Key derivation function: an essential (and usually transparent) component of real-world applications

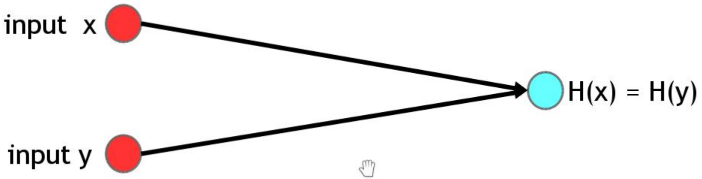
Prof. Andrea VISCONTI

Department of Computer Science
Università degli Studi di Milano

Hash Functions

Properties...

1. **Collision-free**



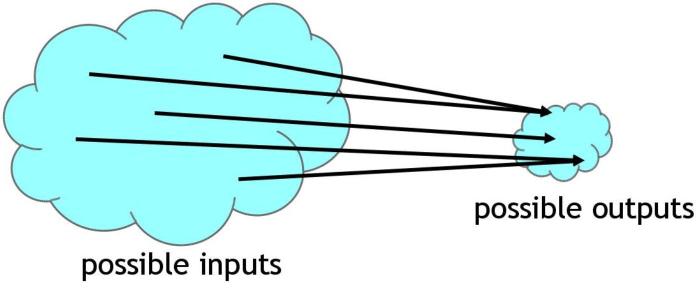
Nobody is able to find two strings x and y s.t. $x \neq y$ and $H(x) = H(y)$

Hash Functions

Collision do exist!

Indeed,

- we input any string of any size — say n, this means 2^n;
- we provide a fixed size output — for example x bits, this means 2^x;

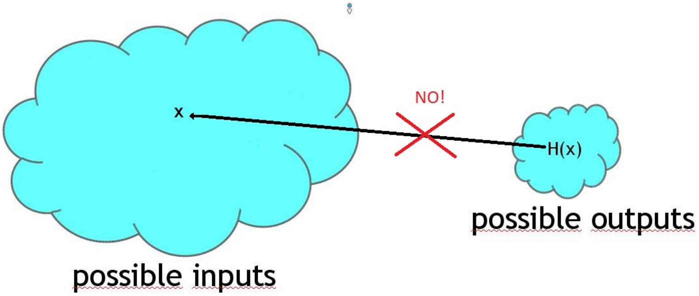


But are you able to find them?

Hash Functions

Properties...

2. **Preimage resistant**

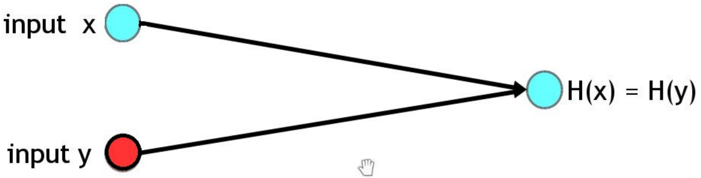


Given $H(x)$, it is computationally infeasible to find x.

Hash Functions

Properties...

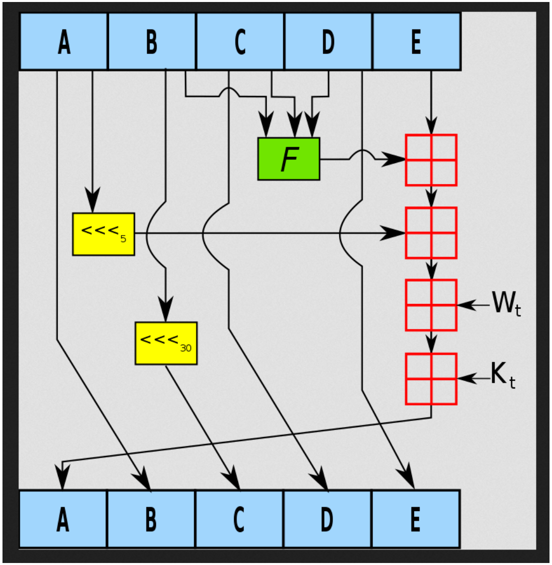
3. **Second preimage resistant**



Given x, it is computationally infeasible to find y s.t. $H(x) = H(y)$.

These three properties ensure that it is hard to cheat.

SHA-1... wiki



Problem Description

Passwords are widely used to protect **secret data** or to gain **access to specific resources**.

They **should be strong** enough to prevent well-know attacks (e.g. dictionary and brute force attacks).

User-chosen passwords are generally **short** and **lack enough entropy**.

