]  [DABC  4 11 38]
   [CDAB  7 16 39]  [BCDA 10 23 40]
   [ABCD 13  4 41]  [DABC  0 11 42]
   [CDAB  3 16 43]  [BCDA  6 23 44]
   [ABCD  9  4 45]  [DABC 12 11 46]
   [CDAB 15 16 47]  [BCDA  2 23 48]
```

```
def Round4():
  I(X,Y,Z) = Y xor (X v not(Z))
  - Let [abcd k s i] denote the operation
  - a=b+((a+I(b,c,d)+X[k]+T[i]) <<< s).
  [ABCD  0  6 49]  [DABC  7 10 50]
  [CDAB 14 15 51]  [BCDA  5 21 52]
  [ABCD 12  6 53]  [DABC  3 10 54]
  [CDAB 10 15 55]  [BCDA  1 21 56]
  [ABCD  8  6 57]  [DABC 15 10 58]
  [CDAB  6 15 59]  [BCDA 13 21 60]
  [ABCD  4  6 61]  [DABC 11 10 62]
  [CDAB  2 15 63]  [BCDA  9 21 64]
```

Collisions MD5 (bloc unique)

```
>>> from hashlib import md5
>>> def calc(s): print md5(''.join(s.split()).decode('hex')).hexdigest()
...
>>> calc('''
... 4dc968ff0ee35c209572d4777b721587
... d36fa7b21bdc56b74a3dc0783e7b9518
... afbfa200a8284bf36e8e4b55b35f4275
... 93d849676da0d155d8360fb5f07fea2
... ''')
008ee33a9d58b51cfeb425b0959121c9
>>>
>>> calc('''
... 4dc968ff0ee35c209572d4777b721587
... d36fa7b21bdc56b74a3dc0783e7b9518
... afbfa202a8284bf36e8e4b55b35f4275
... 93d849676da0d1d55d8360fb5f07fea2
... ''')
008ee33a9d58b51cfeb425b0959121c9
```

Collisions MD5 (préfixe choisi)

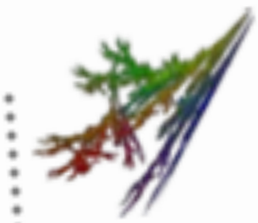
```
$ md5 *.py
MD5 (md5a.py) = c8291765efccefa50638084c8670027b
MD5 (md5b.py) = c8291765efccefa50638084c8670027b
```

```
$ ./md5a.py
This is a triumph.
I'm making a note here:
HUGE SUCCESS.
It's hard to overstate
my satisfaction.
Aperture Science
We do what we must
because we can.
For the good of all of us.
Except the ones who are dead.
```

```
$ ./md5b.py
The cake is a lie.
The cake is a lie.
The cake is a lie.
The cake is a lie.
The cake is a lie.
The cake is a lie.
```

Collisions MD5 (préfixes distincts)

```
$ md5 plane.jpg ship.jpg
MD5 (plane.jpg) = 253dd04e87492e4fc3471de5e776bc3d
MD5 (ship.jpg) = 253dd04e87492e4fc3471de5e776bc3d
```



MD5 Considered Harmful Today

Creating a rogue CA certificate

Alexander Sotirov — *New York, USA*

Marc Stevens — *CWI, Netherlands*

Jacob Appelbaum — *Noisebridge/Tor, SF*

Arjen Lenstra — *EPFL, Switzerland*

David Molnar — *UC Berkeley, USA*

Dag Arne Osvik — *EPFL, Switzerland*

Benne de Weger — *TU/e, Netherlands*

FIPS PUB 180-4

