Cryptanalysis of a class of cryptographic hash functions

Praveen Gauravaram[1] and John Kelsey[2]

[1] Technical University of Denmark, Denmark
Information Security Institute, Australia
p.gauravaram@gmail.com
[2] National Institute of Standards and Technology, USA
john.kelsey@nist.gov

Abstract. We apply new cryptanalytical techniques to perform the generic multi-block multicollision, second preimage and herding attacks on the Damgård-Merkle hash functions with linear-XOR/additive checksums. The computational work required to perform these attacks on the Damgård-Merkle hash functions with linear-XOR/additive checksum of message blocks (GOST), intermediate states (**3C**, MAELSTROM-0, F-Hash) or both is only a little more than what is required on the Damgård-Merkle hash functions. Our generic attacks on GOST answers the open question of Hoch and Shamir at FSE 2006 on the security of the iterated hash functions with the linear mixing of message blocks.

Keywords: XOR-linear/additive checksums, **3C**, GOST, multi-block collision, multicollision, multi-block multicollision, 2nd preimage and herding attacks.

1 Introduction

The Damgård-Merkle construction [7, 29] provides a blueprint for building a cryptographic hash function, given a fixed-length input compression function; this blueprint is followed for nearly all widely-used hash functions. However, the past few years have seen two kinds of surprising results on hash functions, which have led to a flurry of research:

1. *Generic attacks* apply to the Damgård-Merkle construction directly, and make few or no assumptions about the compression function. These attacks involve attacking a t-bit hash function with more than $2^{t/2}$ work, in order to violate some property other than collision resistance. Examples of generic attacks are Joux multicollisions [15], long-message 2nd preimage attacks [8,18], and herding attacks [17].
2. *Cryptanalytic attacks* apply to the compression function of the hash function. However, turning an attack on the compression function into an attack on the whole hash function involves properties of the Damgård-Merkle construction. Examples of cryptanalytic attacks that involve the construction as well as the compression function include multi-block collisions on MD5 and SHA-1 [45, 46].

These results have stimulated interest in new constructions for hash functions, that prevent the generic attacks, provide some additional protection against cryptanalytic attacks or both. The recent call for submissions for a new hash function standard by NIST [33] has further stimulated interest in alternatives to Damgård-Merkle. Examples of recently-proposed alternative constructions include the **3C** construction [11–13], Haifa framework [4], Rivest's proposal to use squarefree sequences to prevent the long-message second preimage attack [40] and the RadioGatun hash proposal [6].

In this paper, we consider a family of variants of Damgård-Merkle, in which a linear-XOR/additive checksum is computed over the message blocks, intermediate states of the hash function, or both.

In a Damgård-Merkle hash with a linear-XOR checksum, each bit of the checksum is a XOR function of the bits of the message, intermediate states or both; the checksum is processed as a final block after the padding and length encoding of the original message have been processed. The **3C** construction [11–13] and the structure of the MAELSTROM-0 hash function [10] follow this pattern. F-Hash [23, 24] uses a XOR checksum of the outputs of the compression function alongside a normal Damgård-Merkle construction. Similarly, in a Damgård-Merkle hash with linear-additive checksums, an additive checksum computed using message blocks, intermediate states or both is processed as a final block. The 256-bit GOST hash function specified in the Russian standard GOST R 34.11 [35] uses an additive checksum mod 2^{256} computed using 256-bit message blocks. GOST, as a one-way and collision resistant hash function, is always used in conjunction with the Russian standard digital signature algorithms GOST R 34.10-94 and GOST R 34.10-2001 that are used to sign certificates and CRLs [26]. In addition, GOST has been specified for use in the cryptographic message syntax in [25] and with the HMAC algorithm [1, 2] for the purposes of message authentication and pseudorandom generation in [36].

The known cryptanalytical techniques of performing the generic attacks on the Damgård-Merkle hash functions are unsuccessful on these designs due to their much larger intermediate state in addition to the algebraic properties of the linear-XOR/additive operations used to compute checksums. Similarly, these designs make many cryptanalytic attacks on the compression function difficult or impossible to extend to attacks on the full hash function. For example, the inapplicability of the known long message second preimage attacks [8, 18] and herding attack [17] on the **3C** construction was shown in [11–13] and in [11] respectively. This applies to F-Hash, MAELSTROM-0, GOST and any hash function with these checksums. The designers of **3C** [11] and MAELSTROM-0 [10] claim that it takes 2^t computations of the hash function to find a (second) preimage on their designs with a t-bit hash value. GOST has been used as a one-way and collision resistant hash function in practice [25, 26, 36].

Unfortunately, all is not true. We first provide a general algorithm for attacking linear-XOR checksums of various kinds. Next, we provide another novel algorithm to defeat additive checksums in hash functions. Our attacks show that adding XOR/additive checksums to the Damgård-Merkle construction turns out to add almost no security against generic attacks. The linear-XOR/additive checksum may sometimes make it more difficult to use a cryptanalytic attack on the compression function to attack the full hash function, but this depends on the fine details of the linear checksum and cryptanalytic attack.

To summarize our results:

1. All known generic attacks on the Damgård-Merkle hash functions can be applied to linear-XOR/additive checksum variants of Damgård-Merkle at very little additional cost using our attacks that defeat these checksums. Thus, Joux multicollisions over multiple blocks, long-message second preimage attack, and herding attack all work just as well against the linear-XOR/additive checksum constructions as against Damgård-Merkle. Our attacks also work on the hash functions with additive checksums computed using some prime modulus.
2. The generic attacks on GOST answer the open question of Hoch and Shamir (FSE'06) [14] on the security of Damgård-Merkle hash functions with the linear mixing of message blocks. Our attacks also work on reasonably short CRCs computed over a message as shown in Section 5.4.
3. Many cryptanalytic collision attacks on the compression function, which the linear-XOR/additive checksum appears to stop from becoming attacks on the full hash function, can be carried out on the full hash function at relatively little additional cost.

4. From our techniques, it is possible to derive requirements on a checksum, if it is to improve security over that of Damgård-Merkle hashes.

1.1 Related Work

In unpublished work, Mironov and Narayan [30] developed a different technique to defeat linear-XOR checksums in hash functions; this technique is less flexible than ours, and does not work for long-message second preimage attacks. However, it is quite powerful, and can be combined with our technique in attacking hash functions with complicated checksums. We compare our technique with theirs in Appendix 7. In [15], Joux provides a technique for finding 2^k collisions for a Damgård-Merkle hash function for only about k times as much work as is required for a single collision. Multi-block collisions are an example of a cryptanalytic attack on a compression function, which must deal with the surrounding hash construction. Lucks [28] and Tuma and Joscak [42] have independently found that if there is a multi-block collision for a hash function with structured differences, concatenation of such a collision will produce a collision on **3C**, a specific hash construction which computes checksum using XOR operation as the mixing function. **3C** does not prevent Joux multicollision attack over 1-block messages [11–13].

Nandi and Stinson [32] have shown the applicability of multicollision attacks to a variant of Damgård-Merkle in which each message block is processed multiple times; Hoch and Shamir [14] extended the results of [32] showing that generalized sequential hash functions with any fixed repetition of message blocks do not resist multicollision attacks. The MD2 hash function [16] which uses a checksum computed using a XOR operation and non-linear S-box over the message was shown to be insecure [20,31]. The technique to solve a system of linear equations used in the cryptanalysis of hash functions with linear-XOR checksums presented in this paper have appeared in [3,5,43]. The approach of solving a system of linear equations has been employed to find collisions [37] and second preimages [22] for the SMASH hash function [19].

1.2 Impact

The main impact of our result is that new hash function constructions that incorporate linear-XOR/additive checksums as a defense against collision attacks and generic attacks do not provide much additional security. Designers who wish to thwart these attacks need to look elsewhere for techniques to do this. We can apply our techniques to specific hash functions and hashing constructions that have been proposed in the literature or are in practical use. They include **3C**, GOST, MAELSTROM-0 and F-Hash[1].

1.3 Guide to the Paper

This paper is organised as follows: First, we provide the descriptions of hash functions analysed in this paper. Next, we demonstrate cryptanlytical techniques to defeat XOR-linear/additive checksums in these designs. We then provide a generic algorithm to perform the 2nd preimage and herding attacks on the hash functions with linear checksums using the above cryptanalytical techniques with some illustrations. Finally, we demonstrate cryptanalytic multi-block collision attacks on the hash functions with linear checksums.

[1] Because our techniques require the ability to find collisions for the compression function, they do not represent a practical threat to applications using these systems at this time.

2 The Damgård-Merkle construction and the Damgård-Merkle hash with checksums

The Damgård-Merkle iterative structure [7, 29] shown in Figure 1 has been a popular framework used in the design of standard hash functions MD5 [39], SHA-1, SHA-224/256 and SHA-384/512 [9].

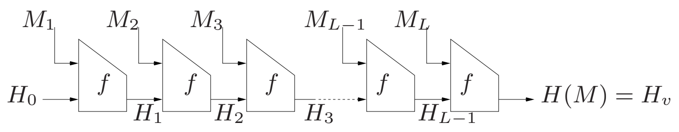


Fig. 1. The Damgård-Merkle construction

The message M, with $|M| \leq 2^l - 1$ bits, to be processed using a Damgård-Merkle hash function H is always padded by appending it with a 1 bit followed by 0 bits until the padded message is l bits short of a full block of b bits. The last l bits are filled in with the binary encoded representation of the length of the unpadded message M to avoid some trivial attacks [21]. This compound message is an integer multiple of b bits and is represented with b-bit data blocks as $M = M_1, M_2, \ldots M_L$. Each block M_i is processed using a fixed-length input compression function f as given by $H_i = f(H_{i-1}, M_i)$ where H_i from $i = 1$ to $L - 1$ are the intermediate states and H_0 is the initial state of H. The final state $H_v = f(H_{L-1}, M_L)$ is the hash value of M.

2.1 Linear-XOR/additive checksum variants of Damgård-Merkle

A number of variant constructions have been proposed, that augment the Damgård-Merkle construction by computing some kind of XOR-linear/additive checksum on the message bits and/or intermediate hash values, and providing the XOR-linear/additive checksum as a final block for the hash function as shown in Figure 2.

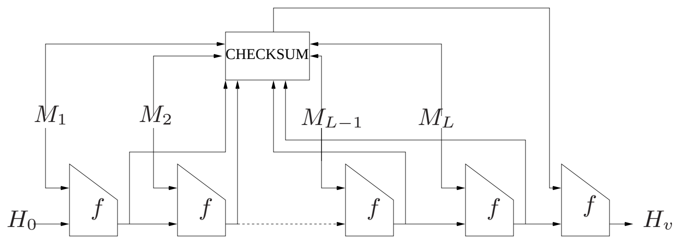


Fig. 2. Hash function structure with a linear-XOR/additive checksum

2.2 3C hash function and its XOR-linear checksum variants

The **3C** construction maintains twice the size of the hash value for its intermediate states using iterative and accumulation chains as shown in Figure 3. In it's iterative chain, the compression function f is iterated in the Damgård-Merkle mode. In it's accumulation chain, the checksum Z is computed by XORing all the intermediate states of the iterative chain. At any iteration i of f, the checksum value is $\bigoplus_{j=1}^{i} H_j$. The hash value H_v is computed by processing the checksum Z padded with 0 bits as the final data block $\overline{Z}$ using the last compression function f.

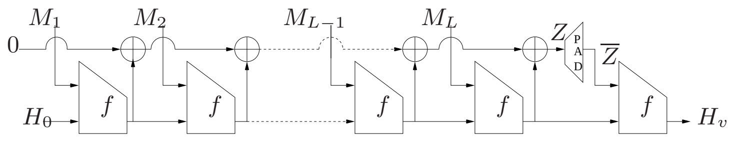


Fig. 3. The **3C**-hash function

A 3-chain variant of **3C** called **3CM** is used as a chaining scheme in the MAELSTROM-0 hash function [10]. At every iteration of f in the iterative chain of **3CM**, the t-bit value in the third chain is updated using an LFSR. This result is then XORed with the data in the iterative chain at that iteration. All the intermediate states in the iterative chain of **3CM** are XORed in it's second chain. Finally, the hash value is obtained by concatenating the data in the second and third chains and processing it using the last f function. F-Hash [23,24], another variant of **3C**, computes the hash value by XORing part of the output of the compression function at every iteration and then processes it as a checksum block using the last compression function. See Appendix A for the description of these variants of **3C**.

2.3 GOST and its additive checksum variants

GOST is a 256-bit hash function specified in the Russian standard GOSR R 34.11 [35]. The functionality of it's compression function f is derived from the block cipher GOST specified in the standard GOST R 34.10-89 [34]. The function f of GOST is iterated in the Damgård-Merkle mode and a mod 2^{256} additive checksum is computed by adding all the 256-bit message blocks in an accumulation chain. We generalise our analysis of GOST by assuming that it's f function has a block length of b bits and hash value of t bits.