They **cannot be directly used as a key** to implement secure cryptographic systems.

Problem Description

Passwords –> HASH(passwords) –> KDF(passwords).

CPU-intensive

Memory-intensive

A possible solution

A possible solution to these issues is to adopt a **Key Derivation Function** (KDF).

Why are **Password-Based Key Derivation Functions** of particular interest in cryptography?
Because they

- input a password/passphrase and derive a cryptographic key;
- allow to increase the size of this key;
- introduce CPU-intensive (or memory) operations;
- allow legitimate users to spend a moderate amount of time on key derivation;
- slow down brute force and dictionary attacks as much as possible.

PBKDF2

In **PKCS#5**, RSA Laboratories described **Password-Based Key Derivation Function version 2** (PBKDF2).

PBKDF2 is one of the **most widely used KDF** in real applications.

(**Wiki...**) PBKDF2 is implemented in many systems:

- Wi-Fi Protected Access security protocols (i.e., WPA and WPA2);
- Firefox Sync;
- iOS passcodes;
- Android full disk encryption (since version 3.0 to 4.3);
- FileVault encryption on Mac;
- Linux Unified Key Setup (LUKS) disk encryption specification;
- WinZip encryption;
- GRUB2 boot loader;
- ...

How does PBKDF2 work?

PBKDF2 inputs

- a user **password/passphrase** p;
- a **random salt** s;
- an **iteration counter** c;
- **derived key length** $dkLen$.

PBKDF2 outputs a **derived key** DK:

$$DK = PBKDF2(p, s, c, dkLen)$$

PBKDF2 can derive keys of **arbitrary length**.

How does PBKDF2 work?

The underlying **pseudo-random function** is HMAC-SHA-1 by default.

Salt: To prevent building universal dictionaries.

Iterations count: defined a priori or at runtime.

In order to slow down the attackers, PBKDF2 introduces **CPU-intensive operations based on an iterated Pseudo-Random Function** (PRF).

As suggested in SP 800-132 (December 2010), it is a **good practice to select the iteration count as large as possible**, as long the time required to generate the key is acceptable for the user.

How does PBKDF2 work?

More precisely, PBKDF2 generates as many blocks T_i as needed to cover the desired key length *dkLen*.

The length of each block T_i is bounded by *hLen*, which is **the length of the underlying PRF output** (e.g. 160 bits (SHA-1), 256 bits (SHA-256), and so on).

$$DK = T_1 || T_2 || \ldots || T_{\lceil dkLen/hLen \rceil}$$

Each block T_i is computed **iterating the PRF many times** as specified by an iteration count c.

$$T_i = U_1 \oplus U_2 \oplus \ldots \oplus U_c$$

How does PBKDF2 work?

Now we can compute the Pseudo-Random Function (PRF):

$$
\begin{aligned}
U_1 &= PRF(p, s||i) \\
U_2 &= PRF(p, U_1) \\
U_3 &= PRF(p, U_2) \\
&\vdots \\
U_c &= PRF(p, U_{c-1})
\end{aligned}
$$

Recall that iteration count c is used to **slow down an attacker** as much as possible.

How does PBKDF2 work?

$$DK = PBKDF2(p, s, c, dkLen)$$

More precisely, the derived key is computed as follows:

$$DK = T_1 || T_2 || \dots || T_{\lceil dkLen/hLen \rceil}$$

Each single block T_i is computed as

$$T_i = U_1 \oplus U_2 \oplus \dots \oplus U_c$$

where

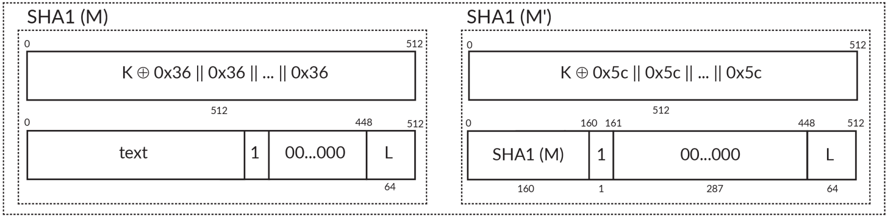
$$U_1 = PRF(p, s||i)$$
$$U_2 = PRF(p, U_1)$$
$$\vdots$$
$$U_c = PRF(p, U_{c-1})$$

About PRF: usually HMAC

HMAC can be defined as follows:

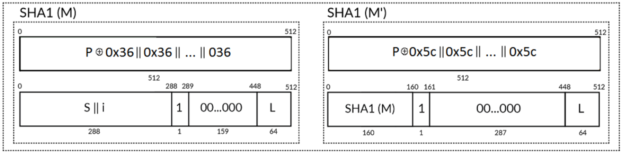
$$HMAC = H(K \oplus opad, H(K \oplus ipad, text))$$

and it can be graphically represented as

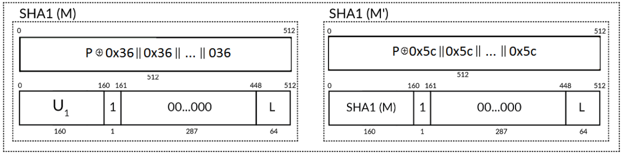


About PRF: usually HMAC (2)

U_1 = HMAC-SHA1 (P, S || i)

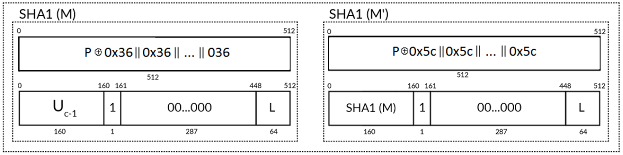


U_2 = HMAC-SHA1 (P, U_1)



$\vdots$

U_c = HMAC-SHA1 (P, U_{c-1})



Weaknesses

RFC 2104 (Feb 1997): IMPLEMENTATION NOTE

HMAC is defined . . . However, if desired, a **performance improvement can be achieved** at the cost of (possibly) modifying the code . . .
The idea is that the intermediate results of the compression function on the B-byte blocks $(K \oplus ipad)$ and $(K \oplus opad)$ can be precomputed only once at the time of generation of the key K, or before its first use. These intermediate results are stored and then used to initialize the IV of H each time that a message needs to be authenticated.
This method saves, for each authenticated message, the application of the compression function of H on two B-byte blocks (i.e., on $(K \oplus ipad)$ and $(K \oplus opad)$).
Such a savings may be significant when ... Choosing to implement HMAC in the above way is a decision of the local implementation and has no effect on inter-operability.

FIPS 198 (March 2002), FIPS 198-1 (July 2008)

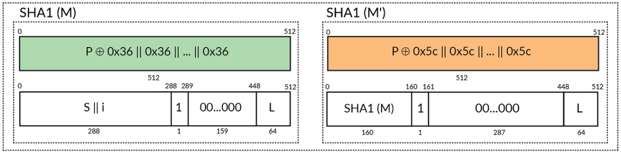
IMPLEMENTATION NOTE:

The HMAC algorithm is . . . Conceptually, the intermediate results of the compression function on the B-byte blocks $(K \oplus ipad)$ and $(K \oplus opad)$ can be precomputed once, at the time of generation of the key K, or before its first use. These intermediate results can be stored and then used to initialize H each time that a message needs to be authenticated using the same key.
For each authenticated message using the key K, this method saves the application of the hash function of H on two B-byte blocks (i.e., on $(K \oplus ipad)$ and $(K \oplus opad)$).
This saving may be significant when ...

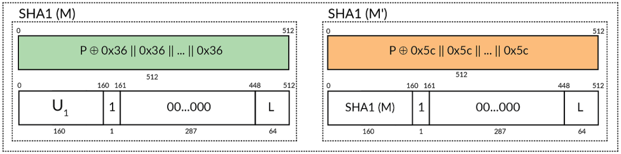
This means that RFC 2104 and FIPS 198 **suggest us a way to avoid 50% of PBKDF2's CPU intensive operations**, by replacing them with **precomputed values**.

FIRST: Precomputing a message block

U_1 = HMAC-SHA1 (P, S || i)

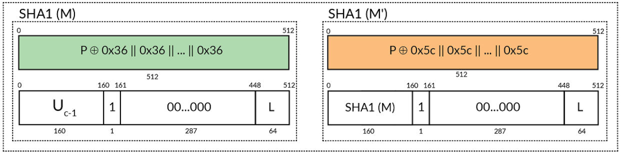


U_2 = HMAC-SHA1 (P, U_1)