FEDERAL INFORMATION PROCESSING STANDARDS PUBLICATION

Secure Hash Standard (SHS)

CATEGORY: COMPUTER SECURITY SUBCATEGORY: CRYPTOGRAPHY

Information Technology Laboratory
National Institute of Standards and Technology
Gaithersburg, MD 20899-8900

This publication is available free of charge from:
http://dx.doi.org/10.6028/NIST.FIPS.180-4

August 2015

SHA-1 et SHA-2

Développé par la NSA vers 1995

Empreinte de 160 bits pour SHA-1

SHA-2 plus sûr mais moins rapide à calculer

Algorithm	Message Size (bits)	Block Size (bits)	Word Size (bits)	Message Digest Size (bits)
SHA-1	$< 2^{64}$	512	32	160
SHA-224	$< 2^{64}$	512	32	224
SHA-256	$< 2^{64}$	512	32	256
SHA-384	$< 2^{128}$	1024	64	384
SHA-512	$< 2^{128}$	1024	64	512
SHA-512/224	$< 2^{128}$	1024	64	224
SHA-512/256	$< 2^{128}$	1024	64	256

Figure 1: Secure Hash Algorithm Properties

EUROCRYPT 2016

Freestart collision for full SHA-1✧

Marc Stevens[1], Pierre Karpman[2,3,4], and Thomas Peyrin[4]

[1] Centrum Wiskunde & Informatica, The Netherlands
[2] Inria, France
[3] École polytechnique, France
[4] Nanyang Technological University, Singapore

marc.stevens@cwi.nl, pierre.karpman@inria.fr, thomas.peyrin@ntu.edu.sg

Abstract. This article presents an explicit freestart colliding pair for SHA-1, *i.e.* a collision for its internal compression function. This is the first practical break of the full SHA-1, reaching all 80 out of 80 steps. Only 10 days of computation on a 64-GPU cluster were necessary to perform this attack, for a cost of approximately $2^{57.5}$ calls to the compression function of SHA-1. This work builds on a continuous series of cryptanalytic advancements on SHA-1 since the theoretical collision attack breakthrough of 2005. In particular, we reuse the recent work on 76-step SHA-1 of Karpman *et al.* from CRYPTO 2015 that introduced an efficient framework to implement (freestart) collisions on GPUs; we extend it by incorporating more sophisticated accelerating techniques such as boomerangs. We also rely on the results of Stevens from EUROCRYPT 2013 to obtain optimal attack conditions; using these techniques required further refinements for this work.

SHATTERED

CRYPTO 2017

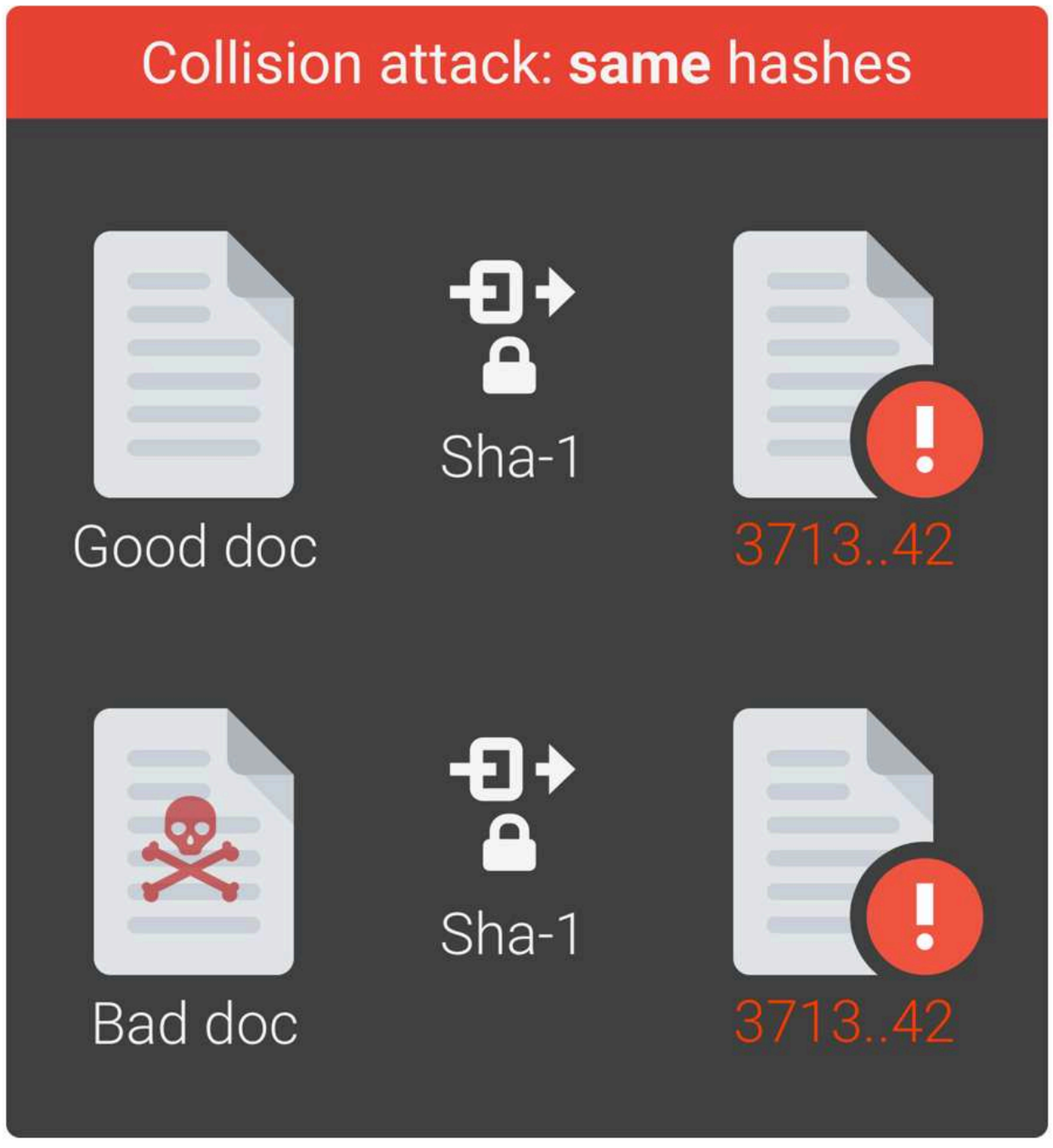
We have broken SHA-1 in practice.