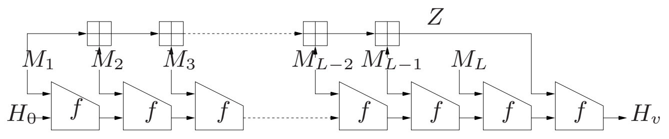


Fig. 4. GOST hash function

An arbitrary length message M to be processed using GOST is split into b-bit blocks $M_1, \ldots, M_{L-1}$. If the last block M_{L-1} is incomplete, it is padded by prepending it with 0 bits to make it a b-bit block. The binary encoded representation of the length of the true message M is processed in a separate block M_L as shown in Figure 4. At any iteration i, the intermediate state in the iterative and accumulation chains is $H_i = f(H_{i-1}, M_i)$ and $M_1 + M_2 \ldots + M_i \bmod 2^b$ respectively where $1 \leq i \leq L$. The hash value of M is $H_v = f(Z, H_L)$ where $Z = M_1 + M_2 \ldots + M_{L-1} \bmod 2^b$. An additive checksum variant of GOST called **3CA** which computes additive checksum using the intermediate states is discussed in Appendix D.

3 New techniques to defeat linear-XOR checksums in hash functions

The linear-XOR/additive checksums in the Damgård-Merkle hashes thwart the known techniques [8, 17, 18] of performing long message second preimage and herding attacks [11]. These designs make it difficult for the attacker to find an *expandable message*, an intermediate multicollision of different length messages, for all the chains simultaneously using the techniques of [8, 18] to find a second preimage. Even if the attacker is provided with an *expandable message* for free, he must still find a linking message block to produce states in all the chains that match the corresponding intermediate states of the long target message. This requires about 2^t computations of f. Similarly, the attacker must find a linking message block to map the state data in all the chains to the state somewhere in the precomputed diamond structure to make the herding attack work [17]. This takes about 2^t computations of f. While the Joux multicollision attack on 1-block messages work on these designs [11], the linear checksums in these designs prevent the technique of Joux to find multicollisions over multiple message blocks.

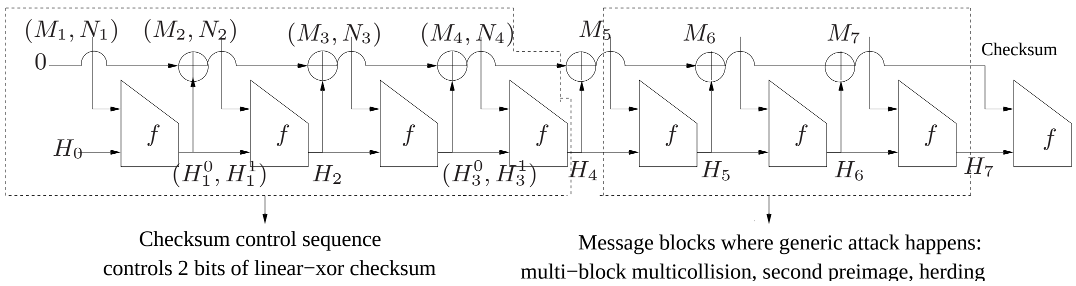
3.1 Extending Joux 1-block multicollision attack on DM to multiple blocks

Let $C(s, n)$ be a collision finding algorithm for a Damgård-Merkle hash function where s denotes the state at which the collision attack is applied and n, the number of message blocks present in each of the colliding messages. $C(s, n)$ can be either a brute force or a cryptanalytic collision finding algorithm. On a t-bit hash function, a brute force $C(s, n)$ requires about $2^{t/2}$ hash function computations to find a collision with 0.5 probability whereas a cryptanalytic $C(s, n)$ requires less effort than that. Joux multicollision attack [15] on a t-bit Damgård-Merkle hash uses a single-block brute force collision finder ($n = 1$) to find a 2^k collision with a computational work of $k \times 2^{t/2}$ computations of the compression function. This attack can be extended to multiple blocks ($n \geq 2$) without any additional work by calling this algorithm with at least two random message blocks in every call to it.

3.2 Checksum control sequences

We define checksum control sequence (CCS) as a chunk of data which lets an attacker to control the checksum value in the hash functions with linear checksums. The attacker constructs the CCS by building a Joux multicollision of the correct size using a random choice of message blocks. He then uses the CCS to actually control the checksum using a checksum control algorithm without changing any intermediate hash value on the iterative chain.

For example, a 2^k 2-block collision on the underlying Damgård-Merkle construction of **3C** (ignoring the linear-XOR checksum) using a brute-force collision finding algorithm gives the attacker k independent choices of parts of the intermediate states that form the CCS. When the attacker wants a particular k-bit checksum value, he can turn the problem of finding which choices to make from the CCS into the problem of solving a system of k linear equations in k unknowns, something the attacker can do very efficiently using existing tools such as Gaussian elimination [3, Appendix A], [5, 43]. This is schematically shown in Figure 5 for $k = 2$ where the attacker performs a 2^2 collision using random 2-block messages to compute the CCS. Then he has a choice to choose either $H_1^0 \oplus H_2$ or $H_1^1 \oplus H_2$ from the first 2-block collision and either $H_3^0 \oplus H_4$ or $H_3^1 \oplus H_4$ from the second 2-block collision of the CCS to control 2 bits of the checksum without changing the hash value after the CCS. An algorithm to defeat the linear XOR checksum on a t-bit **3C** is given in the following Section.



(M_1, N_1)
(M_2, N_2)
(M_3, N_3)
(M_4, N_4)
M_5
M_6
M_7
Checksum
0
H_0
f
(H_1^0, H_1^1)
H_2
(H_3^0, H_3^1)
H_4
H_5
H_6
H_7
Checksum control sequence controls 2 bits of linear−xor checksum
Message blocks where generic attack happens: multi−block multicollision, second preimage, herding

Fig. 5. Using checksum control sequence to control 2 bits of the checksum

3.3 Defeating linear-XOR checksums in hash functions

ALGORITHM: Defeat linear-XOR checksum on 3C
Variables:

1. (e_i^0, e_i^1) = A pair of independent choices of random values after every 2-block collision in the 2^t 2-block collision on **3C** and $e_i^0 \neq e_i^1$ for $i = 1, 2, \ldots, t$.
2. $a = a[1], a[2], \ldots, a[t]$ = Any t-bit string.
3. $D = D[1], D[2], \ldots, D[t]$ = The desired t-bit checksum to be imposed.
4. i, j = Temporary variables.

Steps:

1. Build a CCS for **3C** by constructing a 2^t 2-block collision on the underlying Damgård-Merkle of **3C** using a brute force collision finding algorithm $C(s, 2)$. Now the CCS contains t independent choices of parts of the intermediate states where each choice imposes a random XOR difference on the t-bit linear-XOR checksum at the end of the 2^t 2-block collision.
2. Each of the parts of the CCS gives one choice e_i^0 or e_i^1 for $i = 1, 2, \ldots, t$ to determine some random t-bit value that either is or is not XORed into the final checksum value at the end of the 2^t 2-block collision. Now $e_i^0 = H_{2i-1}^0 \oplus H_{2i}^0$ and $e_i^1 = H_{2i-1}^1 \oplus H_{2i}^1$ for $i = 1, 2, \ldots, t$.
3. For any t-bit string $a = a[1], a[2], \ldots, a[t]$, let $e^a = e_1^a, \ldots, e_t^a$.
4. Find $a = a[1], a[2], \ldots, a[t]$ such that $e_1^{a[1]} \oplus e_2^{a[2]} \oplus \ldots \oplus \ldots e_t^{a[t]} = D$. We now solve the equation: $\bigoplus_{i=1}^{t} e_i^0 \times a[i] \oplus e_i^1 \times (1 - a[i]) = D$.
5. Each bit position of $e_i^{a[i]}$ gives one equation and turn the above into t equations, one for each bit. Let $\overline{a}[i] = 1 - a[i]$.
6. The resulting system is: $\bigoplus_{i=1}^{t} e_i^0[j] \times a[i] \oplus e_i^1[j] \times \overline{a}[i] = D[j]$ $(j = 1, \ldots, t)$. Here there are t linear equations in t unknowns that need to be solved for the solution $a[1], a[2], \ldots, a[t]$.
7. The solution $a[1], a[2], \ldots, a[t]$ lets us determine the blocks in the 2^t 2-block collision that form the prefix to give the desired checksum D.

Work: It requires about $t \times 2^{t/2}$ computations of the compression function to produce a 2^t 2-block collision to construct the CCS and at most $t^3 + t^2$ bit-XOR operations to solve a system of $t \times t$ equations using Gaussian elimination to find a solution with 0.5 probability [3, Appendix A], [5,43].

3.4 Defeating XOR-linear checksums in other designs

A similar technique defeats the checksum in F-Hash. To defeat the checksum in **3CM**, two sets of equations due to the XOR checksum in the second and third chains need to be solved. The attack algorithms on these designs have been placed in the Appendices B.1 and B.2 respectively. If a linear-XOR checksum is computed using message blocks and intermediate states, linear equations due to XOR of the intermediate states and message blocks need to be solved independently.

4 Techniques to defeat checksums in the designs with additive checksums

Consider an additive checksum mod 2^k computed using messages. A $2^{(k/2)+1}$ Joux multicollision does not allow complete control of the checksum, but it does allow an attacker to usually find a pair of messages within the multicollision whose additive checksum differs by any desired value. This can be done by generating all $2^{(k/2)+1}$ possible checksum values from the multicollision, and doing a modified collision search for a pair of messages whose additive difference is the desired value.

Given this technique, a sequence of k successive $2^{(k/2)+1}$ Joux multicollisions can be used to completely control an additive checksum mod 2^k. The first $2^{(k/2)+1}$ multicollision is used to find a pair of $k/2+1$ 1-block messages whose checksum differs by 1, the next multicollision is used to find a pair of $(k/2+1)$ 1-block messages whose checksum differs by 2, and so on through the k^{th} $2^{(k/2)+1}$ multicollision, which yields a pair of $(k/2+1)$ 1-block messages whose checksum differs by 2^{k-1}. At this point, the attacker can easily choose a message to get any checksum he chooses, without affecting the intermediate hash value after the CCS.

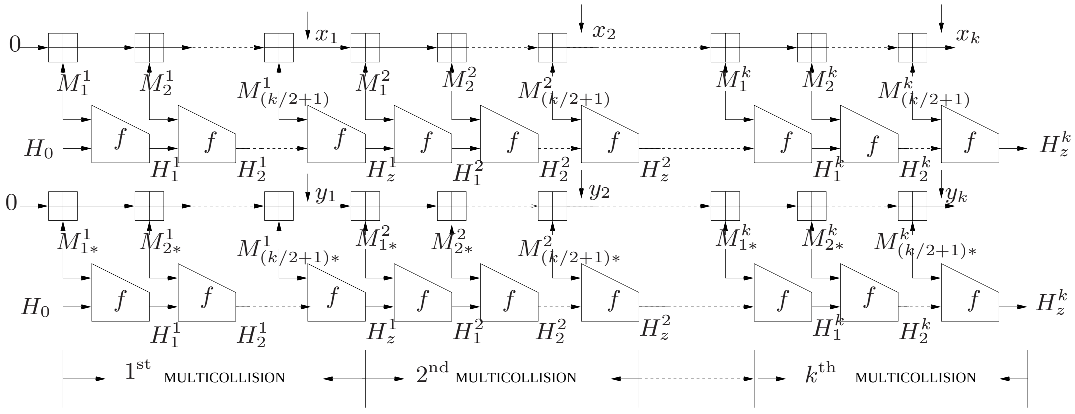


Fig. 6. Defeating additive checksum mod 2^k computed using messages

This technique is schematically represented in Figure 6. In the first $2^{(k/2)+1}$ 1-block multicollision, $f(H_{i-1}^1, M_i^1) = f(H_{i-1}^1, M_{i*}^1) = H_i^1$ where $i = 1$ to $k/2+1$, $H_0^1 = H_0$ is the initial state, H_i^1 are the intermediate states and M_i^1,M_{i*}^1 are the colliding blocks. The intermediate state at the end of the first multicollision is H_z^1 where $z = k/2+1$. We then find a pair of collision paths from the $2^{k/2+1}$ different collision paths in this multicollision, such that their respective additive checksums x_1 and y_1 satisfy the condition $x_1 \equiv y_1 + 1 \bmod 2^b$. In the second $2^{(k/2)+1}$ 1-block multicollision, $f(H_{i-1}^2, M_i^2) = f(H_{i-1}^2, M_{i*}^2) = H_i^2$ where $i = 1$ to $k/2+1$ and the intermediate hash value at the end of second multicollision is H_z^2. We then find a pair of collision paths from the $2^{k/2+1}$ different collision paths in this multicollision, such that their respective additive checksums x_2 and y_2 satisfy the condition $x_2 \equiv y_2 + 2 \bmod 2^b$. This process is repeated for k times. In the k^{th} $2^{(k/2)+1}$ 1-block multicollision, $f(H^k_{i-1}, M^k_i) = f(H^k_{i-1}, M^k_{i*}) = H^k_i$ for $i = 1$ to $k/2 + 1$ and the intermediate hash value at the end of k^{th} multicollision is H^k_z. We then find a pair of collision paths from the $2^{k/2+1}$ different collision paths in this multicollision, such that their respective additive checksums x_k and y_k satisfy the condition $x_k \equiv y_k + 2^{k-1} \bmod 2^b$. We now obtain the CCS by concatenating all these individual collision paths and then we force the additive checksum to the desired one by choosing either of the two available paths in each of the k $2^{k/2+1}$-collision paths. By now the attacker has found a message that forces the checksum to the desired value, without affecting the intermediate hash value after the CCS.

