A Framework for Iterative Hash Functions — HAIFA*

Eli Biham[1]** Orr Dunkelman[2]***

[1] Computer Science Department, Technion.
Haifa 32000, Israel
biham@cs.technion.ac.il
[2] Katholieke Universiteit Leuven
Department of Electrical Engineering ESAT/SCD-COSIC
Kasteelpark Arenberg 10, B-3001 Leuven-Heverlee, Belgium
orr.dunkelman@esat.kuleuven.be

Abstract. Since the seminal works of Merkle and Damgård on the iteration of compression functions, hash functions were built from compression functions using the Merkle-Damgård construction. Recently, several flaws in this construction were identified, allowing for second pre-image attacks and chosen target pre-image attacks on such hash functions even when the underlying compression functions are secure.
In this paper we propose the HAsh Iterative FrAmework (HAIFA). Our framework can fix many of the flaws while supporting several additional properties such as defining families of hash functions and supporting variable hash size. HAIFA allows for an online computation of the hash function in one pass with a fixed amount of memory independently of the size of the message.
Besides our proposal, the recent attacks initiated research on the way compression functions are to be iterated. We show that most recent proposals such as randomized hashing, the enveloped Merkle-Damgård, and the RMC and ROX modes can be all be instantiated as part of the HAsh Iterative FrAmework (HAIFA).

Keywords: Merkle-Damgård, randomized hashing, Enveloped Merkle-Damgård, RMC, ROX, Wide pipe, HAIFA

1 Introduction

Cryptographic hash functions play an increasingly important role in cryptography. Many primitives and protocols rely on the existence of secure cryptographic

* An initial version of this work was presented in the hash function workshop in Krakow, June 2005 and in the second NIST hash function workshop 2006 in Santa Barbara, August 2006.

** This work was supported in part by the Israel MOD Research and Technology Unit.
*** This work was supported in part by the Concerted Research Action (GOA) Ambiorics 2005/11 of the Flemish Government and by the IAP Programme P6/26 BCRYPT of the Belgian State (Belgian Science Policy).

hash functions. Hash functions are usually constructed by means of iterating a cryptographic compression function, while trying to maintain the following three requirements:

1. Pre-image resistance: Given $y = H(x)$ it is hard to find x' s.t. $H(x') = y$.
2. Second pre-image resistance: Given x it is hard to find x' s.t. $H(x) = H(x')$.
3. Collision resistance: It is hard to find x, x' s.t. $H(x) = H(x')$.

The most widely used mode of iteration is the Merkle-Damgård construction [9, 17, 18]. The simple iteration method maintains the collision resistance of the compression function. The pre-image and second pre-image resistance of the compression function were also thought to be preserved in the Merkle-Damgård construction. However, counter examples for these beliefs were suggested recently.

The first evidence for this was by Dean [10] who showed that fix-points of the compression function can be used for a second pre-image attacks against long messages using $O(m \cdot 2^{m/2})$ time and $O(m \cdot 2^{m/2})$ memory (where m is the digest size). Later, Kelsey and Schneier have proposed the same ideas, while removing the assumption that fix-points can easily be found [15]. This improvement was achieved using Joux's multi-collision attack on iterated hash functions [13].

The previous attacks have used very long messages. This led Kelsey and Kohno to show that using a simple pre-computation it is possible to reduce the time requirements of chosen target pre-image attacks[1] for relatively short messages [14]. The total time complexity of the attack is much below the expected $O(2^m)$.

In this work we suggest the HAsh Iterative FrAmework (HAIFA) to replace the Merkle-Damgård construction. HAIFA maintains the good properties of the Merkle-Damgård construction while adding to the security of the transformation, as well as to the scalability of the transformation.

HAIFA has several attractive properties: simplicity, maintaining the collision resistance of the compressions function, increasing the security of iterative hash functions against (second) pre-image attacks, and the prevention of easy-to-use fix-points of the compression function. HAIFA also supports variable hash size and has a built-in support for defining families of hash functions as part of the framework. HAIFA also posses the online hashing property of the Merkle-Damgård construction. The computation of a HAIFA hash function requires one pass on the message, without keeping the entire message in memory, and while using a fixed amount of memory for the hashing of each block.

Along with recent advances in finding collisions on wide spread hash functions from the MD family [5, 6, 21–24], this motivated many suggestions to strengthen hash functions and modes of iteration. These suggestions are either aimed at reducing the security requirements from the compression function [11] or at proposing a mechanism to securely iterate a compression function [1–3, 7].

[1] The herding attack can be deployed in the following scenario: The attacker publishes in advance a digest value. Then, given a previously unknown message, the attacker finds a pre-image to the digest value that contains the unknown message.

The *randomized hashing* scheme [11] proposed by Krawczyk and Halevi aims to reduce the requirements on the collision resistance of the compression function in a collision resistant hash function. By randomizing the actual inputs to the compression functions, the existence of a collision in the compression function can be masked. This change is mostly useful for digital signatures (preventing the attack scenario where the attacker finds two colliding messages and asks the victim to sign the first).

The *enveloped Merkle-Damgård* construction [3] was proposed by Bellare and Ristenpart as a method to maintain the collision resistance, the pseudorandom and the pseudorandom family properties of the compression function. This is very useful for constructions which require the pseudorandom properties of the hash function, e.g., in cases where the hash function is used in MACs.

The last recent proposals for modes of iteration are the *RMC* [1] and *ROX* [2] by Andreeva et al. These two modes aim at preserving the collision resistance of the compression function, along with the second pre-image resistance (Sec) and the pre-image resistance (Pre), and their everywhere and always variants, aSec, eSec, aPre, and ePre.

Besides suggesting modes of iteration for the compression functions, recent research also suggests using larger internal state [16]. The approach, named *wide pipe*, mitigate the flaws of iterated hashing by using a larger internal state than the output size. This approach leads to the fact that internal collisions, i.e., collisions in the chaining value, are eventually as hard as finding a pre-image of the hash function itself (assuming a good compression function is being used).

After presenting HAIFA, we show that HAIFA can be used to instantiate any of these modes. Thus, a HAIFA compression function can easily be made to follow each of these suggestions according to the properties sought by the designer. For example, as a ROX hash function can be instantiated using the HAIFA framework, it is possible to construct a HAIFA hash function that maintains the (a/e)Sec and (a/e)Pre properties of the compression function.

