A Framework for Iterative Hash Functions:

HAIFA

(**HA**sh **I**terative **F**r**A**mework)

Eli Biham Orr Dunkelman

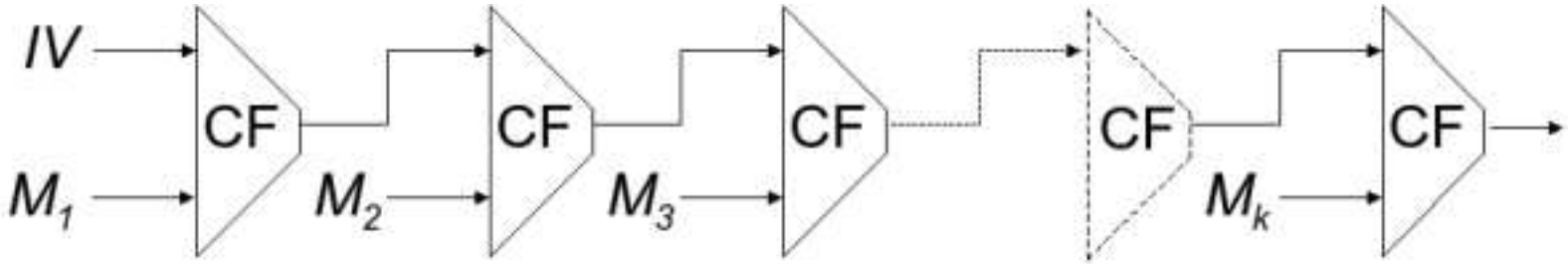
Computer Science Department
Technion, Haifa 32000, Israel

August 25, 2006

The Merkle-Damgård Construction

A method to compute the digest of a message M in one pass, using a compression function with a fixed input size and a fixed output size.

The compression function is represented by $CF : \{0,1\}^{m_c} \times \{0,1\}^{n} \rightarrow \{0,1\}^{m_c}$.



We denote the length of the digest by m (and assume the common case $m_c = m$).

Attacks against the Merkle-Damgård Construction

Let k be the number of blocks of a given message M, and let k' be the number of given messages (for one-of-many attacks).

1. Easily invertible compression functions (the backward attack) — preimage attack — $2^{m_c/2}$ (late 1970's)
2. One-of-many preimage attacks, one-of-many second preimage attacks — $2^m/k'$, $(O(2^{m/2}))$ (Merkle 1979)
3. Multi-collision — $O(2^{m_c/2})$ (Joux, 2004)
4. Fixpoint and second preimage attacks (long messages) — $2^{m_c}/k$, $(O(2^{m_c/2}))$ (Dean, 1999) [mainly for Davies-Meyer functions]
5. One-of-many second preimage attacks — $2^{m_c}/(kk')$, $(O(2^{m_c/2}))$ (Dean, 1999)
6. Expandable messages and second preimage attacks (long messages) — $2^{m_c}/k$, $(O(2^{m_c/2}))$ (Kelsey, Schneier, 2005) [any iterative hash function]
7. Applicable also to one-of-many second preimage attacks (they missed to mention) — $2^{m_c}/(kk')$, $(O(2^{m_c/2}))$

Attacks against the Merkle-Damgård Construction (cont.)

8. Herding attacks (a commitment to some digest value, with a respective pre-computation that allows concatenating any prefix) — Offline $O(2^{m_c/2+t})$, Online: $O(2^{m_c-t})$.

Possible Conclusions

We have to accept that second preimages are as easy to find as collisions.

A possible solution: Incresing the size of the chaining value (m_c) while the digest size (m) remains unchanged.

The wide hash strategy uses this idea. This approach increases the memory requirements of the construction.

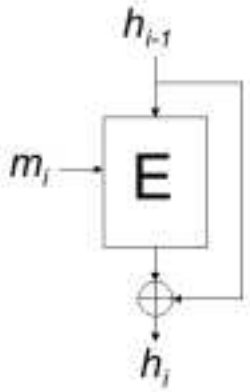
Bound the size of a message by at most $2^{m/2}$ blocks (or bits) — does not help, as the current limit is even smaller (2^{64} bits= 2^{55} blocks).

Fix-Points

The standard protection against backward attacks on hash functions with easily invertible compression functions involves mixing the input to the output of the function (Davies-Meyer).
Though the function becomes difficult to invert with this solution, it is still easy to find fix-points:

1. Select the output before the feedback to be zero.
2. Invert the functions to get the input chaining value h_{i-1}.
3. The output chaining value equals the input chaining value $h_i = h_{i-1}$.