Below, we provide an algorithm to defeat the additive checksum in the GOST hash function.

ALGORITHM: Defeating checksum in GOST

Variables:

1. $i, j, k =$ integers.
2. $chunk[i] =$ a pair of $(b/2) + 1$-message block sequences denoted by $(e^0_i,\, e^1_i)$.
3. $H_0 =$ initial state.
4. $H^i_j =$ the intermediate state on the iterative chain.
5. $(M^i_j, N^i_j) =$ a pair of message blocks each of b bits.
6. $T =$ Table with three columns: a $(b/2) + 1$-collision path, addition modulo 2^b of message blocks in that path and a value of 0 or 1.

Steps:

```
1. For $i = 1$ to $b$:
   – For $j = 1$ to $(b/2) + 1$:
      • Find $M^i_j$ and $N^i_j$ such that $f(H^i_{j-1}, M^i_j) = f(H^i_{j-1}, N^i_j) = H^i_j$ where $H^1_0 = H_0$. That is,
        build a $(b/2) + 1$-block multicollision where each block yields a collision on the iterative
        chain and there are $2^{(b/2)+1}$ different $(b/2) + 1$-block sequences of blocks all hashing to
        the same intermediate state $H^i_{(b/2)+1}$ on the iterative chain.
   – Find a pair of paths from the different $(b/2) + 1$-block sequences whose additive checksum
     differs by $2^{i-1}$. This is performed as follows:
      • $T =$ empty table.
      • for $j = 1$ to $2^{(b/2)+1}$
         * $C^i_j \equiv \sum_{k=1}^{(b/2)+1} X^i_k \bmod 2^b$ where $X^i_k$ can be either $M^i_k$ or $N^i_k$.
         * Add to $T$: $(C^i_j,\, 0,\, X^i_1||X^i_2|| \dots X^i_{(b/2)+1})$
         * Add to $T$: $(C^i_j + 2^{i-1},\, 1,\, X^i_1||X^i_2|| \dots X^i_{(b/2)+1})$.
      • Search $T$ to find colliding paths between the entries with 0 and 1 in the second column of
        $T$. Let these paths of $(b/2)+1$ sequence of blocks be $e^1_i$ and $e^0_i$ where $e^1_i \equiv e^0_i + 2^{i-1} \bmod 2^b$.
   – $chunk[i] = (e^0_i, e^1_i)$.
2. Construct CCS by concatenating individual chunks each containing a pair of $(b/2)+1$ blocks that
   hash to the same intermediate state on the iterative chain. The CCS is $chunk[1] \,||\, chunk[2] \dots \,||\, chunk[b]$.
3. The checksum at the end of the $2^b$ $(b/2)+1$-block collision can be forced to the desired checksum
   $D$ by choosing either of the sequences $e^0_i$ or $e^1_i$ from the CCS which is practically free to use
   and adding blocks in each sequence over modulo $2^b$.
```

Work: The work to defeat the additive checksum in GOST equals the work to construct $b\ 2^{(b/2)+1}$ 1-block collisions plus the work to find a chunk in each $2^{(b/2)+1}$ 1-block collision. It requires about $b \times ((b/2)+1) \times 2^{t/2}$ computations of the compression function and a time and space of $b \times 2^{b/2+1}$ for a collision search to find b chunks. For GOST, it requires about $2^8 \times 129 \times 2^{128} \approx 2^{143}$ computations of the compression function and a time and space of about $256 \times 2^{129} = 2^{137}$.

Remark 1. A more efficient attack is available if the attacker can exert direct control over the message blocks, rather than simply using a large Joux multicollision. In this case, the attacker constructs a Joux multicollision in such a way that each pair of colliding messages has a fixed power of two difference in its low k bits, and a random difference in its high $b-k$ bits. This allows direct control over the low k bits of the checksum, while leaving the high $b-k$ bits uncontrolled. A second Joux multicollision is then constructed, in which each pair of messages differs only in the high $b-k$ bits. The second multicollision may then be used to control the high $b-k$ bits of the checksum by brute force.

Using this technique on GOST with $k = 128$ leads to a CCS of only 256 message blocks. However, each attempt to control the checksum requires a 2^{128} brute force search in this case. Alternatively, with $k = 32$ (and using multi-block collisions to construct the Joux multicollision), the CCS is 1024 message blocks, and an attempt to control the checksum requires only a 2^{32} brute force search.

4.1 Defeating additive checksums in other designs

Similarly, we can defeat additive checksums for the variants of GOST that compute additive checksum mod 2^k using intermediate hash values by building a 2^k Joux multicollision where each collision in it consists of $2^{(k/2)+1}$ multicollisions constructed using a sequence of $(k/2+1)$ 2-block collisions as shown for the **3CA** design in Appendix E. We note that this trick can also be used to defeat the additive checksum computed for a design using both the message blocks and intermediate hash values. In this case, a $2^{(k/2)+1}$ Joux multicollision using 2-block messages is performed to allow an attacker to find a pair of messages (resp. intermediate hash values) within the multicollision whose additive checksum differs by any desired value. This can be done by generating all $2^{(k/2)+1}$ possible checksum values due to messages (resp. intermediate hash values) from the multicollision, and doing a modified collision search for a pair of messages (resp. intermediate hash values) whose additive difference is the desired value. We note that the efficient attack discussed in Remark 1 does not work on the hash functions that compute additive checksum using intermediate hash values as the attacker cannot exert control over the input to the checksum formed using intermediate hash values.

5 Generic attacks on hash functions with linear checksums

The fundamental approach used to perform the generic attacks on all the hash functions with linear checksums is similar. Because the approach is similar, we discuss it here only for **3C** and Appendices C and E discuss these generic attacks for other kinds of XOR-linear/additive checksums respectively. Broadly, it consists of the following steps:

1. Construct a CCS.
2. Combine the CCS with whatever other structure (expandable message, multicollision over single block or multiple blocks, diamond structure) is needed for the generic attack to work.

3. Carry out the generic attack, ignoring its impact on the linear checksum.
4. Use the CCS to control the linear checksum, forcing it to a value that permits the generic attack to work on the full hash function.

5.1 Meaningful multi-block multicollisions

Constructing and using the CCS does not imply random gibberish in the messages produced; using Yuval's trick [47], a brute-force search for the multicollision used in the CCS can produce collision pairs in which each possible message is a plausible-looking one. Here, the attacker can create two documents where one is genuine and the other one a forgery and can vary their meaning in such a way that at some point of variation they collide when processed using the same hash function. For example, the attacker can use this trick to construct CCS for the hash functions that maintain checksums using meaningful colliding messages based on the brute force collision finding techniques. This is possible when the CCSs to defeat the checksums are constructed from individual collisions that span over multiple message blocks as in (Dear Fred/Freddie,)(Enclosed please find/I have sent you) (a check for $100.00/a little something) and so on, where the attacker can choose either side of the slash for the next part of the sentence. In that case, any choice for the CCS used to defeat the checksum will be a meaningful message. The impact of this attack is that one can construct not only meaningful collisions but also second preimages and herded messages with genuine meaning for these hash functions.

5.2 Long-message second preimage attack on 3C

Long message second preimage attack on a t-bit **3C** hash function H is outlined below:
ALGORITHM: LongMessageAttack(M_{target}) on 3C
Find the second preimage for a message of $2^d + d + 2t + 1$ *blocks.*
Variables:

1. M_{target} = the target long message for which a second preimage is to be found.
2. M_{link} = linking message block to connect the intermediate state on the iterative chain at the end of the *expandable message* to some point in the sequence of the intermediate states on the iterative chain of the target message.
3. H_{exp} = the intermediate state on the iterative chain at the end of the *expandable message*.
4. H_t = the result of the 2^t 2-block collision on H starting from the initial state.
5. M_{final} = the second preimage for H of the same length as M_{target}.
6. M_{pref} = the checksum control prefix obtained from the CCS.

Steps:

1. Compute the intermediate hash values for M_{target} using H:
   - H_0 and h_0 are the initial states of the iterative and accumulation chains respectively.
   - M_i is the i^{th} message block of M_{target}.
   - $H_i = f(H_{i-1}, M_i)$ and $h_i = H_i \oplus h_{i-1}$ are the i^{th} intermediate states on the iterative and accumulation chains respectively.
   - The intermediate states on the iterative and accumulation chains are organised in some searchable structure for the attack, such as hash table. The hash values $H_1, \ldots, H_d$ and those obtained in the processing of t 2-block messages are excluded from the hash table as the *expandable message* cannot be made short enough to accommodate them in the attack.

2. Build a CCS for H by constructing a 2^t 2-block collision starting from the initial state H_0. Let H_t be the multicollision value and h_t be the corresponding checksum value which is random.
3. Construct a $(d, d+2^d-1)$-*expandable message* M_{exp} with H_t as the starting state using generic-expandable message algorithm from [18]. Append M_{exp} to the CCS and process it to obtain H_{exp}.
4. Process message blocks from the end of H_{exp} to find M_{link} such that $f(H_{exp}, M_{link})$ collides with one of the intermediate states on the iterative chain stored in the hash table while processing M_{target}. Let this matching value of the target message be H_u and the corresponding state in the accumulation chain be h_u where $d+2t+1 \leq u \leq 2^d+d+2t+1$.
5. Use the CCS built in step 2 to find the checksum control prefix M_{pref} by following the algorithm in Section 3.3. The prefix M_{pref} adjusts the state in the accumulation chain at that point to the desired value h_u of the long target message M_{target}. This is equivalent to adjusting the checksum value at the end of the 2^t 2-block collision.
6. Expand the *expandable message* to produce a message M^* of $u-1$ blocks long.
7. Return the second preimage $M_{final} = M_{pref}||M^*||M_{link}||M_{u+1}\dots M_{2^d+d+1+2t}$ of the same length as M_{target} such that $H(M_{final}) = H(M_{target})$.

Work: The work required to find a second preimage on **3C** is the work to construct a 2^t 2-block collision to build the CCS plus the work to solve a system of $t \times t$ linear equations plus the work to find the *expandable message* plus the work to find the linking message block. So, the *only* additional work required to perform the second preimage attack on **3C** over Damgård-Merkle is the work to find a 2^t 2-block collision and solve a system of $t \times t$ equations. Note that the work to produce the multicollision to build the CCS and solve the equations is very fast compared to the rest of the attack.
Illustration: The computational work required to find a second preimage for **3C-SHA-256** for a target message of $2^{54}+54+512+1$ blocks is $2^{136}+54\times2^{129}+2^{203}$ compression function computations and $2^{24}+2^{16}$ bit-XOR operations assuming abundant memory.

Remark 2. We note that 2^k second preimages can be constructed for hash functions with linear checksums at negligible additional cost using a target message of length $2^d+d+2t+k+1$ blocks. We first find a 2^k collision over 1-block messages with a work of $k\times2^{t/2}$ computations of the compression function and then perform our second preimage attack from the end of the 2^k collision.

5.3 Herding attack on 3C

The herding attack on a t-bit **3C** hash function H is outlined below:

1. Construct a 2^d hash value wide diamond structure for H and output the hash value H_v as the chosen target which is computed using any of the possible 2^{d-1} checksum values or some value chosen arbitrarily. Let h_c be that checksum value.
2. Build a CCS for H using a 2^t collision over 2-block messages. Let H_t be the intermediate state due to this multicollision on H.
3. When challenged with the prefix message P, process P using H_t. Let $H(H_t, P) = H_p$.
4. Find a linking message block M_{link} such that $H(H_p, M_{link})$ collides with one of the 2^d outermost intermediate states on the iterative chain in the diamond structure. If this collision is matched against all of the $2^{d+1}-2$ intermediate states in the diamond structure then a $(1, d+1)$-*expandable message* must be produced at the end of the diamond structure to ensure that the final herded message is always a fixed length.