This paper is organized as follows: In Section 2 we describe the Merkle-Damgård construction and various results regarding the construction. In Section 3 we propose HAIFA. We discuss the security aspects of HAIFA in Section 4. We show how to implement the randomized hashing scheme, the enveloped Merkle-Damgård, and the RMC and ROX constructions using a HAIFA hash function in Section 5. We compare the above constructions with HAIFA in Section 6. Finally, Section 7 summarizes the paper.

2 The Merkle-Damgård Constructions and its Pitfalls

The Merkle-Damgård construction is a simple and elegant way to transform a compression function $C_{MD} : \{0,1\}^{m_c} \times \{0,1\}^n \rightarrow \{0,1\}^{m_c}$ into a hash function [9, 17, 18]. Throughout this paper m_c denotes the size of the chaining value, and n denotes the block size for the compression function. We also denote by m the hash output length (in many cases $m = m_c$).

The message M is padded with its length (after additional padding to make the message a multiple of the block size n after the final padding), and the message is divided into blocks of n bits each, $M = M_1 M_2 M_3 \dots M_k$. An initial chaining value $h_0 = IV \in \{0,1\}^{m_c}$ is set for the hash function (also called initialization vector) and the following process is repeated k times:

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

The final h_k is outputted as the hash value, i.e., $H(M) = h_k$.

It is easy to prove that once a collision in the hash function $H(\cdot)$ is found, then a collision of the compression function $C_{MD}(\cdot)$ is found as well [9, 17, 18]. Thus, the Merkle-Damgård construction retains the collision resistance of the compression function.

When from $h_i = C_{MD}(h_{i-1}, M_i)$ and M_i the value of h_{i-1} can be easily computed, a pre-image attack on $H(\cdot)$ can be mounted using a birthday attack [25]. However, the opposite statement is not true. Even if an inversion attack on $C_{MD}(\cdot)$ requires $O(2^m)$ operations, the security claims for the hash function $H(\cdot)$ cannot offer security better than $O(2^{m/2})$. This surprising property was first noted by Dean [10], which went unnoticed until rediscovered (and expanded) by Kelsey and Schneier [15].

2.1 Fixed Points, Expandable Messages, and Finding Second Pre-Images

It was widely believed by the cryptographic community that the security proof of the Merkle-Damgård construction applies also to second pre-image attacks. However, Dean [10] noticed that this is not true for long messages if the compression function has easy to find fix-points. His observations were later generalized by Kelsey and Schneier [15] that used the multi-collision technique to eliminate the need for easily found fix-points.

Let us consider a long message $M = M_1 M_2 \dots M_l$ that is processed using $H(\cdot)$, a Merkle-Damgård hash function, when the message length is not padded (the Merkle-Damgård strengthening). An attacker that wishes to construct a message M^* such that $H(M^*) = H(M)$ can randomly select messages M' until $H(M')$ equals to any of the l chaining values found during the computation of $H(M)$. Once such an instance is found, the attacker can concatenate to M' the message blocks of M that are hashed starting from the given chaining value, resulting with M^* such that $H(M^*) = H(M)$. This attack is foiled by the Merkle-Damgård strengthening, as the message length which is appended to the message is expected to differ for M and M^*.

Assume that the compression function C_{MD} is such that finding fix-points is easy, i.e., it is easy to find (h, M) satisfying $h = C_{MD}(h, M)$. This is the case for the Davies-Meyer construction that takes a block cipher E that accept m_c-bit plaintexts and n-bit keys and sets

$$h_i = C_{MD}(h_{i-1}, M_i) = E_{M_i}(h_{i-1}) \oplus h_{i-1}.$$

For such a compression function it is easy to find fix-points by computing $h = E_M^{-1}(0)$ for randomly selected messages M.

Dean uses these fix-points to bypass the Merkle-Damgård strengthening. His attack has three main steps:

1. Finding $O(2^{m_c/2})$ fix-points denoted by $A = (h, m)$.
2. Selecting $O(2^{m_c/2})$ single blocks and computing their chaining value denoted by $B = (C_{MD}(IV, \tilde{m}), \tilde{m})$.
3. Finding a collision between a chaining value and a fixed point, i.e., between chaining values in A and in B. Let the colliding chaining be h, and denote the corresponding message block (found in A) by m, i.e., $(h, m) \in A$, and denote the message block associated with h in B by $\tilde{m}$, i.e., $(h, \tilde{m}) \in B$. The attacker repeats the previous attack starting from the message $\tilde{m}||m$ (i.e., trying to add blocks that cause the same chaining values as the original message) and fixes the length by iterating the fixed point as many times as needed.

Once such a message is found, it is easy to expand the number of blocks in the message to the appropriate length by repeating the fix-points as many times as needed.

Kelsey and Schneier transformed the attack to the case where fix-points are not easily found. While Dean's expandable message could be extended by a repetition of a single block, in their attack they use the multi-collision technique to produce an expandable message. They replace the first two steps in Dean's attack in the following procedure. In each iteration $1 \leq i \leq t$ of the procedure a collision between a one block message and a $2^{i-1} + 1$ block message is found. This procedure finds a chaining value that can be reached by messages of lengths between t and $2^{t+1} + t - 1$ blocks. Then, from this chaining value the third step of Dean's attack is executed, and the length of the found message is controlled by the expandable prefix.

2.2 Multi-Collisions in Iterative Hash Functions

Joux identified the fact that when iterative hash functions are used, finding multi-collisions, i.e., a set of messages with the same hash value, is almost as easy as finding a single collision [13]. His main observation is the fact that in iterative hashing schemes, such as the Merkle-Damgård, it impossible to find a collision for each block, e.g., for any h_{i-1} finding M_i and M_i^* such that $C_{MD}(h_{i-1}, M_i) = C_{MD}(h_{i-1}, M_i^*)$. Finding t such one block collision (each starting from the chaining value produced by the previous block collision) it is possible to construct 2^t messages with the same hash value by selecting for ith block of the message either M_i or M_i^*.

Joux also observed that the concatenation of two hash functions, i.e., $H(x) = H_1(x)||H_2(x)$, is not more secure against collision attacks than the stronger of the two underlying hash functions. Moreover, concatenation of several iterative hash functions is as secure as the stronger of the hash functions (up to some a factor of m^{k-1}, where k is the number of hash functions).