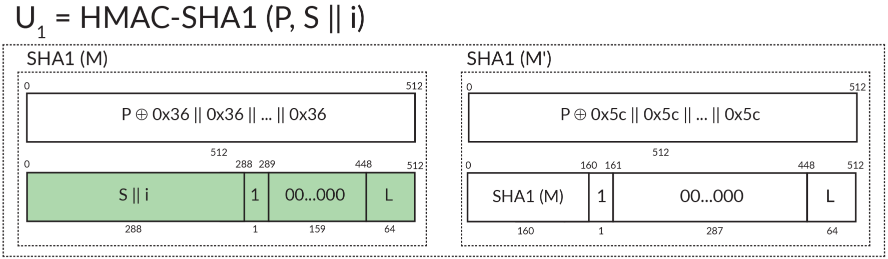
⋮

U_c = HMAC-SHA1 (P, U_{c-1})



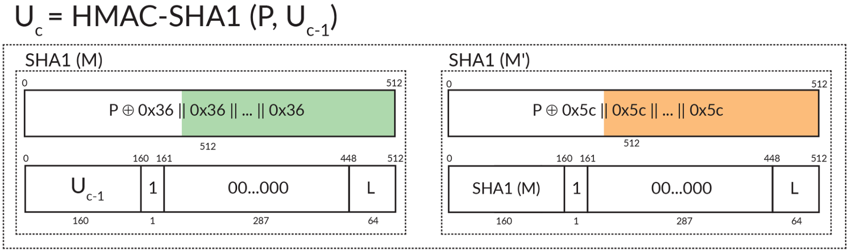
SECOND: Precomputing a word-expansion

A minor weakness provides the possibility to precompute the word-expansion part of the second message block of a keyed hash function (green rectangle). Indeed, **such a block is password-independent**, and given a salt *s* (recall that *s* is a public information) an attacker is able to compute the expansion $W_0 \ldots W_{79}$ in advance.



THIRD: Useless XOR operations

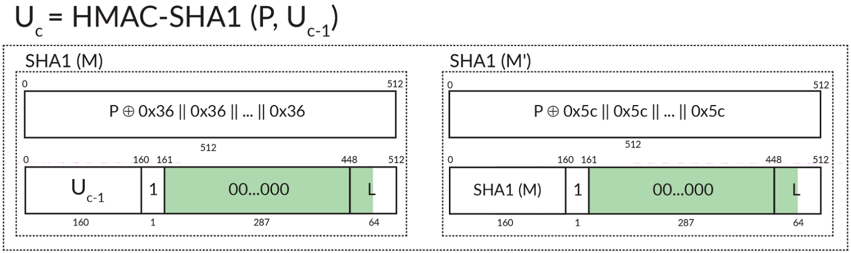
The constant 0x36 and 0x5c are used to pad the first message block up to the hash block size (green and orange rectangles). This means that a number of W_t are set to the same value.



$$W_t = ROTL^1(W_{t-3} \oplus W_{t-8} \oplus W_{t-14} \oplus W_{t-16}) \qquad t \in [16 \ldots 79]$$

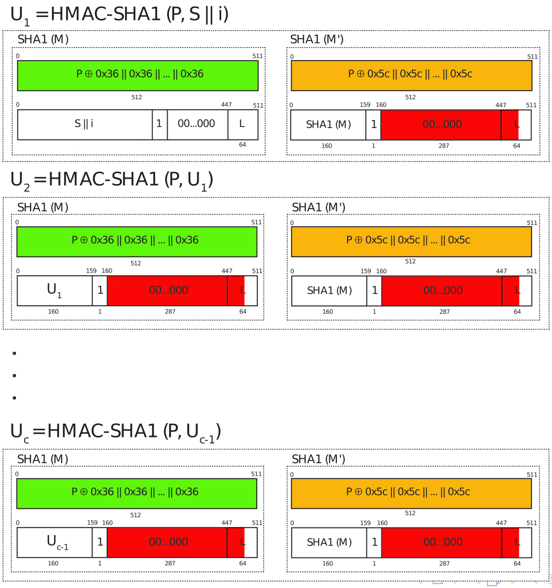
THIRD: Useless XOR operations (2)

It is easy to observe that each SHA-1 message block has **a run of several consecutive zeros** (i.e., a number of $W_t = 0$).



$$W_t = ROTL^1(W_{t-3} \oplus W_{t-8} \oplus W_{t-14} \oplus W_{t-16}) \qquad t \in [16 \ldots 79]$$

FOURTH: Useless XOR operations



U_1 = HMAC-SHA1 (P, S || i)
SHA1 (M)
P ⊕ 0x36 || 0x36 || ... || 0x36
0
511
512
S || i
1
00...000
L
447
64
SHA1 (M')
P ⊕ 0x5c || 0x5c || ... || 0x5c
SHA1 (M)
159
160
287
U_2 = HMAC-SHA1 (P, U_1)
U_1
U_c = HMAC-SHA1 (P, U_{c-1})
U_{c-1}