5. Use the CCS computed in step 2 to force the checksum of the herded message P to h_c using the attack to defeat the checksum described in Section 3.3. Let M_{pref} be the checksum control prefix obtained after solving the system of equations.
6. Finally, output the message $M = M_{pref}||P||M_{link}||M_d$ where M_d are the message blocks in the diamond structure that connect $H(H_p, M_{link})$ to the chosen target H_v[2]. Now $H_v = H(M)$.

Work: The computational work required to perform the herding attack on **3C** is the work required to build the CCS plus the work to solve the system of equations plus the work required to perform the herding attack from [17]. This equals about $t \times 2^{t/2} + 2^{t/2+d/2+2} + d \times 2^{t/2+1} + 2^{t-d-1}$ computations of the compression function and $t^3 + t^2$ bit-XOR operations assuming that all of the $2^{d+1} - 2$ intermediate states are used for searching in the diamond structure. Note that the work required to build the CCS using multicollision and to solve the system of equations is negligible compared to the rest of the attack.
Illustration: The work to perform the herding attack on **3C-SHA-256** with $d = 84$ is $2^{136} + 2^{172} + 84 \times 2^{129} + 2^{171} \approx 2^{172}$ computations of SHA-256 compression function and $2^{24} + 2^{16}$ bit-XOR operations.

Remark 3. We note that [12,13] shows only the application of Joux multicollision attack on **3C** over 1-block messages. However, using our attack technique from Section 3.3, one can find multicollsions for **3C** over multiple blocks by defeating the XOR-linear checksum.

5.4 Extending the generic attacks on to CRCs

The techniques used to defeat the checksums described in Sections 3.3 and 3.4 can be extended to any linear checksum which is reasonably short. Consider a 512-bit CRC computed over a message and used as the final checksum block. Now we construct the CCS using a 2^{512} Joux multicollision and then append whatever message at the end of the multicollision to perform the generic 2nd-preimage and herding attacks. Each bit of the 512-bit CRC is a linear function of 512 binary variables $a[1], a[2], \ldots, a[512]$ where $a[i]$ selects a message block from one of the sides of the collision in the CCS. One can perform the generic 2nd-preimage and herding attacks ignoring the checksum value, then solve the resultant system of 512×512 equations to force the checksum to the value necessary to make the attack work.

Remark 4. We note that for a hash function with XOR-linear/additive checksum of intermediate hash values producing checksum as the hash value, one can find preimages for a given target hash value cheaply by forcing the checksum to the target hash value. For example, assume that checksum value is used as the final hash value in the **3C** construction. The attacker performs the 2^t 2-block multicollision and then processes the block with the encoding of the length ($2t$ blocks) of the message used in the multicollision. Using the target hash value and output of the length encoded block, the attacker finds the desired checksum he needs at the end of the 2^t 2-block multicollision so that he can force the hash value to the target hash value.

[2] Note that when P is processed using the initial state H_0 of H followed by a 2^t 2-block multicollision from the state $H(H_0, P)$, we can output message M with the format $P||M_{pref}||M_{link}||M_d$.

6 On the possibility of generic attacks using cryptanalytic collision attacks

We note that it is difficult to construct the CCSs using cryptanalytic collision finding algorithms such as the ones built on MD5 and SHA-1 [45, 46] in order to defeat linear checksums in hash functions to carry out generic attacks. For example, consider two 2-block colliding messages of format $(M_{2.i-1}, M_{2.i})$,$(N_{2.i-1}, N_{2.i})$ for $i = 1, \ldots, t$ on the underlying **MD** of **3C** based on near collisions due to the first blocks in each pair of the messages. Usually, the XOR differences of the nearly collided intermediate states are either fixed or very tightly constrained as in the collision attacks on MD5 and SHA-1 [45,46]. It is difficult to construct a CCS due to the inability to control these fixed or constrained bits. Similarly, it is also difficult to build the CCSs using collisions of the messages of format $(M_{2.i-1}, M_{2.i})$,$(N_{2.i-1}, M_{2.i})$ for $i = 1, \ldots, t$. We note that one may create variants for the known cryptanalytic collision finding algorithms as in [41] that could be used to construct the CCS. This is an open question now. It is not possible to control the checksum due to 2-block collisions of the format $(M_{2.i-1}, M_{2.i})$, $(M_{2.i-1}, N_{2.i})$ for $i = 1, \ldots, t$ [44] as this format produces a zero linear-XOR checksum difference after every 2-block collision.

Alternatively, assume that two random message blocks are processed initially using f to obtain two different random intermediate states s_1 and s_2 and then a cryptanalytic collision finding algorithm $C(s_1, s_2, 1)$ is called with s_1 and s_2 as parameters which produces either the same or different message blocks that collide. In effect, the XORed-together intermediate states after every two blocks in the 2^t 2-block collision are random and a CCS can be constructed to defeat the checksum.

6.1 Multi-block collision attacks on hash functions with linear checksums

Though we cannot perform generic attacks on the hash functions with linear checksums using structured collisions, we can still perform multi-block collision attacks. Consider a collision finding algorithm $C(s, 1)$ with $s = H_0$ for the GOST hash function H. A call to $C(s, 1)$ results in a pair of b-bit message blocks (M_1, N_1) such that $M_1 \equiv N_1 + \varDelta \bmod 2^b$ and $f(H_0, M_1) = f(H_0, N_1) = H_1$. Now call $C(s, 1)$ with $s = H_1$ which results in a pair of blocks (M_2, N_2) such that $N_2 \equiv M_2 + \varDelta \bmod 2^b$ and $f(H_1, M_2) = f(H_1, N_2) = H_2$. That is, $H(H_0, M_1||M_2) = H(H_0, N_1||N_2)$. Consider $M_1 + M_2 \bmod 2^b = \varDelta + N_1 + N_2 - \varDelta \bmod 2^b = N_1 + N_2 \bmod 2^b$, a collision in the chain which computes additive checksum. Hence, by just appending two structured collisions end to end, we get a collision for the hash function with linear checksum. Similarly, structured collisions on Damgård-Merkle hash functions can be converted to multi-block collisions on other linear checksums as shown in Appendix F.

7 Comparison of our attack with that of Mironov-Narayanan

Independent to our work, Mironov and Narayanan [30] have found an alternative technique to defeat linear-XOR checksum computed using message blocks. We call this design GOST-x and is shown in Figure 7.

While our approach to defeat the XOR checksum in GOST-x requires finding a 2^b-collision using b random 1-block messages (M_i, N_i) for $i = 1$ to b, their technique considers repetition of the same message block twice for a collision. In contrast to the methods presented in this paper for solving system of linear equations for the whole message, their approach solves the system of linear equations once after processing every few message blocks. We note that this constrained choice of

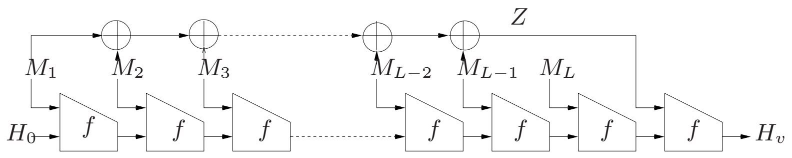


Fig. 7. GOST-x hash function

messages would result in a zero checksum at the end of the 2^b multicollision on this structure and thwarts the attempts to perform the second preimage attack on GOST-x. The reason is that the attacker loses the ability to control the checksum after finding the linking message block from the end of the expandable message which matches some intermediate state obtained in the long target message.

However, we note that their technique with a twist can be used to perform the herding attack on GOST-x. In this variant, the attacker chooses the messages for the diamond structure that all have the same effect on the linear-XOR checksum. These messages would result in a zero checksum at every stage in the diamond structure. Once the attacker is forced with a prefix, processing the prefix gives a zero checksum to start with and then solving a system of equations will find a set of possible linking messages that will all combine with the prefix to give a zero checksum value.

When the approach of [30] is applied to defeat the checksums in **3C**, **3CM** and F-Hash, the 2^t 2-block collision finding algorithm used to construct the CCS must output the same pair of message blocks on the either side of the collision whenever it is called. The technique of [30] imposes constraints not present in our technique, and is not quite as powerful. However, it could be quite capable of defeating linear-XOR checksums in many generic attacks. Because it is so different from our technique, some variant of this technique might be useful in cryptanalytic attacks for which our techniques do not work.

8 Concluding remarks

Our results demonstrate that maintaining large internal state sizes using XOR-linear/additive checksums is not good enough for the security of the hash function. In addition, widening the compression functions [27] along with the XOR-linear/additive checksums provide very little additional security against generic attacks compared to the wide-pipe hash. One question left open by our research is what properties would ensure that a checksum would block generic attacks. It is clear that it must be impossible for the attacker to build and use a CCS, but not clear that this is a sufficient condition.

References

1. Mihir Bellare. New Proofs for NMAC and HMAC: Security Without Collision-Resistance. In Cynthia Dwork, editor, *Advances in Cryptology—CRYPTO '06*, volume 4117 of *Lecture Notes in Computer Science*. Springer-Verlag, 20–24 August 2006. Full version of the paper is available at `http://www-cse.ucsd.edu/users/mihir/papers/hmac-new.html`. Last access date: 7th of August 2007.
2. Mihir Bellare, Ran Canetti, and Hugo Krawczyk. Keying hash functions for message authentication. In Neal Koblitz, editor, *Advances in Cryptology—CRYPTO '96*, volume 1109 of *Lecture Notes in Computer Science*, pages 1–15. Springer-Verlag, 18–22 August 1996. Full version of the paper is available at `http://www-cse.ucsd.edu/users/mihir/papers/hmac.html`. Last access date: 16th of August 2007.

3. Mihir Bellare and Daniele Micciancio. A new paradigm for collision-free hashing: Incrementality at reduced cost. In Walter Fumy, editor, *Advances in Cryptology: Proceedings of EUROCRYPT'97*, volume 1233 of *Lecture Notes in Computer Science*, pages 163–192, 1997. The full version of the paper is available at `http://www-cse.ucsd.edu/~mihir/papers/incremental.html`. Last access date: 29th of August 2007.
4. Eli Biham and Orr Dunkelman. A Framework for Iterative Hash Functions-HAIFA. Technical report, August 2006. The paper and slides of this work are available at `http://csrc.nist.gov/pki/HashWorkshop/2006/program_2006.htm`. Last access date: 15th of February 2007.
5. Don Coppersmith. Two Broken Hash Functions. Technical Report IBM Research Report RC-18397, IBM Research Center, October 1992.
6. Joan Daemen, Michael Peeters, and Gilles Van Assche. RadioGatun, a Belt-and-Mill Hash Function. Technical report, August 2006. The paper and slides of this work are available at `http://csrc.nist.gov/pki/HashWorkshop/2006/program_2006.htm`. Last access date: 15th of February 2007.
7. Ivan Damgård. A Design Principle for Hash Functions. In Gilles Brassard, editor, *Advances in Cryptology: CRYPTO 89*, volume 435 of *Lecture Notes in Computer Science*, pages 416–427. Springer-Verlag, 1989.
8. Richard Drews Dean. *Formal Aspects of Mobile Code Security*. PhD thesis, Princeton University, 1999.
9. Federal Information Processing Standard (FIPS). *Secure Hash Standard*. National Institute for Standards and Technology, August 2002. This document is available at `http://csrc.nist.gov/publications/fips/fips180-2/fips180-2.pdf`. Last access date: 11th of March 2007.
10. Decio Gazzoni Filho, Paulo Barreto, and Vincent Rijmen. The Maelstrom-0 Hash Function. Published at 6th Brazilian Symposium on Information and Computer System Security, 2006.
11. Praveen Gauravaram. *Cryptographic Hash Functions: Cryptanalysis, Design and Applications*. PhD thesis, Information Security Institute, Queensland University of Technogy, June 2007.
12. Praveen Gauravaram, William Millan, Ed Dawson, Matt Henricksen, Juanma Gonzalez Nieto, and Kapali Viswanathan. Constructing Secure Hash Functions by Enhancing Merkle-Damgård Construction (full version). Technical Report QUT-ISI-TR-2006-013, Information Security Institute (ISI), Queensland University of Technology (QUT), July 2006. This technical report is available at `http://www.isi.qut.edu.au/research/publications/technical/qut-isi-tr-2006-013.pdf`. Last access date: 12th of september 2007.
13. Praveen Gauravaram, William Millan, Ed Dawson, and Kapali Viswanathan. Constructing Secure Hash Functions by Enhancing Merkle-Damgård Construction. In *Australasian Conference on Information Security and Privacy (ACISP)*, volume 4058 of *Lecture Notes in Computer Science*, pages 407–420, 2006.
14. Jonathan Hoch and Adi Shamir. Breaking the ICE: Finding Multicollisions in Iterated Concatenated and Expanded (ICE) Hash Functions. To appear in the Proceedings of 13th Annual Fast Software Encryption (FSE) International Conference, 2006. A preliminary version of this paper is available at `http://paginas.terra.com.br/informatica/paulobarreto/hflounge.html#HS06`. Last access date: 27th of January 2007.
15. Antoine Joux. Multicollisions in Iterated Hash Functions. Application to Cascaded Constructions. In Matt Franklin, editor, *Advances in Cryptology-CRYPTO 2004*, volume 3152 of *Lecture Notes in Computer Science*, pages 306–316, Santa Barbara, California, USA, August 15–19 2004. Springer.
16. Burt Kaliski. *RFC 1319: The MD2 Message-Digest Algorithm*. Internet Activities Board, April 1992. This RFC is available at `http://www.ietf.org/rfc/rfc1319.txt`. Last access date: 23rd of May 2007.
17. John Kelsey and Tadayoshi Kohno. Herding Hash Functions and the Nostradamus Attack. In Serge Vaudenay, editor, *Advanes in Cryptology-EUROCRYPT 2006*, volume 4004 of *Lecture Notes in Computer Science*, pages 183–200. Springer, 2006.
18. John Kelsey and Bruce Schneier. Second Preimages on n-bit Hash Functions for Much Less than 2n̂ Work. In Ronald Cramer, editor, *Advances in Cryptology - EUROCRYPT 2005*, volume 3494 of *Lecture Notes in Computer Science*, pages 474–490. Springer, 2005.
19. Lars Knudsen. SMASH – A cryptographic hash function. In Henri Gilbert and Helena Handschuh, editors, *IWFSE: International Workshop on Fast Software Encryption*, volume 3557 of *Lecture Notes in Computer Science*, pages 228–242. Springer, 2005.
20. Lars Knudsen and John Mathiassen. Preimage and Collision attacks on MD2. In Henri Gilbert and Helena Handschuh, editors, *Fast Software Encryption FSE: 12th International Workshop*, volume 3557 of *Lecture Notes in Computer Science*, pages 255–267. Springer, 2005.
21. Xuejia Lai and James L. Massey. Hash Functions Based on Block Ciphers. In R. A. Rueppel, editor, *Advances in Cryptology—EUROCRYPT 92*, volume 658 of *Lecture Notes in Computer Science*, pages 55–70. Springer-Verlag, 24–28 May 1992.
22. Mario Lamberger, Norbert Pramstaller, Christian Rechberger, and Vincent Rijmen. Second preimages for SMASH. In *RSA Conference*, volume 4377 of *Lecture Notes in Computer Science*, pages 101–111. Springer, 2007.