It is worth mentioning that using fix-points of several blocks, Joux proved that the concatenation of hash functions is as secure against pre-image attacks as the strongest of all the hash functions. These results have disproved several widely believed assumptions on the behavior of hash functions.

2.3 Herding Iterative Hash Functions

Kelsey and Kohno have observed that it is possible to perform a time-memory tradeoff for several instances of pre-image attacks [14]. In their attack, an attacker commits to a public digest value that corresponds to some meaningful message, e.g., prediction of the outcome of NIST's hash function competition. After the announcement of the result, the attacker publishes a message that has the pre-published digest value and contains the correct information along with some suffix.

The attack is based on selecting the digest value carefully, helping the attacker to perform a pre-image attack on this value. In the pre-computation phase, the attacker starts with 2^t possible chaining values h_i (these values can either be randomly selected or fixed in advance). The attacker then chooses $O(2^{m_c/2-t/2})$ single blocks, and for each chaining value and each block computes the output of the compression function given this input. The large amount of generated chaining values is expected to generate collisions. More precisely, it is expected that for each starting chaining value there is another chaining value, such that they are compressed to the same chaining value (not necessarily under the same message block). For each chaining value the attacker stores the message block that causes a collision in the table, and repeats the above process with the newly found chaining values. Once the attacker has only one chaining value, it is used to compute the digest value to be published (maybe after padding or some other message extension).

In the online phase of the attack, the attacker needs to perform only $O(2^{m_c-t})$ operations until a message whose chaining value is among the 2^t original values is found. Once such a message is found, the attacker can retrieve from the stored date the message blocks that would lead to the designated digest.

We note that unlike the previous attacks that require long messages, this attack appends relatively short suffix (about t blocks) to the "real" message. We also note, that the total time complexity of the attack is about $O(2^{m_c/2+t/2})$ off-line operations for the first step of the attack and $O(2^{m_c-t})$ online operations for the second step. For $t = m_c/3$ the overall time complexity of this attack is $O(2^{2m_c/3})$ for finding a pre-image.

3 The HAsh Iterative FrAmework (HAIFA)

We propose the HAsh Iterative FrAmework to solve many of the pitfalls of the Merkle-Damgård construction. The main ideas behind HAIFA are the introduction of number of bits that were hashed so far and a salt value into the compression functions. Formally, instead of using a compression function

of the form $C_{MD} : \{0,1\}^{m_c} \times \{0,1\}^n \rightarrow \{0,1\}^{m_c}$, we propose to use $C : \{0,1\}^{m_c} \times \{0,1\}^n \times \{0,1\}^b \times \{0,1\}^s \rightarrow \{0,1\}^{m_c}$, i.e., in HAIFA the chaining value h_i is computed as

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

where $\#bits$ is the number of bits hashed so far and $salt$ is a salt value.

Thus, to hash a message M using $C(\cdot)$ and the salt $salt$ and obtaining m bits of digest value (as long as $m \leq m_c$), the following operations are performed:

1. Pad M according to the padding scheme described in Section 3.1.
2. Compute IV_m the initial value for a digest of size m using the prescribed way in Section 3.2.
3. Iteratively digest the padded message using $C(\cdot)$, starting from the initial value IV_m and using the salt. We note that in case an additional block is padded to the message, the compression function is called on this block with $\#bits = 0$.
4. Truncate the final chaining value if needed (see Section 3.2).

3.1 The Padding Scheme

The padding scheme used in HAIFA is very similar to the one used in the Merkle-Damgård construction: In HAIFA the message is padded with 1, as many needed 0's, the length of the message encoded in a fixed number of bits, and the digest size:

1. Pad a single bit of 1.
2. Pad as many 0 bits as needed such that the length of the padded message (with the 1 bit and the 0's) is congruent modulo n to $(n - (t + r))$.
3. Pad the message length encoded in t bits.
4. Pad the digest size encoded in r bits.

Adding the digest size ensures that even if two messages M_1 and M_2 are found, such that under IV_{l_1} and IV_{l_2} (M_1 hashed to obtain l_1 bits and M_2 hashed to a digest of l_2 bits) their chaining values collide, then the last block changes this behavior. This approach is similar to the one used in the Merkle-Damgård strengthening, even though it deals with a scenario of variable output sizes.

We note that when a full padding block is added for the hashing (i.e., the full original message was already processed by the previous calls to the compression function, and the full message size was already input to the previous call), the compression function is called with the number of bits hashed so far set to zero. This allows for the compression function to identify whether this is the last block and moreover, whether this is a full padding block. It is easy to see that when the number of bits hashed so far is not a multiple of the block length, then this is certainly the last block (and the question whether there is a full padding block can be easily deduced from the length of the message). We note that for the null

string there is a padding of a full padding block anyway. Appendix A contains the exact algorithm for finding the last block of the message.

The security is not affected by this method, as the length of the message is necessarily embedded in the full padding block, and was already input to the previous call to the compression function.

3.2 Variable Hash Size

Different digest sizes are needed for different applications. This fact has motivated NIST to publish SHA-224 and SHA-384 as truncated variants of SHA-256 and SHA-512, respectively. We note that these truncated hash functions use the same construction, but with different initial values.

Our framework supports truncation that allows arbitrary digest sizes (up to the output size of the compression function), while securing the construction against attacks that try to find two messages that have similar digest values. This problem eliminates the easy solution of just taking the number of output bits from the output of the compression function.

Let IV be an initial value chosen by the designer of the compression function, and let m be the required length of the output. For producing hash values of m bits the following initial value is computed[2]

$$IV_m = C(IV, m, 0, 0).$$

The value m is encoded in the first r bits, followed by a single bit 1, and $n-r-1$ 0's. In other words, m is found in the first r bits of the block. We note that even though $\#bits = 0$, this call is distinct from all other calls to the compression function. This is due to the fact that the only other call to the compression function with $\#bits = 0$ is in the case of using an additional padding block, for which the message block is either all 0's or a 1 followed by as many 0's as needed, whereas in this case the message block contains the encoding of m.

After the final block is processed, the digest is composed of the m first bits of h_t, where h_t is the last chaining value. We advise implementors to make sure that these m bits are the most diffused bits of the chaining value.

This suggestion allows for supporting various digest lengths in a simple and straightforward way. An implementation of a HAIFA-based hash function requires only the value of IV for the ability to produce any hash length, while in implementations where only a single output length l is needed, IV_l can be precomputed and hardcoded into the implementation.