FOURTH: Useless XOR operations

In 2016 [?], we show that eighty words W_i can be also represented as:

$$W_i = \begin{cases} ROTL^1(W[i-3] \oplus W[i-8] \oplus W[i-14] \oplus W[i-16]) & i \in [16 \ldots 31] \\ ROTL^2(W[i-6] \oplus W[i-16] \oplus W[i-28] \oplus W[i-32]) & i \in [32 \ldots 63] \\ ROTL^4(W[i-12] \oplus W[i-32] \oplus W[i-56] \oplus W[i-64]) & i \in [64 \ldots 79] \end{cases}$$

Using the new message scheduling function, the number of useless XOR operations is increased from 27 to 61 (out of 192).

$$w[70] = ROTL^1(w[67] \oplus w[62] \oplus w[56] \oplus w[54])$$

$$\begin{aligned} w[70] &= ROTL^4(w[58] \oplus w[38] \oplus w[14] \oplus w[6]) \\ &= ROTL^4(w[58] \oplus w[38] \oplus 0 \oplus 0) \end{aligned}$$

FOURTH: Useless XOR operations

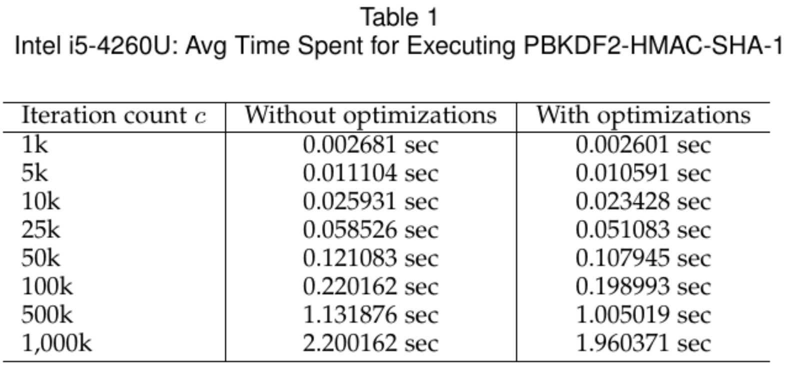


Table 1
Intel i5-4260U: Avg Time Spent for Executing PBKDF2-HMAC-SHA-1

Iteration count c	Without optimizations	With optimizations
1k	0.002681 sec	0.002601 sec
5k	0.011104 sec	0.010591 sec
10k	0.025931 sec	0.023428 sec
25k	0.058526 sec	0.051083 sec
50k	0.121083 sec	0.107945 sec
100k	0.220162 sec	0.198993 sec
500k	1.131876 sec	1.005019 sec
1,000k	2.200162 sec	1.960371 sec

Optimizations

[OPT–01] Early exit:

Assuming that we require a 256-bit derived key, two SHA-1 fingerprint are necessary — i.e., $DerKey = T_1 || T_2$, with T_1 and T_2 160-bit length each.

Since blocks T_i are independent of each other, firstly we generate a block T_1 and then we compute the second if and only if T_1 is equal to the first part of the 256-bit derived key.
If not so, the chosen password p is certainly wrong.

Optimizations (2)

[OPT–02] Block reduction:

To precompute the first message block of a keyed hash function (green and orange rectangle) and reuse such a value in all the subsequent HMAC invocations reduces the number of blocks that have to be computed from "$4 * \textit{iteration count}$" to "$2 + 2 * \textit{iteration count}$".

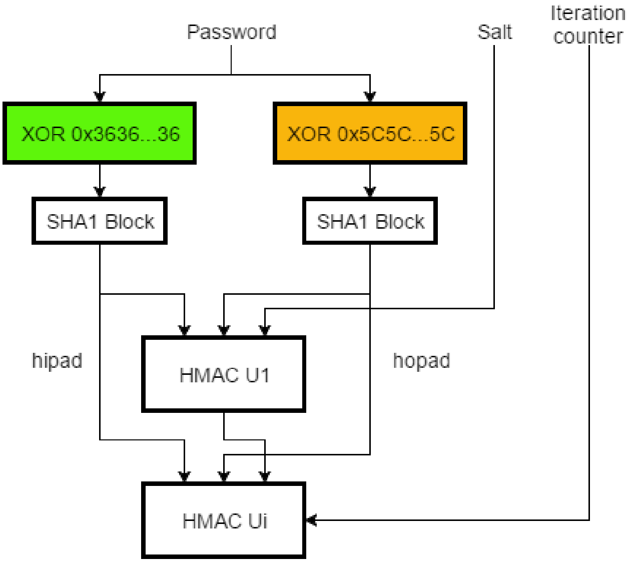
Optimizations (3)