23. Duo Lei. F-HASH: Securing Hash Functions Using Feistel Chaining. Cryptology ePrint Archive, Report 2005/430, 2005. The paper is available at `http://eprint.iacr.org/2005/430.pdf`. Last access date: 11th of November 2006.
24. Duo Lei. New Integrated proof Method on Iterated Hash Structure and New Structures. Cryptology ePrint Archive, Report 2006/147, 2006. The paper is available at `http://eprint.iacr.org/2006/147.pdf`. Last access date: 5th of November 2006.
25. Serguei Leontiev and Grigorij Chudov. *RFC:4357 Additional Cryptographic Algorithms for Use with GOST 28147-89, GOST R 34.10-94, GOST R 34.10-2001, and GOST R 34.11-94 Algorithms*. Internet Engineering Task Force, January 2006. This Informational RFC 4490 is available at `http://www.ietf.org/rfc/rfc4490.txt`. Last access date: 12th of September 2007.
26. Serguei Leontiev and Dennis Shefanovski. *RFC:4491 Using the GOST R 34.10-94, GOST R 34.10-2001, and GOST R 34.11-94 Algorithms with the Internet X.509 Public Key Infrastructure Certificate and CRL Profile*. Internet Engineering Task Force, May 2006. This standards track RFC 4491 is available at `http://www.ietf.org/rfc/rfc4491.txt`. Last access date: 11th of September 2007.
27. Stefan Lucks. A Failure-Friendly Design Principle for Hash Functions. In Bimal Roy, editor, *Advances in Cryptology - ASIACRYPT 2005*, volume 3788 of *Lecture Notes in Computer Science*, pages 474–494. Springer-Verlag, 2005.
28. Stefan Lucks. Hash Function Modes of Operation. Presented at ICE-EM RNSA 2006 Workshop on Recent Advances in Stream Ciphers and Hash Functions at the Queensland University of Technology (QUT), Brisbane, Australia., June 2006.
29. Ralph Merkle. One way Hash Functions and DES. In Gilles Brassard, editor, *Advances in Cryptology: CRYPTO 89*, volume 435 of *Lecture Notes in Computer Science*, pages 428–446. Springer-Verlag, 1989.
30. Ilya Mironov and Arvind Narayanan. Personal communication during the rump session of Crypto'06, August 2006.
31. Frédéric Muller. The MD2 Hash Function Is Not One-Way. In Pil Joong Lee, editor, *Advances in Cryptology - ASIACRYPT 2004, 10th International Conference on the Theory and Application of Cryptology and Information Security*, volume 3329 of *Lecture Notes in Computer Science*, pages 214–229. Springer, 2004.
32. Mridul Nandi and Douglas Stinson. Multicollision attacks on some generalized sequential hash functions. Cryptology ePrint Archive, Report 2006/055, 2006. The paper is available at `http://eprint.iacr.org/2006/055`. Last access date: 19th of September 2006.
33. National Institute of Standards and Technology (NIST). Announcing the Development of New Hash Algorithms for the Revision of Federal Information Processing Standard (FIPS) 180-2, Secure Hash Standard, January 2007. This notice by NIST is available at `http://www.csrc.nist.gov/pki/HashWorkshop/timeline.html` with the Docket No: 061213336-6336-01. Last access date: 16th of February 2007.
34. Government Committee of the Russia for Standards. GOST R 34.10-94, Gosudarstvennyi Standard of Russian Federation, Cryptographic Protection for Data Processing Systems Government Committee of USSR for Standards, 1989 (in Russian)., 1989.
35. Government Committee of the Russia for Standards. GOST R 34.11-94, Gosudarstvennyi Standard of Russian Federation, Information Technology, Cryptographic Data Security, Hashing function, 1994.
36. Vladimir Popov, Igor Kurepkin, and Serguei Leontiev. *RFC:4357 Additional Cryptographic Algorithms for Use with GOST 28147-89, GOST R 34.10-94, GOST R 34.10-2001, and GOST R 34.11-94 Algorithms*. Internet Engineering Task Force, January 2006. This Informational RFC 4357 is available at `http://www.ietf.org/rfc/rfc4357.txt`. Last access date: 11th of September 2007.
37. Norbert Pramstaller, Christian Rechberger, and Vincent Rijmen. Breaking a New Hash Function Design Strategy Called SMASH. In Bart Preneel and Stafford E. Tavares, editors, *Selected Areas in Cryptography*, volume 3897 of *Lecture Notes in Computer Science*, pages 233–244. Springer, 2006.
38. Vincent Rijmen and Paulo S. L. M. Barreto. The WHIRLPOOL hash function. This hash function was adopted as a standard by ISO/IEC 10118-3:2004, 2004. The specification is available at `http://paginas.terra.com.br/informatica/paulobarreto/WhirlpoolPage.html`. Last access date: 27th of August 2007.
39. Ronald Rivest. The MD5 message-digest algorithm. Internet Request for Comment RFC 1321, Internet Engineering Task Force, April 1992.
40. Ronald Rivest. Abelian Square-free Dithering and Recoding for Iterated Hash Functions. Technical report, October 2005. The paper and slides of this work are available at `http://csrc.nist.gov/pki/HashWorkshop/2005/program.htm`. Last access date: 15th of February 2007.
41. Marc Stevens, Arjen K. Lenstra, and Benne de Weger. Chosen-Prefix Collisions for MD5 and Colliding X.509 Certificates for Different Identities. In Moni Naor, editor, *EUROCRYPT*, volume 4515 of *Lecture Notes in*

Computer Science, pages 1–22. Springer, 2007. A version of this paper is available with a different title at `http://eprint.iacr.org/2006/360`. Last access date: 1st of September,2007.

42. Jiri Tuma and Daniel Joscak. Multi-block Collisions in Hash Functions based on 3C and 3C+ Enhancements of the Merkle-Damgård Construction. In Min Surp Rhee and Byoungcheon Lee, editor, *Information Security and Cryptology ICISC*, volume 4296 of *Lecture Notes in Computer Science*, pages 257–266, 2006.
43. David Wagner. A Generalized Birthday Problem. In Moti Yung, editor, *Advances in Cryptology - CRYPTO 2002*, volume 2442 of *Lecture Notes in Computer Science*, pages 288–303. Springer, 2002.
44. Xiaoyun Wang, Yiqun Lisa Yin, and Hongbo Yu. Efficient collision search attacks on SHA-0. In Victor Shoup, editor, *Advances in Cryptology—CRYPTO '05*, volume 3621 of *Lecture Notes in Computer Science*, pages 1–16. Springer, 2005, 14–18 August 2005.
45. Xiaoyun Wang, Yiqun Lisa Yin, and Hongbo Yu. Finding collisions in the full SHA-1. In Victor Shoup, editor, *Advances in Cryptology—CRYPTO '05*, volume 3621 of *Lecture Notes in Computer Science*, pages 17–36. Springer, 2005, 14–18 August 2005.
46. Xiaoyun Wang and Hongbo Yu. How to Break MD5 and Other Hash Functions. In Ronald Cramer, editor, *Advances in Cryptology - EUROCRYPT 2005*, volume 3494 of *Lecture Notes in Computer Science*, pages 19–35. Springer, 2005.
47. Gideon Yuval. How to swindle Rabin. *Cryptologia*, 3(3):187–189, July 1979.

A Other proposals of hash functions with linear checksums

A.1 F-Hash hash function

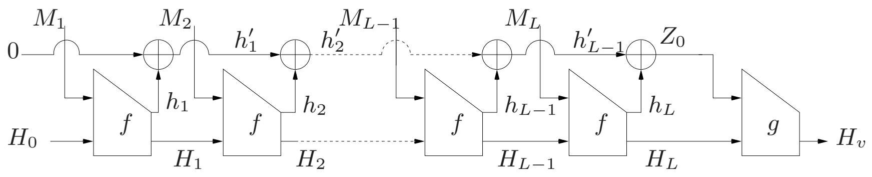


Fig. 8. The F-Hash hash function

The initial state of a t-bit F-Hash hash function shown in Figure 8 is H_0 and the compression function f is a Feistel structure based on a round function F of r rounds [23, 24]. The message M to be processed using F-Hash is split into equal size blocks M_i for $i = 1$ to L including padding which is similar to the padding of Damgård-Merkle hash functions. The two intermediate hash values h_i and H_i) each of t bits for every iteration of the compression function f are given by $H_i = f_r(M_i, H_{i-1})$ and $h_i = f_{r-1}(M_i, H_{i-1})$ where $i = 1, 2, \ldots, L$; f_r is the r-round iteration of F and f_{r-1} is the r-1-round iteration of F. The accumulation value at any iteration i is given by $h'_i = \bigoplus_{j=1}^{i} h_j$. The intermediate hash values (H_i, h_i) together at any iteration i are considered as iterative intermediate hash values. F-Hash computes checksum $Z_0 = \bigoplus_{i=1}^{L}(h_i)$ using part h_i of the iterative intermediate hash values. The hash value is $H_v = g(H_L, Z_0)$ where g is the iteration of F for r rounds.

A.2 3CM used in MAELSTROM-0

Inspired by **3C** and its variants, Filho *et al.* [10] have proposed a variant of **3C** called **3CM** as a replacement for the Merkle-Damgård construction in the Whirlpool hash function [38] for better protection against the multi-block collision attacks. This new construction is called MAELSTROM-0. In the **3CM** construction, for every iteration of f, the t-bit accumulation value in the third chain is updated using a linear feedback shift register (LFSR) denoted by ζ in Figure A.2. The LFSR ζ is an implementation of one-byte left shift of the t-bit accumulation chaining value and a conditional one byte XOR applied to that by a constant. Then modulo 2 addition of this result with the iterative chain data is performed. At every iteration i of the compression function in **3CM**, the iterative intermediate hash values are denoted by H_i, the checksum values in the second chain by h'_i and the checksum values in the third chain by h''_i. At any iteration i, the checksum value of the second chain is $h'_i = h'_{i-1} \oplus H_i$ or $\bigoplus_{i=1}^{i} H_i$. At any iteration i, the checksum value in the third chain is $h''_i = \zeta(h''_{i-1}) \oplus H_i$. After processing all the message blocks including the last block containing the Merkle-Damgård strengthening, the checksum values of the second and third chains are concatenated and padded with 0's if necessary to obtain a b-bit data block. This block is processed using the final compression function denoted by g in Figure A.2. For example, if the compression function f is SHA-1, then the concatenated checksum values from both the chains are padded with 192 0 bits to obtain a 512-bit block. For example, if the compression function is SHA-256 then no padding is performed for the concatenated checksum block as it is already 512 bits.