4 The Security of HAIFA Hash Functions

We first note that proving the HAIFA hash function is collision resistant if the underlying compression function is collision resistant is quite easy. The same arguments that are used to prove that the Merkle-Damgård construction retains

[2] We alert the reader that in [7] a typo suggested that $IV_m = C(m, IV, 0, 0)$.

the collision resistance of the underlying compression function, can be used to prove that HAIFA does so as well. We consider the strongest definition of a collision in the compression function where the attacker has a control over all input parameters to the compression function and tries to generate the same output. Let M_1 and M_2 be the two colliding messages with respective lengths l_1 and l_2. If l_1 and l_2 are different, then the paddings are necessarily different (due to the different length), and a collision in the compression function is found. If the lengths of the messages are the same, one can start from the joint digest and trace backwards till the point where the inputs to the compression function differ. For example, if the messages are hashed with a different salt, then the last block is necessarily a collision. Otherwise, both messages were hashed with the same salt and have the same length (and thus have the same $\#bits$ in each call to the compression function), and the same argument as the one for the Merkle-Damgård mode shows that there must exist a message block i such that $M_i^1 \neq M_i^2$ or $h_{i-1}^1 \neq h_{i-1}^2$ (where the superscript denotes the corresponding message), for which $C(h_{i-1}^1, M_1^i, s, i \cdot n) = C(h_{i-1}^2, M_2^i, s, i \cdot n)$.

We note that any iterative construction can be attacked by some of the attacks described in Section 2. However, as noted before, using our ideas, it is possible to reduce the applicability of these attacks, by preventing an efficient pre-computation that reduces the online computational phase of these attacks.

We continue by first discussing some of the security reasons behind the added parameters to the compression function. Then we analyze the general security feature of a HAIFA hash function.

4.1 Number of Bits Hashed so Far

The inclusion of the number of bits hashed so far was suggested (with small variants) in order to prevent the easy exploitation of fix-points. The attacker is forced to work harder in order to find fix-points. While for C_{MD}, once a fix-point (h, M) such that $h = C_{MD}(h, M)$ is found, it can be used as many times as the attacker sees fit [10, 15]. Even if the compression function does not mix the $\#bits$ parameter well, once an attacker finds a fix-point of the form $(h, M, \#bits, salt)$ such that $h = C(h, M, \#bits, salt)$, she cannot concatenate it to itself as many times as she wishes because the number of bits hashed so far has changed.

We note that it is possible to use the number of blocks that were treated so far instead. However, current schemes keep track of the number of bits hashed so far which is used for for the padding, rather than the number of blocks. Thus, it is easier for implementations to consider only one parameter (number of bits) rather two (somewhat related) parameters (number of bits and number of blocks).

It is interesting to consider message authentication codes based on the following HAIFA hash function $H(\cdot)$: $MAC_k(M) = H(k, M)$. While for a Merkle-Damgård construction or suggestions that use the number of blocks hashed so far, this suggestion is clearly not secure against message expansion techniques, for HAIFA this construction is secure. The reason for that is that the last block (or the one before it, in case an additional padding block is added) is compressed

with the number of bits that were processed so far. If this value is not a multiple of a block, then the resulting digest does not equal the chaining value that is needed to the expansion of the message. If the message is a multiple of a block, then an additional block is hashed (with the parameter $\#bits$ set to 0). Thus, the chaining value required for the expansion remains obscure to the attacker.

4.2 Salt

The *salt* parameter can be considered as defining a family of hash functions as needed by the formal definitions of [19] in order to ensure the security of the family of hash functions. This parameter can be viewed as an instance of the randomized hashing concept, thus, inheriting all the "goodies" such concept provides:

- Ability to define the security of the hash function in the theoretical model.
- Transformation of all attacks on the hash function that can use precomputation from an off-line part and an on-line part to only on-line parts (as the exact *salt* is not known in advance).
- Increasing the security of digital signatures, as the signer chooses the *salt* value, and thus, any attack aiming at finding two messages with the same hash value has to take the *salt* into consideration.

We note that the salt can be application specific (e.g., a string identifying the application), a serial number that follows the application (e.g., the serial number of the message signed), a counter, or a random string. It is obvious that s can also be set as a combination of these values.

4.3 Security against the Multi-Collision Attack

Let us consider the multi-collision attack. This attack works against all iterative hashing schemes, independent of their structure. While the time complexity for finding collisions for each block is not different in our framework than in the Merkle-Damgård construction, an attacker cannot pre-compute these multi-collisions before the choosing of the salt value.

4.4 Preventing Attacks Based on Fix-Points

As noted before, our framework prevents Dean's attack, as it is highly unlikely that some fix-point of the compression function can be repeated. We note that the existence of an additional random input is not sufficient to ensure security against this kind of attacks. For example, when considering the randomized hashing modes proposed in [11] it is evident that this attack still applies to them. Dean's attack can be easily applied to the randomized Merkle-Damgård constructions $H_r(M) = H(M \oplus (r|r|r \ldots |r))$ and $\tilde{H}_r(M) = H_r(0|M)$, as once a fix-point is found, it remains a fix-point for these constructions. We note that the first two steps of Dean's attack can be mounted off-line just as in Dean's attack on regular Merkle-Damgård hash functions for these two constructions.

As for Kelsey and Schneier's attack, just like in Joux's multi-collision attack, the attacker has to know the value of the salt before being able to generate the expandable message. Thus, an attacker who tries to generate a second pre-image, has to wait till the original message and salt are provided (or generate an expandable message for each and every possible salt).

We conclude by noting that like in Joux's multi-collision attack, once the attacker is given the salt, the attacker can repeat the Kelsey and Schneier attack.

4.5 Mitigating the Herding Attack

Under HAIFA, the precomputation phase of the herding attack is infeasible without the knowledge of the salt that is used. We note that if a security of $O(2^m)$ against pre-image attacks such as the herding attack is requested, then the size of the salt must be at least $m_c/2$ bits long, in order to prevent the herding attack. We also note that when the attacker is the one choosing the salt, then the herding attack cannot be avoided, but due to the $\#bits$ parameters, the attacker can mount the attack starting from one specific point (rather than moving the diamond structure around and compensating using an expandable message after the diamond structure).

We note that if the length of the salt is short, then an attacker can still use precomputation to overcome the security proposed by HAIFA. It is therefore recommended that the salt length would be of 64 bits at least, or at least $m_c/2$ bits when possible.