Fix-points can be used to find second preimages.



Fix-Points (cont.)

Protection: Either

1. Apply the mixing more than once, say every 20 rounds in SHA-1.
2. Mix the chaining value into the expanded message (e.g., like a few additional message words, or by XORing them to a few words)
3. In the design of a new compression function, there is a lot of freedom for such a mixing.

This protection against easily-to-find fixpoints does not help against expandable messages.

Protection Against the Second Preimage Attacks

Kelsey and Schneier proposed to add the block index as an input to the compression function, i.e.,

$$h_i = CF(h_{i-1}, M_i, i),$$

instead of

$$h_i = CF(h_{i-1}, M_i).$$

But the block index is never kept nowadays in the state of the hash.
We propose to use the current bit count as an additional input

$$h_i = CF(h_{i-1}, M_i, \#\text{bits}),$$

where #bits is the number of already hashed bits, including this block.
This #bits is already computed by every hash function, to the padding with the length.

Protection Against the Second Preimage Attacks (cont.)

This solution protects against the second preimage attacks, as each computed block becomes **static** — cannot be used in any other location of any message, just at the original location.

It also ensures that there will not be cycles of chaining values (even if there is some value twice, e.g., as a fix-point, it will not be repeated).

Notice that it is **not needed to increase** the chaining size m_c any more.

Protection Against Message Expansion

The previous solution also protects against the ability to append data at end of a message, without knowing the message (e.g., when the hash function is used for MAC).
In block i,

$$\begin{aligned} \#\text{bits} &= n \cdot i, \\ h_i &= CF(h_{i-1}, M_i, n \cdot i). \end{aligned}$$

In the padding block(s), and in particular in the last block k:

$$\begin{aligned} \#\text{bits} &= \text{length} \neq k \cdot i, \\ \#\text{bits} &= \text{length} \leq k \cdot i - \text{message length field}, \\ h_i &= CF(h_{i-1}, M_i, \text{length}) \neq CF(h_{i-1}, M_i, n \cdot i). \end{aligned}$$

Protection Against Message Expansion (cont.)

But, in order to append another block to the message, the attacker will have to know

$$CF(h_{k-1}, M_k, k \cdot i),$$

rather than

$$CF(h_{k-1}, M_k, \text{length}),$$

so it is no longer possible to append extra blocks!

[similar property would not hold if the block index would be used]

Variable Hash Size

In some functions the hash size can be truncated (e.g., Tiger), or uses a truncated version of another function (e.g., SHA-384).
In some of these cases the initial value is replaced to ensure that the output will be different.
We propose a general construction for such functions.

Variable Hash Size (cont.)

Let $h_i = CF(h_{i-1}, M_i)$ be the compression function of some hash function with m_c bits of chaining, and let $0 < m \leq m_c$ be the (possibly) truncated output size.

Let IV_m be the initial value for the function with the m bits of output.

We propose to select a global IV, and compute

$$IV_m = CF(m, IV, \#\text{bits})$$

(or $IV_m = CF(m, IV)$), where m is padded in some standard way to fit a block.

Note that any suggestion such that $m \geq m_c$ can not achieve greater security than possible when $m_c = m$.

Variable Hash Size (cont.)

Advantages:

1. A general method for truncation of the hash size for any hash function
2. IV_m can be computed in advance, so that no extra time is required to hash a message (just like it is done in SHA-384, SHA-512)
3. On the other hand, applications that need several hash sizes, can compute IV_m on the fly, or actually compute

$$\text{truncate}_m(\text{HashFromIV}(m||M)).$$

Variable Hash Size (cont.)

A potential problem:

1. The compression function is the same for all these functions
2. An attacker may wish to have the same hash value for two different hash sizes (up to truncation)
3. If the attacker has the freedom to select different prefixes, he may apply a collision attack (or birthday attack) on the first few blocks, getting the same chaining value. The rest of the chaining values will be the same, as well as the hash value (up to truncation).
4. In order to solve this possible problem, we propose to change the padding algorithm slightly to include the hash size as well:
   (a) append a single 1, and may 0's.
   (b) **append hash size**.
   (c) append length.
5. It may also be useful to input the hash size as an additional input to the compression function, but at this moment it seems unnecessary.

Last Modification: Family of Hash Functions and Salts

Observe that theoretic definitions of hash functions define **families** of hash functions $h_i()$, $i \in \{0, 1, \ldots, l\}$.