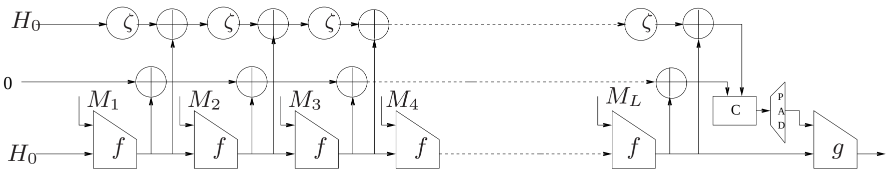


Fig. 9. The **3CM** construction used in Maelstrom-0

B Techniques to defeat linear checksums in F-Hash and 3CM

B.1 Defeating linear checksum in F-Hash

The following algorithm is used to defeat the checksum in F-Hash.
ALGORITHM: Defeat linear checksum in F-Hash
Variables:

1. (e_i^0, e_i^1) = A pair of independent choices of random values after every 2-block collision in the 2^t 2-block collision on F-Hash and $e_i^0 \neq e_i^1$ for $i = 1, 2, \ldots, t$.
2. $a = a[1], a[2], \ldots, a[t]$ = Any t-bit string.
3. $D = D[1], D[2], \ldots, D[t]$ = The desired t-bit checksum to be imposed.
4. i, j = Temporary variables.

Steps:

1. Build a CCS for F-Hash by building a 2^t 2-block multicollision by calling a brute-force collision finding algorithm $C(s, 2)$ t times on the Damgård-Merkle chain of F-Hash.
2. Each of the parts of the CCS gives us one choice (e_i^0 or e_i^1) for $i = 1, 2, \ldots, t$ to determine some random t-bit value that either is or is not XORed into the final checksum value at the end of 2^t multicollision. Note that $e_i^0 = h_{2i-1}^0 \oplus h_{2i}^0$ and $e_i^1 = h_{2i-1}^1 \oplus h_{2i}^1$ for $i = 1, 2, \ldots, t$ where $e_0^0 = 0$ and $e_0^1 = 0$.

3. For any t-bit string $a = a[1], a[2], \ldots, a[t]$, let $e^a = e_1^a, \ldots, e_t^a$.
4. D is the desired checksum to be imposed at the end of the 2^t 2-block multicollision.
5. Find $a = a[1], a[2], \ldots, a[t]$ such that $e_1^{a[1]} \oplus e_2^{a[2]} \oplus \ldots \oplus \ldots e_t^{a[t]} = D$. By treating $a[1], a[2], \ldots, a[t]$ as variables, solve the equation

$$\bigoplus_{i=1}^{t} e_i^0 \times a[i] \oplus e_i^1 \times (1 - a[i]) = D$$

6. Each bit position of $e_i^{a[i]}$ gives us one equation and turn the above into t equations, one for each bit. Let $\overline{a}[i] = 1 - a[i]$.
7. The resulting system is:

$$\bigoplus_{i=1}^{t} e_i^0[j] \times a[i] \oplus e_i^1[j] \times \overline{a}[i] = D[j] \ (j = 1, \ldots, t)$$

   Here there are t equations in t unknowns over modulo 2 addition which can be solved for the solution $a[1], a[2], \ldots, a[t]$.
8. The solution $a[1], a[2], \ldots, a[t]$ lets us determine the blocks in the 2^t 2-block multicollision that give the desired checksum D.

Work: It requires $t \times 2^{t/2}$ computations of the compression function to find a 2^t 2-block multicollision to construct the CCS and at most $t^3 + t^2$ bit-XOR operations to solve the system of $t \times t$ equations using Gaussian elimination to find a solution with a significant probability.

B.2 Defeating linear checksum in 3CM

The following algorithm is used to defeat the checksum in **3CM**.
ALGORITHM: Defeat linear checksum in 3CM
Variables:

1. (e_i^0, e_i^1) = A pair of independent choices of random values on the second chain after every 2-block collision in the 2^t 2-block multicollision on **3CM** and $e_i^0 \neq e_i^1$ for $i = 1, 2, \ldots, t$.
2. (s_i^0, s_i^1) = A pair of independent choices of random values on the third chain after every 2-block collision in the 2^t 2-block multicollision on **3CM** and $s_i^0 \neq s_i^1$ for $i = 1, 2, \ldots, t$.
3. $a = a[1], a[2], \ldots, a[t]$ and $c = c[1], c[2], \ldots, c[t]$ are any two t-bit strings.
4. $D_1 = D_1[1], D_1[2], \ldots, D_1[t]$ = The desired t-bit checksum to be imposed on the second chain.
5. $D_2 = D_2[1], D_2[2], \ldots, D_2[t]$ = The desired t-bit checksum to be imposed on the third chain.
6. i, j = Temporary variables.

Steps:

1. Build a CCS for **3CM** by building a 2^t 2-block multicollision by calling a brute-force collision finding algorithm $C(s, 2)$ t times on the Damgård-Merkle chain of **3CM**.
2. Each of the parts of the CCS gives one choice (e_i^0 or e_i^1) (resp. (s_i^0 or s_i^1)) for $i = 1, 2, \ldots, t$ on the second chain (resp. third chain) to determine some random t-bit value that either is or is not XORed into the final checksum value in the second chain (resp. third chain) at the end of the 2^t 2-block multicollision. Note that $e_i^j = H_{2i-1}^j \oplus H_{2i}^j$ for $i = 1, 2, \ldots, t$ where j is either 0 or 1 and $e_0^j = 0$. In addition, $s_i^j = H_{2i}^j \oplus \zeta((h'')_{2i-1}^j)$ for $i = 1, 2, \ldots, t$ where j is either 0 or 1 and $(h'')_0 = H_0$. At any iteration i, the checksum value in the third chain is $h_i'' = \zeta(h_{i-1}'') \oplus H_i$.

3. For any t-bit string $a = a[1], a[2], \ldots, a[t]$, let $e^a = e_1^a, \ldots, e_t^a$.
4. Now D_1 (resp. D_2) is the desired checksum to be imposed on the second chain (resp. third chain) at the end of the 2^t 2-block multicollision.
5. Find $a = a[1], a[2], \ldots, a[t]$ such that $e_1^{a[1]} \oplus e_2^{a[2]} \oplus \ldots \oplus \ldots e_t^{a[t]} = D_1$. Similarly, find $c = c[1], c[2], \ldots, c[t]$ such that $s_1^{c[1]} \oplus s_2^{c[2]} \oplus \ldots \oplus \ldots s_t^{c[t]} = D_2$.
   By treating $a[1], a[2], \ldots, a[t]$ as variables, we solve the equation
   $$\bigoplus_{i=1}^{t} e_i^0 \times a[i] \oplus e_i^1 \times (1 - a[i]) = D_1.$$
   Similarly, by treating $c[1], c[2], \ldots, c[t]$ as variables, we solve the equation
   $$\bigoplus_{i=1}^{t} s_i^0 \times c[i] \oplus s_i^1 \times (1 - c[i]) = D_2.$$
6. Let $\overline{a}[i] = 1 - a[i]$ and $\overline{c}[i] = 1 - c[i]$.
7. Now turn the above into two systems of t equations in t unknowns as below:
   The resulting two system of equations are:
   $$\bigoplus_{i=1}^{t} e_i^0[j] \times a[i] \oplus e_i^1[j] \times \overline{a}[i] = D_1[j] \ (j = 1, \ldots, t)$$
   $$\bigoplus_{i=1}^{t} s_i^0[j] \times c[i] \oplus s_i^1[j] \times \overline{c}[i] = D_2[j] \ (j = 1, \ldots, t)$$
   Here there are two sets of t equations in t unknowns which can be solved for two solutions $a[1], a[2], \ldots, a[t]$ and $c[1], c[2], \ldots, c[t]$.
8. The solution $a[1], a[2], \ldots, a[t]$ (resp. $c[1], c[2], \ldots, c[t]$) lets us determine the intermediate hash values of the second chain (resp. third chain) in the 2^t 2-block multicollision that gives the desired checksum D_1 (resp. D_2). Using these intermediate hash values, the data blocks that give the desired checksums can be found.

Work: It requires $t \times 2^{t/2}$ computations of the compression function to produce 2^t 2-block multicollision to construct the CCS and at most $2 \times (t^3 + t^2)$ work to solve the above two systems of $t \times t$ equations using Gaussian elimination to find a solution with a probability of 0.5 [3, Appendix A] [5, 43].

C Generic attacks on F-Hash, 3CM and Maelstrom-0

C.1 Long message 2nd-preimage attack on F-Hash

The attacker starts with a long target message for which she aims to find the 2nd preimage. Then he builds the CCS by constructing a 2^t 2-block multicollision to control the checksum, builds an *expandable message* and appends it to the CCS and then carries out the long message 2nd preimage attack from the end of the *expandable message*. The attacker then uses the CCS to adjust the accumulation chaining value at that point to match the desired value which is equivalent to adjusting the checksum of the 2^t 2-block multicollision. Finally, he expands the *expandable message* to make up for all the message blocks skipped in the long message 2nd preimage attack resulting in a new message which gives the same hash value as the long target message.
ALGORITHM: LongMessageAttack(M_{target}) on F-Hash
Find the 2nd preimage for a message of $2^d + d + 2t + 1$ blocks.
Variables:

1. M_{target} = The target long message for which a 2nd preimage is to be found.
2. M_{link} = Linking message block used to connect the iterative intermediate hash values at the end of the *expandable message* to some point in the sequence of the iterative intermediate hash values of the target message.
3. H_{exp} = The intermediate iterative hash values at the end of the *expandable message*.
4. H_t = The result of the 2^t 2-block multicollision on the iterative chain of H starting from the initial state H_0.
5. M_{final} = The 2nd preimage of the same length as M_{target} such that $H(M_{final}) = H(M_{target})$.
6. M_{pref} = The checksum control prefix obtained from the CCS to force the linear checksum to the desired checksum.

Steps:

1. Compute the intermediate hash values for M_{target} using H:
   - H_0 and h_0 are the initial states on the iterative and accumulation chains of H respectively.
   - M_i is the i^{th} message block of M_{target}.
   - $(H_i, h_i) = f(H_{i-1}, M_i)$ and $h'_i = \bigoplus_{j=1}^{i-1} h_j$ are the i^{th} intermediate hash values on the iterative and accumulation chains respectively.
   - The iterative and accumulation chaining states are organised in some searchable structure for the attack, such as hash table. The elements $H_1, \ldots, H_d$ and the elements obtained in the processing of t 2-block messages are excluded from the hash table as the *expandable messages* cannot be made short enough to accommodate them in the attack.
2. Build a CCS by constructing a 2^t 2-block multicollision on H as described in Section B.1 starting from the initial state H_0. Let H_t be the multicollision hash value. The corresponding checksum value h'_t due to the 2^t 2-block multicollision on H is random and its value depends on the choice of the t 2-block messages from the CCS that give the collision H_t.
3. Construct a $(d, d + 2^d - 1)$ *expandable message* M_{exp} with H_t as the starting intermediate state using the generic technique to find the *expandable messages*. Append the *expandable message* M_{exp} to the CCS. Let H_{exp} be the iterative intermediate hash value at the end of the *expandable message* M_{exp}.
4. Try different message blocks from the end of H_{exp} to find a linking message block M_{link} such that $f(H_{exp}, M_{link})$ matches some iterative intermediate hash value H_u stored in the hash table while processing M_{target}. Let this matching value of the target message be H_u and the corresponding accumulation chaining value be h'_u where $d + 2t + 1 \leq u \leq 2^d + d + 2t + 1$.
5. Use the CCS built in step 2 to find the checksum control prefix M_{pref} to adjust the accumulation chaining value at that point to match the desired accumulation value h'_u in the target message M_{target}. Using h'_u, the desired checksum value at the end of the 2^t 2-block multicollision is calculated and this value is adjusted in such a way that the desired checksum h'_u is obtained. The prefix M_{pref} is obtained by solving a system of $t \times t$ linear equations following Section 3.3.
6. Expand the *expandable message* to produce a message M^* which is $u - 1$ blocks long.
7. Return the 2nd preimage $M_{final} = M_{pref}||M^*||M_{link}||M_{u+1} \ldots M_{2^d+d+1+2t}$ of the same length as M_{target} such that $H(M_{final}) = H(M_{target})$.