4.6 Final Notes about the Security of HAIFA

The approach of increasing the chaining value was promoted in [16] and it may seem that our suggestion supports this approach. However, the analysis in [16] assumes that the hash function is a "good" hash function for all the bits of the chaining value, and the compression function does not accept the additional inputs that HAIFA support. Therefore, a large increase in m_c is needed (typically, doubling the size). In our approach the *salt* and $\#bits$ parameters are treated separately from the chaining value, and allow us to protect from the weakness of a short chaining value without doubling the size of the chaining values. Thus, it is expected that HAIFA hash functions will be faster than wide hash constructions.

We conclude that the security of a Merkle-Damgård hash function against a pre-image attacks is equivalent to its security against collision attacks. For HAIFA this is not the case, as we have shown earlier. We give the security level of an ideal hash function and of the Merkle-Damgård and HAIFA constructions (under two cases — with a variable salt, and with a fixed salt) in Table 1.

5 Modeling Other Constructions in HAIFA

In this section we show that recent proposals for modes of iteration can be viewed as an instance of HAIFA. This shows that the interface proposed by HAIFA is robust enough for any of the possible proposals for modes of iteration.

Type of Attack	Ideal Hash Function	MD	HAIFA fixed salt	HAIFA with (distinct) salts
	$=$	$\geq$	$\geq$	$\geq$
Preimage	2^{m_c}	2^{m_c}	2^{m_c}	2^{m_c}
One-of-many pre-image ($k' < 2^s$ 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, $k' < 2^s$ messages)	$2^{m_c}/k'$	$2^{m_c}/k$	$2^{m_c}/k'$	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 [14]	–	Offline: $2^{m_c/2+t/2}$ Online: 2^{m_c-t}	Offline: $2^{m_c/2+t/2}$ Online: 2^{m_c-t}	Offline: $2^{m_c/2+t/2+s}$ Online: 2^{m_c-t}

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

Table 1. Complexities of Attacks on Merkle-Damgård and HAIFA Hash Functions with Comparison for an Ideal Hash Function

We note that in these constructions the padding schemes are not compatible with the added digest size to the padding. This can be solved easily during the identification of the last block by removing it in the compression function. We also note that in the following constructions, HAIFA's padding scheme does not affect any of the security properties of the hash functions.

5.1 Randomized Hashing

The main purpose of randomized hashing is to reduce the level of requirements from the compression function in order to achieve a collision-resistant hash function [11]. The randomized hashing is especially useful for digital signatures, where the collision resistance is the most important requirement from the hash function. In order to achieve these properties the following two constructions were suggested:

$$H_r(M_1||M_2||\dots||M_k) \stackrel{def}{=} H(M_1 \oplus r||M_2 \oplus r||\dots||M_k \oplus r)$$
$$\tilde{H}_r(M) \stackrel{def}{=} H_r(0||M) = H(r||M_1 \oplus r||M_2 \oplus r||\dots||M_k \oplus r)$$

where H is a Merkle-Damgård hash function.

It is easy to see that by setting the salt to $s = r$ and ignoring the input of number of bits, we can instantiate a randomized hashing scheme in a HAIFA compression function. Let C_{MD} be the compression function used in the randomized hashing, thus we set:

$$C_{HAIFA1}(h_{i-1}, M_i, \#bits, s) = C_{MD}(h_{i-1}, M_i \oplus s),$$

which would result in an implementation of H_r, i.e., $HAIFA^{C_{HAIFA1}} = H_r^{C_{MD}}$.

To implement $\tilde{H}_r$, we use a slightly modified construction:

$$C_{HAIFA2}(h_{i-1}, M_i, \#bits, s) = \begin{cases} C_{MD}(C_{MD}(h_{i-1}, s), M_i \oplus s) & \text{If } 0 < \#\text{bits} \leq n \\ C_{MD}(h_{i-1}, M_i \oplus s) & \text{Otherwise} \end{cases}$$

It is easy to see that for any non-empty message, the first block is hashed only after the random value is hashed. For the empty message (null string) the computed hash is $C_{MD}(IV, s)$ which is the value that would have been computed for $\tilde{H}_r(\cdot)$. We conclude that $HAIFA^{C_{HAIFA2}} = \tilde{H}_r^{C_{MD}}$.

Note that in both cases, if the variable message length feature is not wanted it can be ignored by letting $h_i = C(h_{i-1}, M_i, \#bits, s) = h_{i-1}$ if the following conditions hold: $\#bits = 0$, the field that contains the digest size in the padding is set to 0.

5.2 Enveloped Merkle-Damgård

The Enveloped Merkle-Damgård [3] is a proposal for a mode of iteration that preserves the following three properties: collision resistance, pseudorandom oracle, and pseudorandom family. In order to achieve this, the Merkle-Damgård construction is altered in the following manner:

- The padding scheme pads the message with 1, as many 0's as requires, and the message length, such that the padded message, $PAD_{EMD}(M)$ has a length which equals to $n - m_c \bmod n$.
- The final digest value is computed as

  $$h_k = C_{MD}(IV_2, h_{k-1}||M_k),$$

  for a second initialization vector IV_2.

Let C_{MD} be the compression function used for the enveloped Merkle-Damgård, and let C_{HAIFA3} be defined as:

$$C_{HAIFA3}(h_{i-1}, M_i, \#bits, s) = \begin{cases} C_{MD}(IV_2, h_{i-1}||fix\_pad(M_i)) & \text{Last block} \\ C_{MD}(h_{i-1}, M_i) & \text{Otherwise} \end{cases}$$

As noted earlier, identifying the last block is a trivial operation in HAIFA. Thus, it is easy to see that $HAIFA^{C_{HAIFA3}} = EMD^{C_{MD}}$, i.e., the suggested HAIFA hash function is equivalent to the enveloped Merkle-Damgård one.

As the padding scheme used in the enveloped Merkle-Damgård scheme is slightly different than the one used in HAIFA. Fixing the padding in the last block $fix\_pad(\cdot)$ can be done as the HAIFA padding can be easily shortened. When the first m_c zero bits of the padding (assuming $m_c < n$) can be removed in the last block, then the operation is straightforward. If there are not sufficiently such bits, then a full padding block would have been needed in the enveloped Merkle-Damgård, and the generation of the correct full padding block can be done in the compression function.