All the solutions shown till now cannot protect against one-of-many second preimage attacks.

We can adopt families of hash functions as a solution: Users will select a one function of the family every time they compute hash, either at random, or by incrementing by one, or as the frame number or sequence number of the message that is transmitted (there is such a number in every secure communication frame) or of the document (many signature implementations keep sequential numbers anyway), or date, etc.

[The member number might be called an IV in the sense of IV in a stream cipher, but the term is already used; we will call it **salt** instead, as in UNIX password hashing]

Such a modified definition of a cryptographic hash function should protect against one-of-many preimages, and other attacks that use several messages generated by the legal user.

Last Modification: Family of Hash Functions and Salts (cont.)

The salt should be used as an additional input to the compression function (but not change the IV), and should be added to the padding.
Some applications that cannot select a hash function from a family, due to application dependent requirements, may select a fixed member (e.g., salt=0).
The cost of this solution is negligible.

In some sense, salt can be viewed as a long-term non-colliding value that protects different hashes from colliding, just like the serial number does for different blocks in the same message.
Salt can also be used as a key for keyed hash functions — may be possible to design unified HASH/MAC designs. We did not explore that yet.
The salt suggestion has similarties to the randomized hashing proposal. However, unlike in randomized hashing, in the HAIFA framework, the salt (random string) is added to each and every compression function call. This allows a better protection against the herding attack.

New Designs

New designs of hash functions can easily apply these guidelines, with compression functions of the form

$$h_i = CF(h_{i-1}, M_i, \#\text{bits}, \text{salt}),$$

where $h_0 = IV_m$ is

$$IV_m = CF(m, IV, 0, 0),$$

and with padding that include a single 1, many 0's, salt, m, and the length of the message.

Old Designs

Notice that the change of initial values for different hash sizes can easily be combined into any existing iterative function, by prepending a block with the hash size m to any message (and truncating the result to m bits). The change of the padding is however more complicated.

All change that add parameters to the compression functions, can instead be considered as part of the message block, effectively reducing the block size.

Some implementations may prefer to add a block with that extra information every several blocks, for example, making a tradeoff between the extra protection and the cost in speed and complexity of the application. This approach is very risky when the number of blocks is relatively small (allowing for herding attacks).

The extra fields in the padding can be added at the end of the message before the original padding, without being counted in #bits.

Summary

1. We proposed the HAIFA framework for design of new hash functions: a mode of operation with a modified kind of compression functions
2. This framework protects against most known generic attacks.
3. A few attacks can only be protected by increasing m_c (e.g., multi-collisions) — in case they are considered a security risk, then $m < m_c$ should be selected. However, $\boldsymbol{m = m_c}$ is still OK against second preimage attacks.
4. Existing hash functions can be used at some additional cost.

5. We recommend designing new hash function using this framework
6. It may be discussed whether all or only some of the suggestions should be applied.
7. We recommend that functions that do not follow these guidelines (e.g., all existing ones) will be discontinued in the long term.
8. [Some of these changes may be appropriate for block ciphers as well].

Summary (cont.)

Type of Attack	Ideal Hash Function $=$	MD $\geq$	HAIFA fixed salt $\geq$	HAIFA with (distinct) salts $\geq$
Preimage	2^{m_c}	2^{m_c}	2^{m_c}	2^{m_c}
One-of-many pre-image (k' targets)	$2^{m_c}/k'$	$2^{m_c}/k'$	$2^{m_c}/k'$	2^{m_c}
Second-pre-image (k blocks)	2^{m_c}	$2^{m_c}/k$	2^{m_c}	2^{m_c}
One-of-many second pre-image(k blocks in total, k' messages)	$2^{m_c}/k'$	$2^{m_c}/(kk')$	$2^{m_c}/k'$	2^{m_c}
Collision	$2^{m_c/2}$	$2^{m_c/2}$	$2^{m_c/2}$	$2^{m_c/2}$
Multi-collision (k-collision)	$2^{m_c(k-1)/k}$	$\lceil \log_2 k \rceil 2^{m_c/2}$	$\lceil \log_2 k \rceil 2^{m_c/2}$	$\lceil \log_2 k \rceil 2^{m_c/2}$
Herding	–	Offline: $2^{m_c/2+t}$ Online: 2^{m_c-t}	Offline: $2^{m_c/2+t}$ Online: 2^{m_c-t}	Offline: $2^{m_c/2+t+s}$ Online: 2^{m_c-t}

The figures are given for MD and HAIFA hash functions that use an ideal compression function.

Questions?