Work: The computational effort required to perform the 2nd preimage attack on F-Hash is the same as the effort to find 2nd preimages for **3C**. Using the generic expandable-message finding algorithm, this effort equals $t \times 2^{t/2} + d \times 2^{t/2+1} + 2^{t-d+1}$ computations of the compression function and $t^3 + t^2$ bit XOR-operations.

C.2 Long message 2^{nd}-preimage attack on 3CM and Maelstrom-0

The attacker starts with a long target message for which he aims to find the 2^{nd} preimage. Then he builds the CCS by constructing a 2^t 2-block multicollision to control the checksum in the second and third chains of **3CM**, builds an *expandable message* and appends it to the CCS and then carries out the long message 2^{nd} preimage attack from the end of the *expandable message*. The attacker then uses the CCS to adjust the chaining values in the second and third chains at that point to match the desired checksum values in the second and third chains. This is equivalent to adjusting the checksum values of these two chains at the end of the 2^t 2-block multicollision. Finally, he expands the *expandable message* to make up for all the message blocks skipped in the long message 2^{nd} preimage attack producing a new message which produces the same hash value as the long target message. Since Maelstrom-0 uses **3CM** as the underlying module with a Davies-Meyer compression function, the long message 2^{nd} preimage attack on **3CM** is also applicable to Maelstrom-0.
ALGORITHM: LongMessageAttack(M_{target}) on 3CM
Find the 2^{nd} *preimage for a message of* $2^d + d + 2t + 1$ *blocks.*
Variables:

1. M_{target} = The target long message for which a 2^{nd} preimage is to be found.
2. M_{link} = Linking message block used to connect the iterative intermediate hash value at the end of the *expandable message* to some point in the sequence of the iterative hash values of the target message.
3. H_{exp} = The intermediate iterative hash value at the end of the *expandable message*.
4. H_t = The result of the 2^t 2-block multicollision on the iterative chain of H starting from the initial state H_0.
5. M_{final} = The 2^{nd} preimage of the same length as M_{target} such that $H(M_{final}) = H(M_{target})$.
6. M_{pref} = The checksum control prefix obtained from the CCS to force the linear checksum to the desired checksum.

Steps:

1. Compute the intermediate hash values for M_{target} using H:
   - H_0, h'_0 and h''_0 are the initial states on the iterative, second and third chains of H respectively.
   - M_i is the i^{th} message block of M_{target}.
   - $H_i = f(H_{i-1}, M_i)$, $h'_i = h'_{i-1} \oplus H_i$ and $h''_i = \zeta(h''_{i-1}) \oplus H_i$ are the intermediate chaining values at any iteration i in the iterative, second and third chains of **3CM** respectively.
   - The iterative, second and third chaining states are organised in some searchable structure for the attack, such as hash table. The elements $H_1, \ldots, H_d$ and the iterative intermediate hash values obtained in the processing of t 2-block messages are excluded from the hash table as the *expandable messages* cannot be made short enough to accommodate them in the attack.
2. Build a CCS by constructing a 2^t 2-block multicollision on H as described in Section B.2 starting from the initial state H_0. Let H_t be the multicollision intermediate hash value. The corresponding checksum values in the second and third chains denoted by h'_t and h''_t respectively due to the 2^t 2-block multicollision on H are random. Their values depend on the choices of the t 2-block messages from the CCS that produce the collision H_t.

3. Construct a $(d, d + 2^d - 1)$ *expandable message* M_{exp} with H_t as the starting state using either of the methods to find the *expandable messages* from [8, 18]. Append the *expandable message* M_{exp} to the CCS. Let H_{exp} be the iterative intermediate hash value at the end of the *expandable message* M_{exp}.
4. Try different message blocks from the end of H_{exp} to find a linking message block M_{link} such that $f(H_{exp}, M_{link})$ matches some iterative intermediate hash value H_u stored in the hash table while processing M_{target}. Let this matching value of the target message be H_u and the corresponding intermediate hash values in the second and third chains be h'_u and h''_u where $d + 2t + 1 \leq u \leq 2^d + d + 2t + 1$.
5. Use the CCS built in step 2 to find the checksum control prefix M_{pref} to adjust the chaining values in second and third chains at that point to match the desired checksum values h'_u and h''_u in the target message M_{target}. Using h'_u and h''_u, the desired checksum values in the second and third chains at the end of the 2^t 2-block multicollision are calculated and these values are adjusted in such a way that the desired checksums h'_u and h''_u are obtained. The prefix M_{pref} is obtained by solving a system of $t \times t$ linear equations as described in Section B.2.
6. Expand the *expandable message* to produce a message M^* which is $u - 1$ blocks long.
7. Return the 2nd preimage $M_{final} = M_{pref}||M^*||M_{link}||M_{u+1} \ldots M_{2^d+d+1+2t}$ of the same length as M_{target} such that $H(M_{final}) = H(M_{target})$.

Work: The effort required for the 2nd preimage attack on **3CM** involves the effort in finding a 2^t 2-block multicollision plus the effort in solving two sets of $t \times t$ system of linear equations plus the effort in finding the *expandable message* plus the effort to find the linking message block. So, the *only* additional effort in performing the 2nd preimage attack on **3CM** over Damgård-Merkle hash function is the effort required to solve two systems of $t \times t$ equations and producing a 2^t 2-block multicollision. Using the generic expandable-message finding algorithm, this effort equals $t \times 2^{t/2} + d \times 2^{t/2+1} + 2^{t-d+1}$ computations of the compression function and at most $2 \times (t^3 + t^2)$ bit-XOR operations.

C.3 Herding attack on F-Hash

The following steps outline the herding attack on a t-bit F-Hash hash function H:

1. A 2^d hash value wide diamond structure is constructed for H with 2^d different arbitrary states $H_1, H_2, \ldots, H_{2^d}$ as the starting iterative chain hash values. It is constructed by finding 1-block collisions similar to the construction of the diamond structure for the Damgård-Merkle hash functions. The final hash value H_f, which is the output of the compression function g, is computed using any of the possible 2^{d-1} checksum values or some value chosen arbitrarily. Let h'_c be that checksum value.
2. Build the CCS for H using a 2^t multicollision over 2-block messages as described in Section B.1. Let H_t be the 2^t 2-block multicollision value on the iterative chain of H.
3. When challenged with the prefix message P, process P using H_t as the starting intermediate hash value on the iterative chain. Let $H(H_t, P) = H_p$.
4. Find the linking message M_{link} such that the state $H(H_p, M_{link})$ matches one of the 2^d outermost intermediate hash values on the iterative chain in the diamond structure. If the match is compared to all the $2^{d+1} - 2$ intermediate hash values in the diamond structure then a $(1, d+1)$-*expandable message* must be produced at the end of the diamond structure ensuring that the final herded message is always a fixed length.

5. Use the CCS computed in step 2 to force the checksum of the herded message P to h'_c using the techniques as described in Section B.1 to defeat the checksum in the 2^t multicollision. Let M_{pref} be the checksum control prefix obtained after solving the system of equations due to the CCS.
6. Finally, output the message $M = M_{pref}||P||M_{link}||M_d$ where M_d are the message blocks which contributes in the construction of the diamond structure. The value $H(M)$ will be the same as the chosen target H_f.

Work: The effort to perform the herding attack on F-Hash is the effort required to build the CCS plus the effort to solve the system of equations due to the CCS plus the effort required to perform the herding attack on the Damgård-Merkle hash functions from [17]. This equals $t \times 2^{t/2} + 2^{t/2+d/2+2} + 2^{t-d}$ computations of the compression function and $t^3 + t^2$ bit-XOR operations assuming that only the outermost 2^d hash values are used for searching in the diamond structure. If all the $2^{d+1} - 2$ intermediate hash values are used for searching in the diamond structure then the effort required equals $t \times 2^{t/2} + 2^{t/2+d/2+2} + d \times 2^{t/2+1} + 2^{t-d-1}$ computations of the compression function and $t^3 + t^2$ bit-XOR operations.

C.4 Herding attack on 3CM and Maelstrom-0

The following steps outline the herding attack on a t-bit **3CM** hash function H. Since Maelstrom-0 uses **3CM** as the underlying module, this herding attack applies to Maelstrom-0 as well.

1. A 2^d hash value wide diamond structure is constructed for H with 2^d different arbitrary states $H_1, H_2, \ldots, H_{2^d}$ as the starting iterative chain hash values. The final hash value H_f, which is the output of the compression function g, is computed using any of the possible 2^{d-1} checksum values from the second and third chains or some values chosen arbitrarily. Let h'_c and h''_c be those checksum values in the second and third chains respectively.
2. Build the CCS for H using a 2^t multicollision over 2-block messages as described in Section B.2. Let H_t be the 2^t 2-block multicollision value on the iterative chain of H.
3. When challenged with the prefix message P, process P using H_t as the starting intermediate hash value on the iterative chain. Let $H(H_t, P) = H_p$.
4. Find the linking message M_{link} such that the state $H(H_p, M_{link})$ matches one of the 2^d outermost hash values values on the iterative chain in the diamond structure. If the match is compared to all the $2^{d+1} - 2$ intermediate hash values in the diamond structure then a $(1, d+1)$-*expandable message* must be produced at the end of the diamond structure ensuring that the final herded message is always a fixed length.
5. Use the CCS computed in step 2 to force the checksums of the herded message P in the second and third chains to h'_c and h''_c using the techniques described in Section B.1 to defeat the checksum in the 2^t multicollision. Let M_{pref} be the checksum control prefix obtained after solving the system of equations due to the CCS.
6. Finally, output the message $M = M_{pref}||P||M_{link}||M_d$ where M_d are the message blocks that contributed to the construction of the diamond structure. The value $H(M)$ will be the same as the chosen target H_f.

Work: The effort to perform the herding attack on **3CM** is the effort required to build the CCS plus the effort to solve the system of equations due to the CCS plus the effort required to perform the herding attack on the Damgård-Merkle hash functions from [17]. This equals $t \times 2^{t/2} + 2^{t/2+d/2+2} +$

2^{t-d} computations of the compression function and $2 \times (t^3 + t^2)$ bit-XOR operations assuming that only the outermost 2^d hash values are used for searching in the diamond structure. If all the $2^{d+1} - 2$ intermediate hash values are used for searching in the diamond structure then the effort required equals $t \times 2^{t/2} + 2^{t/2+d/2+2} + d \times 2^{t/2+1} + 2^{t-d-1}$ computations of the compression function and $2 \times (t^3 + t^2)$ bit-XOR operations.

D Additive checksum variant of GOST

D.1 3CA: A Variant of GOST hash function

We propose a variant for GOST called **3CA** which computes additive checksum using modular addition of intermediate hash values over 2^t as shown in Figure 10.

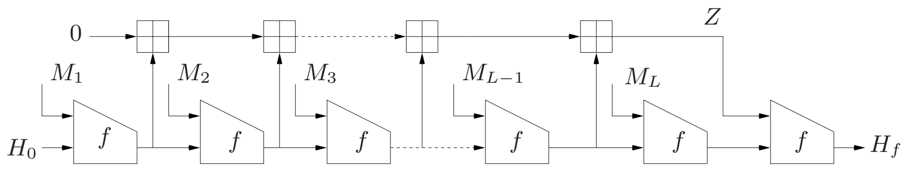


Fig. 10. 3CA-hash function

E Generic attacks on the hash functions with additive checksums

E.1 Long message 2nd-preimage attack on GOST

Long message 2nd preimage attack on a b-bit block, t-bit hash vaue GOST hash function H is outlined below.

ALGORITHM: LongMessageAttack(M_{target}) on GOST

Find the second preimage for a message of $2^d + d + b \times (b/2 + 1) + 1$ *blocks.*

Variables:

1. M_{target} = the target long message for which a second preimage is to be found.
2. M_{link} = the linking message block to connect the intermediate state at the end of the *expandable message* to some point in the sequence of intermediate states of the long target message.
3. H_{exp} = the intermediate state at the end of the *expandable message*.
4. H_t = the intermediate state at the end of the CCS.
5. M_{final} = the 2nd preimage of the same length as M_{target} such that $H(M_{final}) = H(M_{target})$.
6. M_c = the desired checksum block.
7. M_{pref} = the checksum control prefix obtained to force the checksum to the desired checksum.

Steps:

1. Compute the intermediate states for M_{target} using H:
   - H_0 is the initial state of H.
   - M_i is the ith message block of M_{target}.
   - $H_i = f(H_{i-1}, M_i)$ is the intermediate state on the iterative chain of H.