5.3 RMC and ROX

The RMC [1] construction suggests a method to preserve seven properties of a compression function $C_{RMC} : \{0,1\}^{m_c} \times \{0,1\}^n \times \{0,1\}^k \rightarrow \{0,1\}^{m_c}$ in the hash function $RMC^{RO_1,RO_2} : \{0,1\}^k \times \{0,1\}^s \times \{0,1\}^* \rightarrow \{0,1\}^m$. The preserved properties are collision resistance, Second pre-image resistance, and pre-image resistance, along with their everywhere and always variants. The RMC construction maintains these properties by using the XOR-linear hash scheme [4]. The construction uses two random oracles (with fixed output length). The first random oracle RO_1 is used to produce strings which are XORed to the chaining values (just like in the XOR-linear scheme). The second random oracle RO_2 is used for the padding scheme.

Let $\nu(i)$ be the largest number j for which 2^j divides i. As in the XOR-linear construction, when hashing block i, the string $\mu_{\nu(i)}$ is XORed to the chaining value, unlike the XOR-linear scheme, this value is computed using a call to the random oracle RO_1. The parameters to the random oracle are the key of the hash function (corresponding to our salt) and a salt along with the encoding of $\nu(i)$, i.e., let the key be denoted by K and the salt by s, then

$$\mu_{\nu(i)} = RO_1(K, s, \nu(i)).$$

We note that the output of the random oracle is of length m_c bits.

The padding scheme is composed of as many calls to RO_2 that are needed to achieve the right length of padding (with some minimal length of padding), where in each call RO_2 is called with the key K, the salt s, and the serial call number. The padding scheme is out of the scope of our paper, but we shall denote it as $pad_{RMC}(K, s, M)$.

The RMC hash function that uses RO_1, RO_2 and C_{RMC} is the following scheme:

- Pad M using $RO_2(K, s, M)$, where K is the key (of length k bits) and s is the salt. Denote the padded message by $padded_{RMC}(M) = M_1||M_2||\ldots||M_t$.
- Set $h_0 = IV$.
- Generate $\mu_i = RO_1(K, s, i)$, for $i = 0, \ldots, \lceil \log_2(|M|) \rceil$.
- For $i = 1, \ldots, l$ compute $h_i = C_{RMC}(h_{i-1} \oplus \mu_{\nu(i)}, M_i, K)$.

To support the RMC mode we embed the salt and the key in the salt of HAIFA. Thus,

$$C_{HAIFA4}(h_{i-1}, M_i, \#bits, s) = \begin{cases} C_{RMC}(h_{i-1} \oplus RO_1(s_1, s_2, \nu(i)), M_i||pad_{RO2(s_1,s_2)}) & \text{Last block} \\ C_{RMC}(h_{i-1} \oplus RO_1(s_1, s_2, \nu(i)), M_i, s_1) & \text{Otherwise} \end{cases}$$

where $s = (s_1, s_2)$, i.e., the salt is composed of the concatenation of the key and the salt of RMC. We note that computing $\nu(i)$ in HAIFA is very easy, as it involves dividing the #bits parameter by n to obtain the block number, and then compute $\nu(i)$ as in RMC.

Given the above transformation, and assuming that both hash functions use the same random oracles, then $HAIFA^{C_{HAIFA4}} = RMC^{C_{RMC}}$.

In [2] the RMC construction was slightly improved, and some parameters were changed a little bit to suggest the ROX construction. Let C_{MD} be the compression function, then $ROX^{RO_1,RO_2}_{C_{MD}} : \{0,1\}^k \times \{0,1\}^* \rightarrow \{0,1\}^m$ is defined as:

- Pad M using $RO_2(K, M)$, where K is the key (of length k bits). Denote the padded message by $padded_{ROX}(M) = M_1||M_2||\ldots||M_t$.
- Set $h_0 = IV$.
- Generate $\mu_i = RO_1(K, \tilde{m}, i)$, for $i = 0, \ldots, \lceil \log_2(|M|) \rceil$, where $\tilde{m}$ is the first k bits of the message.
- For $i = 1, \ldots, l$ compute $h_i = C_{MD}(h_{i-1} \oplus \mu_{\nu(i)}, M_i)$.

We note that M replaces the salt s from RMC. We set the size of the salt to $2k$, where the first half of the salt is K, and the second half is m (which are known immediately once the first block is hashed).

Thus, $C_{HAIFA5} : \{0,1\}^m_c \times \{0,1\}^n \times \{0,1\}^b \times \{0,1\}^{2k}$ is as follows:

$$C_{HAIFA5}(h_{i-1}, M_i, \#bits, s) = \begin{cases} C_{MD}(h_{i-1} \oplus RO_1(s_1, s_2, \nu(i)), M_i||pad_{RO2(s_1)}) & \text{Last block} \\ C_{MD}(h_{i-1} \oplus RO_1(s_1, s_2, \nu(i)), M_i) & \text{Otherwise} \end{cases}$$

Where $s = (s_1, s_2)$, and $s_2 = first_k(M)$ (the value of s_1 is set by the user as well as part of the "key" to the hash function). It is easy to see that $HAIFA^{C_{HAIFA5}} = ROX^{C_{MD}}$ given the access to the "same" random oracles.

Another possible reduction is based on increasing the chaining value by k bits which are initialized to 0. When the first block is processed, these k bits are set to $first_k(M)$ (in the first block $0 < \#bits \leq n$), and then this value is not altered, while being transfered to all the following calls for the compression function.

5.4 Wide-Pipe Hash Constructions

The wide-pipe hash design principal was introduced by Lucks in [16] to reduce the effects of attacks based on internal collisions on the hash function. By increasing the chaining value size, e.g., choosing $m_c = 2m$ (in a double-pipe construction), the cost of the attacks based on internal collisions is at least $2^{m_c/2} = 2^m$, thus allowing for a 2^m security goal against pre-image and second pre-image attacks.

The scheme employs two compression functions. The first one is $C' : \{0,1\}^{m_c} \times \{0,1\}^n \rightarrow \{0,1\}^{m_c}$, while the second one is $C'' : \{0,1\}^{m_c} \rightarrow \{0,1\}^m$. The digest value is computed like in the Merkle-Damgård construction using $C'(\cdot)$ as the compression function. Then, the computed chaining value is compressed using one call to $C''(\cdot)$ to produce the digest.