[OPT–03] Input size:

A generic HMAC implementation has to address the problems of the size of password p and message *text*. In PBKDF2, excluding the computation of U_1, we have not a generic HMAC implementation but a specific one. We known in advance the computation of the first message block (i.e. [OPT–02]), and we have to manage only the second one.

Since the second message block always inputs a 160-bit message, namely *SHA-1(M)*, we avoid **length checks** and the **chunk splitting** operations during the computation of $U_2, \ldots, U_c$, thus **reducing the overhead** necessary to compute an HMAC implementation.

Optimizations (3)



Optimizations (4)

[OPT–04] Useless XOR operations:

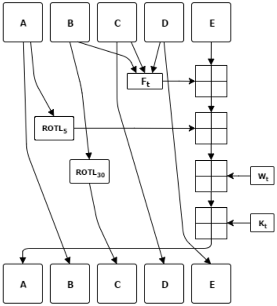
$$W_i = \begin{cases} ROTL^1(W[i-3] \oplus W[i-8] \oplus W[i-14] \oplus W[i-16]) & i \in [16 \ldots 31] \\ ROTL^2(W[i-6] \oplus W[i-16] \oplus W[i-28] \oplus W[i-32]) & i \in [32 \ldots 63] \\ ROTL^4(W[i-12] \oplus W[i-32] \oplus W[i-56] \oplus W[i-64]) & i \in [64 \ldots 79] \end{cases}$$

Exploiting the previous Equations, we can avoid 61 out of 192 XOR operations.

Optimizations (5)

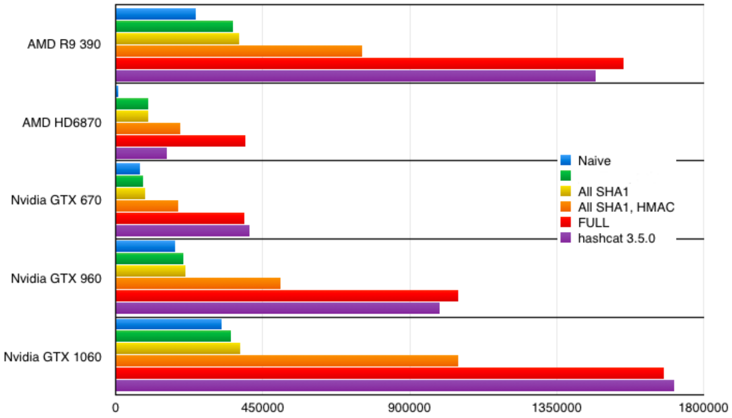
[OPT–05] Three-round optimization

In the first round we have to compute the following equation:
$f_0 + E + ROTL(A, 5) + W_0 + K_0$.

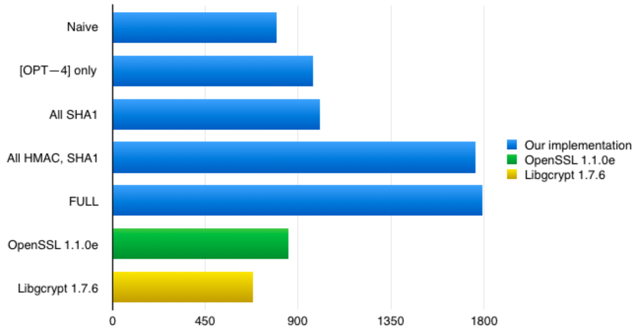


We can precompute $f_0 + E + ROTL(A, 5) + K_0 = 0x9FB498B3$ and reduce the first round to a single operation, thus saving 3 operations out of 4. This approach can be also applied to second and third round.

GPU performances



CPU performances (AMD 8 cores, 4GHz)



Alternative approaches

Password Hashing Competition (PHC) and KDFs

- PBKDF2 (standard de facto)
- Argon
- Lyra
- Makwa
- Catena
- Yescrypt
- ...

Alternative approaches

They are based on...

- Hash functions
- HMAC
- AES-XTS
- Sponge functions
- LFSR
- ...

Thank you for your attention!