This industry cryptographic hash function standard is used for digital signatures and file integrity verification, and protects a wide spectrum of digital assets, including credit card transactions, electronic documents, open-source software repositories and software updates.

It is now practically possible to craft two colliding PDF files and obtain a SHA-1 digital signature on the first PDF file which can also be abused as a valid signature on the second PDF file.

For example, by crafting the two colliding PDF files as two rental agreements with different rent, it is possible to trick someone to create a valid signature for a high-rent contract by having him or her sign a low-rent contract.

Infographic | Paper



EUROCRYPT 2019

From Collisions to Chosen-Prefix Collisions Application to Full SHA-1

Gaëtan Leurent[1] and Thomas Peyrin[2,3]

[1] Inria, France
[2] Nanyang Technological University, Singapore
[3] Temasek Laboratories, Singapore

gaetan.leurent@inria.fr, thomas.peyrin@ntu.edu.sg

Abstract. A chosen-prefix collision attack is a stronger variant of a collision attack, where an arbitrary pair of challenge prefixes are turned into a collision. Chosen-prefix collisions are usually significantly harder to produce than (identical-prefix) collisions, but the practical impact of such an attack is much larger. While many cryptographic constructions rely on collision-resistance for their security proofs, collision attacks are hard to turn into a break of concrete protocols, because the adversary has limited control over the colliding messages. On the other hand, chosen-

Real World Crypto 2020

SHA-1 is a Shambles

We have computed the very first **chosen-prefix collision for SHA-1**. In a nutshell, this means a complete and practical break of the SHA-1 hash function, with dangerous practical implications if you are still using this hash function. To put it in another way: all attacks that are practical on MD5 are now also practical on SHA-1.

Check our paper **here** for more details. Slides from RWC are also available.

Our Contributions

Complexity Improvements

We have significantly improved the complexity of SHA-1 attacks, with a speedup factor around 10. More precisely, we have reduced the cost of a collision attack from $2^{64.7}$ to $2^{61.2}$, and the cost of a chosen-prefix collision attack from $2^{67.1}$ to $2^{63.4}$ (on a GTX 970 GPU).

Record Computation

We implemented the entire chosen-prefix collision attack with those improvements. This attack is extremely technical, contains many details, various steps, and requires a lot of engineering work. In order to perform this computation with a small academic budget, we rented cheap gaming or mining GPUs from GPUserversrental, rather that the datacenter-grade hardware used by big cloud providers. We have successfully run the computation during two months last summer, using 900 GPUs (Nvidia GTX 1060).

As a side result, this shows that it now costs less than 100k USD to break cryptography with a security level of 64 bits (i.e. to compute 2^{64} operations of symmetric cryptography).

FIPS PUB 202

FEDERAL INFORMATION PROCESSING STANDARDS PUBLICATION

SHA-3 Standard: Permutation-Based Hash and Extendable-Output Functions

CATEGORY: COMPUTER SECURITY SUBCATEGORY: CRYPTOGRAPHY

Information Technology Laboratory
National Institute of Standards and Technology
Gaithersburg, MD 20899-8900

Function	Output Size	Security Strengths in Bits		
		Collision	Preimage	2nd Preimage
SHA-1	160	< 80	160	$160 - L(M)$
SHA-224	224	112	224	$\min(224, 256 - L(M))$
SHA-512/224	224	112	224	224
SHA-256	256	128	256	$256 - L(M)$
SHA-512/256	256	128	256	256
SHA-384	384	192	384	384
SHA-512	512	256	512	$512 - L(M)$
SHA3-224	224	112	224	224
SHA3-256	256	128	256	256
SHA3-384	384	192	384	384
SHA3-512	512	256	512	512
SHAKE128	d	$\min(d/2, 128)$	$\geq \min(d, 128)$	$\min(d, 128)$
SHAKE256	d	$\min(d/2, 256)$	$\geq \min(d, 256)$	$\min(d, 256)$

Table 4: Security strengths of the SHA-1, SHA-2, and SHA-3 functions

Que faut-il utiliser ?