It is easy to see that transforming the wide pipe strategy into a HAIFA construction is a straightforward (and very similar to the transformation used

in the EMD mode):

$$C_{HAIFA6}(h_{i-1}, M_i, \#bits, s) = \begin{cases} C''(C'(IV_2, h_{i-1}||fix\_pad(M_i))) & \text{Last block} \\ C'(h_{i-1}, M_i) & \text{Otherwise} \end{cases}$$

As noted earlier, identifying the last block is a trivial operation in HAIFA. Thus, it is easy to see that $HAIFA^{C_{HAIFA6}} = WidePipe^{C',C''}$, i.e., the suggested HAIFA hash function is equivalent to the wide pipe hash function.

6 Comparing Other Constructions with HAIFA

While other constructions propose different security assurances, they lack in the security against realistic and practical attacks. For example, Joux's multi-collision attack is applicable against EMD with the same ease it is applicable against Merkle-Damgård. Unlike HAIFA where the collisions are to be chosen after the salt is known, in EMD this can even be in an off-line manner.

For the randomized hashing scheme, Joux's multi-collision attack can be performed off-line, where the actual multi-collision is updated according to the random string afterwards, or can be performed in an on-line manner (once the random value is known). The long second pre-image attack can also be easily applied to the randomized hashing scheme, as the pre-processing of the messages is independent of the random string (which can later be XORed into the message to suggest the actual pre-image). The herding attack is also easily applicable to the randomized hashing scheme, as again, the only difference in the attack is the fact that resulting message is to be XORed with the random string (block by block). This problems are addressed in HAIFA by the stronger diffusion of the salt into the compression function, thus, eliminating the easy use of the pre-computed values.

The EMD scheme is mostly Merkle-Damgård, and thus, it is not surprising that all the mentioned attacks are applicable with the same ease to EMD. Thus, EMD proposes no additional security against any of these attacks. As noted earlier, the application of the multi-collision attack can be done off-line. It is quite obvious that security wise, HAIFA can offer a greater deal of security over the this mode.

The RMC and ROX transformation are as susceptible to multi-collision attacks as all the other iterated constructions. We note that just like in a general HAIFA hash function, the salt has to be known (transforming the attack to an on-line attack when this is relevant). Due to the keying, the long second pre-image attacks are no longer possible as the change of the key affects the compressed values in a non-predictable way. As for the herding attack, while the herding attack can always be applied against any iterated construction when the key is known. When the key is unknown, the attacker has to try all possible keys. Unlike a general HAIFA hash function, the attacker can still manipulate the location of the diamond structure, by using a sequence of $\mu(i)$ which repeats several times, e.g., 1,2,1,3,1,2,1,.... This allows for the shift of the diamond

Type of Attack	Randomized Hashing	EMD	RMC & ROX	Double Pipe	HAIFA
Multi-collision	App.	App.	App.*	Not App.	App.*
Long Second Preimage	App.	App.	Not App.	Not App.	Not App.
Herding	App.	App.	Partially App.	Not App.	Almost Not App.

* — When the salt is known
App. — Applicable

Table 2. Comparison of the Various Modes of Iteration

structure (when adding at the end of the diamond structure an expandable message to compensate for the different possible locations). Thus, RMC and ROX are almost as secure against the herding attack as HAIFA hash function can achieve.

While the wide-pipe offers better security against attacks based on internal collision, such a hash function requires an additional memory to achieve this security. In HAIFA, the only additional parameter which has to be stored is the salt (the $\#bits$ parameter is stored for the Merkle-Damgård strengthening anyway). Thus, to ensure the security of a HAIFA construction against the herding attack, a salt of size $m/2$ is sufficient, while to achieve the same goal in the wide-pipe strategy, one needs to increase the internal state by a factor of 2. Thus, while the internal memory of a wide-pipe is $2m$, in HAIFA a $1.5m$ bits of memory are sufficient.

Moreover, in the case of a salt, its effect on the compression function can be relatively mild and scarce (e.g., mixed every few rounds if the designer so chooses), while in the wide-pipe strategy, insufficient mixing of the additional state is a security hazard. Hence, the additional performance load and the more complex hash functions of the wide-pipe strategy might not be suitable for constrained applications.

In Table 2 we summarize these results.

7 Summary

In this paper we have presented HAIFA as a replacement for the Merkle-Damgård construction. The main differences are the addition of the number of bits hashed so far to the compression function along with a salt value. In cases where there is no need to add salt (e.g., message authentication codes) it is possible to set its value to 0.

We note that even today's compression functions can be used in HAIFA hash functions by changing the API of such compression functions. For example, by setting in SHA-1 64 bits (out of the 512 bits of each block) to represent the number of bits hashed so far, and 64 bits to represent the salt, the new compression function would hash 384 bits per call of the compression function.

This increases the computational effort of hashing long messages by a factor of about 4/3, but at the same time provides security against various attacks. New hash functions are expected to mix the salt and the number of bits much more efficiently.

We showed that the API of the HAIFA framework is sufficient to support recently proposed modes of operation. This shows that the interface suggested by HAIFA is sufficient for any modern hash function design, and thus we suggest that the new hash function designers should support the API of HAIFA. In case they wish to achieve some specific security property, they can choose a compression function with properties which make HAIFA meet the requirements.

Finally, we recommend that new hash functions should be designed under the HAsh Iterative FrAmework.

Acknowledgements

The authors wish to thank William Speirs and Praveen Gauravaram for pointing out typos in previous versions of this paper.