   - The iterative and accumulation chaining states are organised in some searchable structure for the attack, such as hash table. The elements $H_1, \ldots, H_d$ and those obtained in the processing of $b \times ((b/2)+1)$ 1-block messages are excluded from the table as the *expandable messages* cannot be made short enough to accommodate them in the attack.
2. Construct CCS for GOST following the method from Section 4 producing H_t as the multicollision value on the iterative chain and let M_t be the corresponding checksum block which is random.
3. Construct a $(d, d+2^d-1)$ *expandable message* M_{exp} from the end of H_t using generic-expandable message algorithm from [18]. Now H_{exp} is the intermediate state at the end of M_{exp}.
4. Try message blocks from the end of H_{exp} to find a linking message block M_{link} such that $f(H_{exp}, M_{link})$ collides with one of the intermediate states of M_{target} stored in the hash table. Let this matching intermediate state of the target message be H_u and the checksum of data blocks of M_{target} until that point be M_u where $d+b\times((b/2)+1)+1 \leq u \leq 2^d+d+b\times((b/2)+1)+1$.
5. Using M_u, find the desired checksum M_c at the end of CCS by working backwards from the point of match with the intermediate state of the target message.
6. Find the checksum control prefix M_{pref} which produces the desired checksum M_c at the end of the CCS while still maintaining collisions on the iterative chain using the algorithm to defeat the checksum in GOST as described in Section 4.
7. Expand the *expandable message* to produce a message M^* which is $u-1$ blocks long.
8. Return the second preimage $M_{final} = M_{pref}||M^*||M_{link}||M_{u+1}\ldots$ $M_{2^d+d+b\times(b/2)+1))+1}$ of the same length as M_{target} such that $H(M_{final}) = H(M_{target})$.

Work: The computational work required to find a second preimage on GOST is the work required to defeat the checksum to the target checksum following the algorithm in Section 4 plus the work to find the *expandable message* M_{exp} plus the work to find the linking message block M_{link}. So, the only additional work required to perform the second preimage attack on GOST over **DM** hash functions is the work to construct the CCS which is practically free to use. The total work equals $(b/2+1) \times b \times 2^{t/2} + d \times 2^{t/2+1} + 2^{t-d+1}$ computations of the compression function and a time and space of $b \times 2^{b/2+1}$. Note that the work to produce the multicollision to build the CCS and perform the modified collision search is very fast compared to the rest of the attack.
Illustration:
The computational work required to find a second preimage on GOST for a message of $2^{54} + 54 + 256 \times 129 + 1$ blocks is $2^{143} + 54 \times 2^{129} + 2^{203}$ computations of the compression function and a time and space of $256 \times 2^{129} = 2^{137}$.

E.2 Herding attack on GOST

The following steps outline the herding attack on a t-bit GOST hash function H:

1. Precompute a 2^d hash value wide diamond structure and output the hash value H_v as the chosen target. This value is computed using either any of the possible 2^{d-1} checksum values or some value chosen arbitrarily. Let M_c be that checksum value.
2. Construct the CCS for H starting from the initial state of the hash function H and let H_t be the intermediate state on the iterative chain at the end of CCS.
3. When challenged with the prefix message P, process P using H_t as the starting state on the iterative chain[3]. Let $H(H_t, P) = H_p$.

[3] We note that when the forced prefix message P is processed using the initial state H_0 of H followed by constructing CCS using the state $H(H_0, P)$, we can output message M with the format $P||M_{pref}||M_{link}||M_d$.

4. Find the linking message block M_{link} such that the state $H(H_p, M_{link})$ collides with one of the 2^d outermost intermediate states on the iterative chain in the diamond structure. If this collision is matched against all of the $2^{d+1} - 2$ intermediate states then a $(1, d+1)$-*expandable message* must be produced at the end of the diamond structure to make sure that the final herded message is always a fixed length.
5. Use the techniques from the algorithm in Section 4 to force the checksum of the herded message P to M_c. Let M_{pref} be the checksum control prefix which produces the desired checksum M_c.
6. Finally, output the message $M_{pref}||P||M_{link}||M_d$ where M_d are the message blocks in the diamond structure that connect the state $H(H_p, M_{link})$ to the chosen target H_v. Now $H(M) = H_v$.

Work:

The computational work required to perform the herding attack on GOST is the work required to construct the CCS plus the work to find the checksum control prefix plus the work required to perform the herding attack from [17]. This equals $b \times ((b/2)+1) \times 2^{t/2} + 2^{t/2+d/2+2} + 2^{t-d}$ computations of the compression function and a time and space of $b \times 2^{b/2+1}$ assuming that only the outermost 2^d states are used for searching in the diamond structure. If all the $2^{d+1} - 2$ intermediate states in the diamond structure are used for searching then the work required equals $b \times ((b/2)+1) \times 2^{t/2} + 2^{t/2+d/2+2} + d \times 2^{t/2+1} + 2^{t-d-1}$ computations of the compression function and a time and space of $b \times 2^{b/2+1}$. Note that the work required to build the CCS using multicollision and perform the modified collision search is negligible compared to the rest of the attack.

Illustration:

The computational work required to perform the herding attack on GOST with $d = 84$ is $2^{143} + 2^{172} + 2^{172}$ computations of the compression function assuming that only the outermost 2^{84} states in the diamond structure are used for searching and time and space of $256 \times 2^{129} = 2^{137}$.

E.3 Defeating the additive checksum in 3CA

We follow the steps below in order to defeat the additive checksum in **3CA** shown in Figure 10:

1. We perform a 2^t multicollision starting from the initial state H_0 of the hash function where each collision contains a $2^{(t/2)+1}$ multicollision performed over 2-block messages.
2. In every $2^{t/2+1}$ multicollision in the 2^t multicollision from $i = 1$ to t, we then search for a pair of $2 \times ((t/2)+1)$-chaining sequences, which we call as chunk, giving the same hash chaining output but checksum values that differ by 2^{i-1}. We perform this task as follows:
   – We initialize an empty table.
   – For every collision path in the $2^{t/2+1}$ 2-block multicollision:
     - We add to the table: the additive checksum of chaining values of that path, the collision path and an index of 0.
     - We add to the table: 2^{i-1} added to the additive checksum of the chaining values computed above, the collision path and an index of 1.
   – We then search for a match between the entries with index 0 and the entries with index 1.
3. We then obtain a workable CCS by concatenating such individual chunks all hashing to the same chaining state at the end of 2^t $2^{(t/2)+1}$ 2-block multicollision.
4. We then use this workable CCS to find the collision path which produces the checksum to the desired checksum at the end of the 2^t multicollision. The message blocks of this collision path would form the checksum control prefix.

Work: The work required to defeat the additive checksum in **3CA** is the work to construct the CCS plus the work required to use the CCS to force the additive checksum to the required checksum. This equals the work to find the 2^t $2^{(t/2)+1}$ 2-block multicollision plus the work to find the individual chunks in every $2^{(t/2)+1}$ 2-block multicollision. This equals about $t \times ((t/2)+1)) \times 2^{t/2}$ computations of the compression function plus a time and space of $t \times 2^{t/2}$.
Illustration: On **3CA** instantiated with the compression function of SHA-256, it requires about $2^8 \times 129 \times 2^{128} \approx 2^{143}$ computations of SHA-256 compression function and a time and space of $2^8 \times 2^{129} = 2^{137}$.

E.4 Generic attacks on 3CA

The generic algorithm given in Section 5 can be used to perform the 2nd preimage and herding attacks on the 3CA hash function. These attacks are similar to those on GOST and hence are left out from the discussion here. To find the 2nd preimage of a long target message of $2^{54} + 54 + 2(256)(129) + 1$ blocks processed using **3CA** based on the compression function of SHA-256, it requires $2^{143} + 54 \times 2^{129} + 2^{203}$ computations of the compression function and a time and space of $256 \times 2^{129} = 2^{137}$.

Similarly, the work to perform the herding attack on **3CA** instantiated with the compression function of SHA-256 using a 2^{84} hash value wide diamond structure (i.e width of the diamond is $d = 84$) is $2^{143} + 2^{172} + 2^{172}$ computations of the compression function assuming that only the outermost 2^{84} chaining values are used for searching in the diamond structure and time and space of $256 \times 2^{129} = 2^{137}$.

F Multi-block collision attacks on F-Hash and 3CA

F.1 Multi-block collision attack on F-Hash

Consider a collision finding algorithm $C(s, n)$ with the state $s = H_0$ for the F-Hash hash function H. For F-Hash, we define a collision for the compression function f at iteration i as finding two message blocks M_i and N_i such that $M_i \neq N_i$, $f(H_{i-1}, M_i) = f(H'_{i-1}, N_i) = (H_i, h_i)$ where either $H_{i-1} = H'_{i-1}$ or $H_{i-1} \neq H'_{i-1}$.

Let $n = 2$ and a call to $C(s, 2)$ results in a pair of messages (M, N) where $M = M_1||M_2$ and $N = N_1||N_2$ such that $H(H_0, M_1) = (H_1, h_1)$, $H(H_0, N_1) = (H_1^*, h_1^*)$, $H_1 \oplus H_1^* = \varDelta_H$, $h_1 \oplus h_1^* = \varDelta_h$ and $H(M) = H(N) = (H_2, h_2)$. Now a second call to $C(s, 2)$ with $s = H_2$ results in two pairs of blocks (M_3, M_4) and (N_3, N_4) such that $H(H_2, M_3) = (H_3, h_3)$, $H(H_2, N_3) = (H_3^*, h_3^*)$, $H_3 \oplus H_3^* = \varDelta_H$, $h_3 \oplus h_3^* = \varDelta_h$ and $H(H_3, M_4) = H(H_3^*, N_4) = (H_4, h_4)$. This is depicted in Figure 11.

Since, $H_1 \oplus H_1^* = \varDelta_H$ and $H_3 \oplus H_3^* = \varDelta_H$, $H_1 \oplus H_2 \oplus H_3 \oplus H_4 = H_1^* \oplus H_2 \oplus H_3 \oplus H_4^*$, a collision on the iterative chain of F-Hash. In addition, since, $h_1 \oplus h_1^* = \varDelta_h$ and $h_3 \oplus h_3^* = \varDelta_h$, $h_1 \oplus h_2 \oplus h_3 \oplus h_4 = h_1^* \oplus h_2 \oplus h_3 \oplus h_4^*$, a collision on the accumulation chain of F-Hash.

F.2 Multi-block collision attack on 3CA

Consider a collision finding algorithm $C(s, 2)$ which finds a collision on the iterative chain of **3CA** after processing every two blocks starting from $s = H_0$. Let $f(H_0, M_1) = H_1$, $f(H_0, N_1) = H_1^*$, $f(H_1, M_2) = f(H_1^*, N_2) = H_2$ be the intermediate hash values after processing single blocks from the state H_0. Let $f(H_2, M_3) = H_3$, $f(H_2, N_3) = H_3^*$, $f(H_3, M_4) = f(H_3^*, N_4) = H_4$ be the intermediate

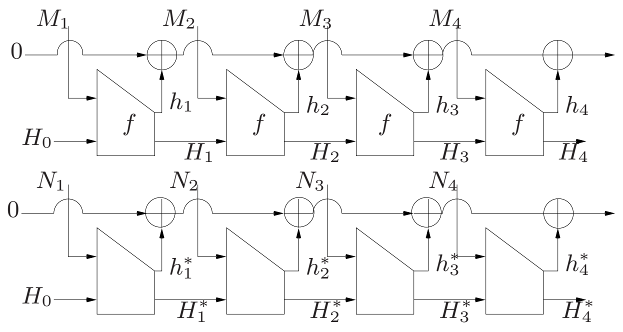


Fig. 11. Multi-block collision attack on F-Hash

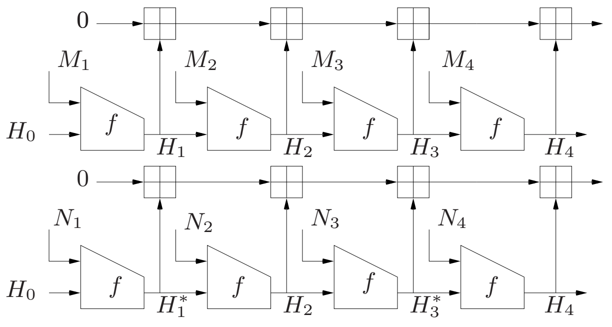


Fig. 12. Multi-block collision attack on the 3CA-hash function

hash values after processing single blocks from the state H_2. Assume the additive differences of the intermediate intermediate hash values as $H_1^* - H_1 \equiv \Delta \bmod 2^t$ and $-H_3^* + H_3 \equiv \Delta \bmod 2^t$. Now consider the checksum value $H_1 + H_2 + H_3 + H_4 \bmod 2^t = H_1^* - \Delta + H_2 + H_3^* + \Delta + H_4 \bmod 2^t = H_1^* + H_2 + H_3^* + H_4$. This is a collision for the checksum. Hence, concatenation of two structured 2-block collisions with the additive differences as described above would produce a multi-block collision for the **3CA** structure.