MD5 et **SHA-1** sont désuets

SHA-256 **& co** compromis raisonnable

SHA-3 pour le futur ? reste à le tester…

Codes HMAC

HMAC : du hachage au MAC

Principe : transformer toute fonction de hachage de qualité cryptographique en un code d'authentification de message (MAC).

HMAC-MD5

HMAC-SHA-1

HMAC-SHA256

FIPS PUB 198-1

FEDERAL INFORMATION PROCESSING STANDARDS PUBLICATION

The Keyed-Hash Message Authentication Code (HMAC)

CATEGORY: COMPUTER SECURITY **SUBCATEGORY: CRYPTOGRAPHY**

Table 1: The HMAC Algorithm

STEPS	STEP-BY-STEP DESCRIPTION
Step 1	If the length of $K = B$: set $K_0 = K$. Go to step 4.
Step 2	If the length of $K > B$: hash K to obtain an L byte string, then append $(B\text{-}L)$ zeros to create a B-byte string K_0 (i.e., $K_0 = H(K) \| 00...00$). Go to step 4.
Step 3	If the length of $K < B$: append zeros to the end of K to create a B-byte string K_0 (e.g., if K is 20 bytes in length and $B = 64$, then K will be appended with 44 zero bytes x'00').
Step 4	Exclusive-Or K_0 with *ipad* to produce a B-byte string: $\boldsymbol{K_0 \oplus ipad}$.
Step 5	Append the stream of data '*text*' to the string resulting from step 4: $\boldsymbol{(K_0 \oplus ipad) \| text}$.
Step 6	Apply H to the stream generated in step 5: $\mathbf{H((K_0 \oplus ipad) \| text)}$.
Step 7	Exclusive-Or K_0 with *opad*: $\boldsymbol{K_0 \oplus opad}$.
Step 8	Append the result from step 6 to step 7: $\mathbf{(K_0 \oplus opad) \| H((K_0 \oplus ipad) \| text)}$.
Step 9	Apply H to the result from step 8: $\boldsymbol{H((K_0 \oplus opad) \| H((K_0 \oplus ipad) \| text))}$.

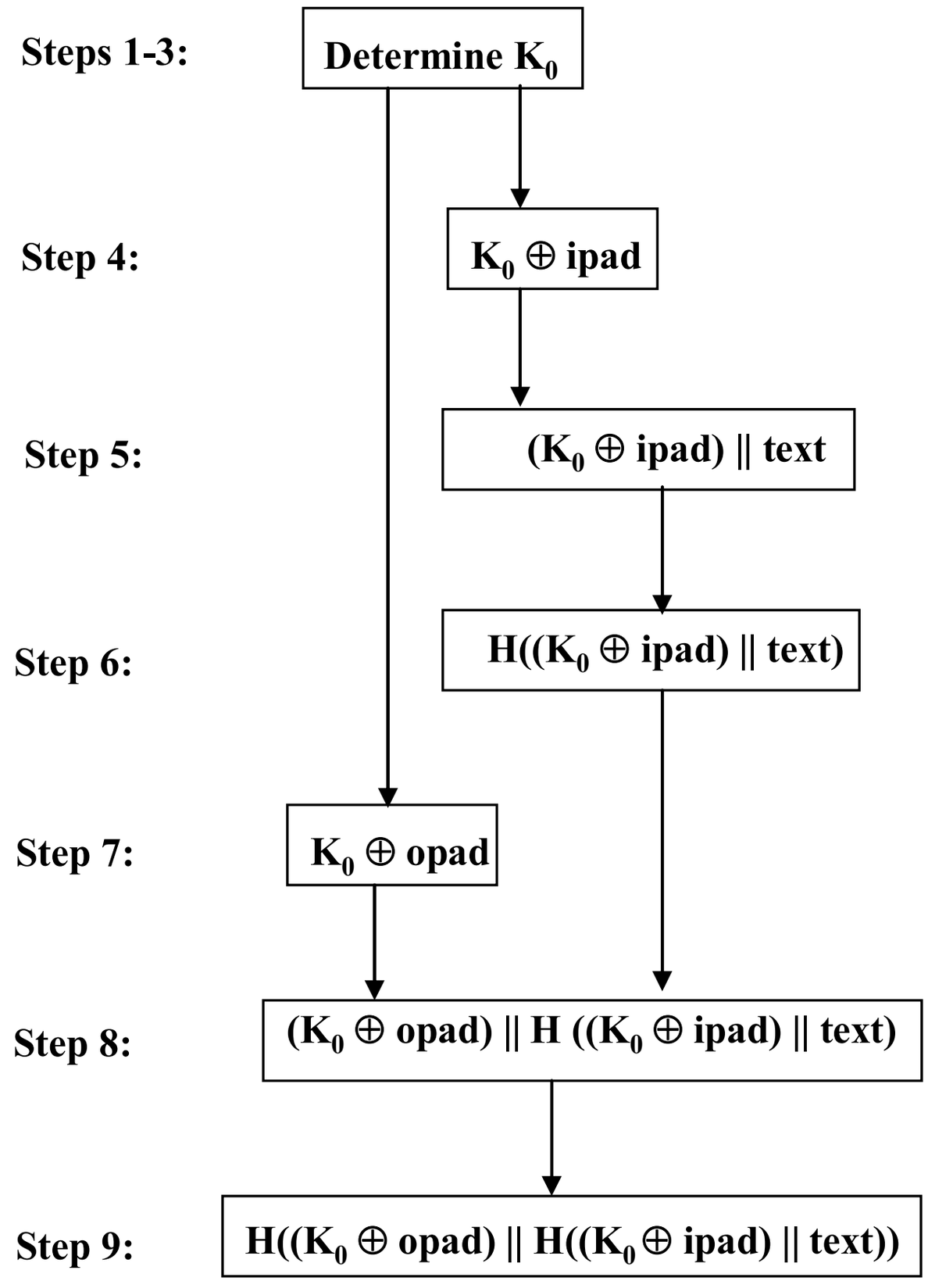


Figure 1: Illustration of the HMAC Construction

2.3 HMAC Parameters and Symbols

HMAC uses the following parameters:

B	Block size (in bytes) of the input to the Appr
H	An Approved hash function.
ipad	Inner pad; the byte x‘36’ repeated *B* times.
K	Secret key shared between the originator and
K_0	The key *K* after any necessary pre-processing
L	Block size (in bytes) of the output of the App
opad	Outer pad; the byte x‘5c’ repeated *B* times.
text	The data on which the HMAC is calculated; *t* The length of *text* is *n* bits, where $0 \leq n < 2^B$ -
x ‘N’	Hexadecimal notation, where each symbol in
bits.	
||	Concatenation.
$\oplus$	Exclusive-Or operation.

Pause exercice

Cette histoire de HMAC semble bien compliqué à Alice, qui préfère utiliser MD5(K‖M) pour le contrôle d'intégrité de ses messages.