References

1. Elena Andreeva, Gregory Neven, Bart Preneel, Thomas Shrimpton, *Seven-Properties-Preserving Iterated Hashing: The RMC Construction*, ECRYPT document STVL4-KUL15-RMC-1.0, private communications, 2006.
2. Elena Andreeva, Gregory Neven, Bart Preneel, Thomas Shrimpton, *Seven-Properties-Preserving Iterated Hashing: ROX*, ECRYPT document STVL4-KUL15-ROX-2.0, private communications, 2007.
3. Mihir Bellare, Thomas Ristenpart, *Multi-Property-Preserving Hash Domain Extension: The EMD Transform*, NIST 2nd hash function workshop, Santa Barbara, August 2006.
4. Mihir Bellare, Philip Rogaway, *Collision-resistant hashing: Towards making UOWHFs practical*, dvances in Cryptology, proceedings of CRYPTO 1997, Lecture Notes in Computer Science 1294, pp. 470–484, Springer-Verlag, 1997.
5. Eli Biham, Rafi Chen, *Near-Collisions of SHA-0*, Advances in Cryptology, proceedings of CRYPTO 2004, Lecture Notes in Computer Science 3152, pp. 290–305, Springer-Verlag, 2004.
6. Eli Biham, Rafi Chen, Antoine Joux, Patrick Carribault, Christophe Lemuet, William Jalby, *Collisions of SHA-0 and Reduced SHA-1*, Advances in Cryptology, proceedings of EUROCRYPT 2005, Lecture Notes in Computer Science 3621, pp. 36–57, 2005.
7. Eli Biham, Orr Dunkelman, *A Framework for Iterative Hash Functions — HAIFA*, NIST 2nd hash function workshop, Santa Barbara, August 2006.
8. Florent Chabaud, Antoine Joux, *Differential Collisions in SHA-0*, Advances in Cryptology, proceedings of CRYPTO 1998, Lecture Notes in Computer Science 1462, pp. 56–71, Springer-Verlag, 1998.
9. Ivan Damgård, *A Design Principle for Hash Functions*, Advances in Cryptology, proceedings of CRYPTO 1989, Lecture Notes in Computer Science 435, pp. 416–427, Springer-Verlag, 1990.

10. Richared D. Dean, *Formal Aspects of Mobile Code Security.*, Ph.D. dissertation, Princeton University, 1999.
11. Shai Halevei, Hugo Krawczyk, *Strengthening Digital Signatures via Randomized Hashing*, Advances in Cryptology, proceedings of CRYPTO 2006, Lecture Notes in Computer Science 4117, pp. 41–59, Springer-Verlag, 2006.
12. Jonathan J. Hoch, Adi Shamir, *Breaking the ICE — Finding Multicollisions in Iterated Concatenated and Expanded (ICE) Hash Functions*, preproceedings of Fast Software Encryption 2006, pp. 199–214, 2006.
13. Antoine Joux, *Multicollisions in Iterated Hash Functions*, Advances in Cryptology, proceedings of CRYPTO 2004, Lecture Notes in Computer Science 3152, pp. 306–316, Springer-Verlag, 2004.
14. John Kelsey, Tadayoshi Kohno, *Herding Hash Functions and the Nostradamus Attack*, preproceedings of Cryptographic Hash Workshop, held in NIST, Gaithersburg, Maryland, 2005.
15. John Kelsey, Bruce Schneier, *Second Preimages on n-Bit Hash Functions for Much Less than* 2^n, Advances in Cryptology, proceedings of EUROCRYPT 2005, Lecture Notes in Computer Science 3494, pp. 474–490, Springer-Verlag, 2005.
16. Stefan Lucks, *A Failure-Friendly Design Principle for Hash Functions*, Advances in Cryptology, proceedings of ASIACRYPT 2005, Lecture Notes in Computer Science 3788, pp. 474–494, Springer-Verlag, 2005.
17. Ralph C. Merkle, *Secrecy, Authentication, and Public Key Systems*, UMI Research press, 1982.
18. Ralph C. Merkle, *One Way Hash Functions and DES*, Advances in Cryptology, proceedings of CRYPTO 1989, Lecture Notes in Computer Science 435, pp. 428–446, Springer-Verlag, 1990.
19. Phillip Rogaway, Thomas Shrimpton, *Cryptographic Hash-Function Basics: Definitions, Implications, and Separations for Preimage Resistance, Second-Preimage Resistance, and Collision Resistance*, proceedings of Fast Software Encryption 2004, Lecture Notes in Computer Science 3017, pp. 371–388, Springer-Verlag, 2004.
20. US National Bureau of Standards, *Secure Hash Standard*, Federal Information Processing Standards Publications No. 180-2, 2002.
21. Xiaoyun Wang, Xuejia Lai, Dengguo Feng, Hui Chen, Xiuyuan Yu, *Cryptanalysis of the Hash Functions MD4 and RIPEMD*, Advances in Cryptology, proceedings of EUROCRYPT 2005, Lecture Notes in Computer Science 3494, pp. 1–18, 2005.
22. Xiaoyun Wang, Yiqun Lisa Yin, Hongbo Yu, *Finding Collisions in the Full SHA-1*, Advances in Cryptology, proceedings of CRYPTO 2005, Lecture Notes in Computer Science 3621, pp. 17–36, 2005.
23. Xiaoyun Wang, Hongbo Yu, *How to Break MD5 and Other Hash Functions*, Advances in Cryptology, proceedings of EUROCRYPT 2005, Lecture Notes in Computer Science 3494, pp. 19–35, 2005.
24. Xiaoyun Wang, Hongbo Yu, Yiqun Lisa Yin, *Efficient Collision Search Attacks on SHA-0*, Advances in Cryptology, proceedings of CRYPTO 2005, Lecture Notes in Computer Science 3621, pp. 1–16, 2005.
25. Gideon Yuval, *How to Swindle Rabin*, Cryptologia, Vol. 3, pp. 187–190, 1979.

A Identifying the Last Block in HAIFA

Some of our reductions are based on the fact that the last block is easily identified by the compression function itself. This can be done by the compression function

by using the inputs in the following manner: In case $(\#bits \bmod n) < n-(t+r)$ and $\#bits > 0$, then this is the last message block, and no full padding block is added. Otherwise, there are two options: $\#bits \neq 0 \bmod n$, which means that a full padding block with $\#bits = 0$ is coming just after it, and $\#bits = 0 \bmod n$. The latter case covers the following possible events:

- This is an intermediate block of the message.
- This is a call to the compression function during the computation of IV_m.
- This is a call to the compression function for a full padding block.

The first case is easy to identify, by $\#bits > 0$. In this case, this is not the actual last block that is compressed.

For the remaining cases, $\#bits = 0$. If the call is during the computation of IV_m, then the last r bits of the message block are 0. Otherwise, i.e., this is a full padding block, it has one of two forms $10^{n-r-t-1}encode_t(length)encode_r(m)$ or $0^{n-r-t}encode_t(length)encode_r(m)$ where $length$ is the length of the message.

We conclude that all these options lead to the following algorithm to determine whether this is the last message block (or whether this is the computation of IV_m):

1. If $\#bits \neq 0 \bmod n$ return "Last block".
2. If $\#bits = 0$ and the r last bits of M_i are equal to 0, return "Computation of IV_m".
3. If $\#bits = 0$ and the r last bits of M_i are not equal to 0, return "Last block".
4. Return "Not the last block".