Pourquoi est-ce une très mauvaise idée ?

Keying Hash Functions for Message Authentication

Mihir Bellare* Ran Canetti† Hugo Krawczyk‡

June 1996

Abstract

The use of cryptographic hash functions like MD5 or SHA for message authentication has become a standard approach in many Internet applications and protocols. Though very easy to implement, these mechanisms are usually based on ad hoc techniques that lack a sound security analysis.

We present new constructions of message authentication schemes based on a cryptographic hash function. Our schemes, NMAC and HMAC, are proven to be secure as long as the underlying hash function has some reasonable cryptographic strengths. Moreover we show, in a quantitative way, that the schemes retain almost all the security of the underlying hash function. In addition our schemes are efficient and practical. Their performance is essentially that of the underlying hash function. Moreover they use the hash function (or its compression function) as a black box, so that widely available library code or hardware can be used to implement them in a simple way, and replaceability of the underlying hash function is easily supported.

New Proofs for NMAC and HMAC: Security without Collision-Resistance

Mihir Bellare*

April 2014

Abstract

HMAC was proved in [4] to be a PRF assuming that (1) the underlying compression function is a PRF, and (2) the iterated hash function is weakly collision-resistant. However, subsequent attacks showed that assumption (2) is false for MD5 and SHA-1, removing the proof-based support for HMAC in these cases. This paper proves that HMAC is a PRF under the sole assumption that the compression function is a PRF. This recovers a proof based guarantee since no known attacks compromise the pseudorandomness of the compression function, and it also helps explain the resistance to attack that HMAC has shown even when implemented with hash functions whose (weak) collision resistance is compromised. We also show that an even weaker-than-PRF condition on the compression function, namely that it is a privacy-preserving MAC, suffices to establish HMAC is a secure MAC as long as the hash function meets the very weak requirement of being computationally almost universal, where again the value lies in the fact that known attacks do not invalidate the assumptions made.