On hash functions using checksums

Gauravaram, Praveen; Kelsey, John; Knudsen, Lars Ramkilde; Thomsen, Søren Steffen

Publication date:
2008

Document Version
Early version, also known as pre-print

Link back to DTU Orbit

Citation (APA):
Gauravaram, P., Kelsey, J., Knudsen, L. R., & Thomsen, S. S. (2008). *On hash functions using checksums*. MAT report No. 2008-06

On hash functions using checksums *

Praveen Gauravaram[1**], John Kelsey[2], Lars Knudsen[1], and Søren Thomsen[1***]

[1] DTU Mathematics, Technical University of Denmark, Denmark
p.gauravaram@mat.dtu.dk,lars.r.knudsen@mat.dtu.dk,mail@znoren.dk
[2] National Institute of Standards and Technology (NIST), USA
john.kelsey@nist.gov

Abstract. We analyse the security of iterated hash functions that compute an input dependent checksum which is processed as part of the hash computation. We show that a large class of such schemes, including those using non-linear or even one-way checksum functions, is not secure against the second preimage attack of Kelsey and Schneier, the herding attack of Kelsey and Kohno, and the multicollision attack of Joux. Our attacks also apply to a large class of cascaded hash functions. Our second preimage attacks on the cascaded hash functions improve the results of Joux presented at Crypto'04. We also apply our attacks to the MD2 and GOST hash functions. Our second preimage attacks on the MD2 and GOST hash functions improve the previous best known short-cut second preimage attacks on these hash functions by factors of at least 2^{26} and 2^{54}, respectively.

Keywords: Iterated hash functions, checksums, multicollisions, second preimage and herding attack.

1 Introduction

Cryptographic hash functions are important tools of cryptology used in many applications for secure and efficient information processing. In theory, hash functions are expected to resist collision and (second) preimage attacks to provide security in such applications. Furthermore, they are often assumed to behave "randomly". In practice, the requirement of these security properties depends entirely on the application.

For efficiency reasons, iterated hash functions are used in many real-life applications. The Merkle-Damgård construction [3, 20] (**MD** construction in the rest of this article) is the most commonly used iterated hash function framework to design hash functions; almost all widely used hash functions follow this framework. Given a fixed length input collision resistant compression function, a variable length input collision resistant hash function can be designed using this framework.

A number of so-called generic attacks on the **MD** construction have shown that this construction does not satisfy some expected security properties. This structure is not as resistant to second preimage attacks as an ideal hash function as shown by Kelsey and Schneier [14] and it also has other undesired weaknesses in the form of multicollision [11] and herding attacks [13].

In this paper, we study the security of iterated hash functions that use checksums with respect to the generic attacks of [11,13,14] on the **MD** construction. This class of hash functions use checksums in much the same way as the compression function. For a long time, checksums have been sought for use in iterated hash functions aiming to increase the security of the overall hash function without degrading its efficiency significantly. Hence, it is desirable that a checksum is at least as fast as the compression function itself. Applications may even require checksum to be much faster than the

[*] Part of the results of this paper were published in [7].

[**] Author is supported by the Danish Research Council for Technology and Innovation grant number 274-08-0052.

[***] Author is supported by the Danish Research Council for Technology and Production Sciences, grant no. 274-05-0151.

compression function so that the speed of the total hash function construction approximates that of the hash function without checksum. The hash function proposal of Quisquater and Girault [27] which uses an additive checksum and the MD2 hash function of Rivest [28] which uses a non-linear checksum are some earlier checksum based hash functions.

In this class of hash functions, a checksum is computed using the message and/or intermediate hash values and subsequently appended to the message, which is then processed using the hash function. If H is a **MD** hash function then we define a checksum-based hash construction $\tilde{H}$ to process a message m by $\tilde{H}(m) = H(m \| C(m))$, where C is the checksum function computed over m. Our notation is slightly abusive, as padding of the message is ignored. However, the attacks presented later in this paper do not rely on any specific padding rule, and hence we shall be using this notation repeatedly. We let the sizes of the hash and checksum states be d and n bits, respectively. The hash value of $\tilde{H}$ is also n bits. The function C could be as simple as an XOR of its inputs as in 3C [8], a modular addition as in GOST [30], a simple non-linear function as in MD2 [12] or some complex one-way function such as the SHA-1 [23, 25] compression function. An example construction using SHA-1 as the checksum function for SHA-256 hash function is defined by $\mathsf{SHA256}(m \| \mathsf{SHA1}(m))$, which we shall return to later.

In this paper, we analyse a general class of checksum-based hash functions that can be defined as $\tilde{H}(m) = H(m \| C(m))$ with respect to the generic attacks of [11, 13, 14] where the checksum function C can be linear, non-linear or even one-way (hard to invert: inverting C requires about 2^d evaluations). Our attacks on the construction $\tilde{H}$ are summarised as follows:

1. The construction $\tilde{H}$ with any checksum including one-way checksum succumbs to the long message second preimage attack of [14] and herding attack of [13] whenever $d < n$.
2. More efficient second preimage and herding attacks work on the construction $\tilde{H}$ having a linear or non-linear but easily invertible C (C may be inverted in 1 evaluation) whenever $d < 2n - 2$.
3. Multicollisions on the construction $\tilde{H}$ for any checksum function when $d \leq n$. If the checksum function is easily invertible, multicollisions can be found for $\tilde{H}$ independently of the size of d.

We note that a checksum-based hash function can also be viewed as a cascaded construction in which the hash values of hash functions in the cascade are combined and processed in the end. Hence, our attacks on the construction $\tilde{H}$ also apply to the cascade of two independent hash functions H and G defined by $H(m) \| G(m)$ [26]. Similarly, our attacks easily extend to complex looking cascaded structures of form $H(m) \| G(m \| H(m))$ [11, Section 4.3]. Our techniques combined with the long message second preimage attack of [14] improve the previous second preimage attack of Joux [11] on the cascade hash functions whose complexity is upper bounded by the brute force second preimage attack on the strongest hash function (the one with the largest hash value) in the cascade. This result also solves the open question of Dunkelman and Preneel [5] on the application of the long message second preimage attack of [14] on the cascaded hashes.

Our multicollision attacks on the cascaded hash functions complement the results of [11] and [5], which show that such schemes are only as collision resistant as the strongest of the individual hash functions in the cascade. Dunkelman and Preneel [5] generalised the herding attack of [13] on the cascaded hash functions. We note that this attack is also applicable to the checksum based hash function $\tilde{H}$ since by using the compression function of one of the hash functions in the cascade as a checksum function, a cascaded hash function can be turned into the checksum based hash $\tilde{H}$. Hence, this herding attack works on $\tilde{H}$ with any C even when $d \leq n$.

Our results also have implications to the security of the 128-bit MD2 hash function [12] developed by Rivest and the Russian standard GOST hash function [30]. Given a target message of 2^{64} 128-bit blocks, we find second preimages for MD2 in 2^{71} evaluations of its compression function improving the previous work of $2^{97.6}$ based on a preimage attack [15]. Similarly, for a target message of 2^{128} 256-bit blocks, our techniques can find second preimages for GOST in about 2^{137} evaluations of its compression function in comparison to the previous short-cut attack of 2^{192} [19].

We summarise the state of art of hash functions using checksums in Table 1 by combining our results with those of [5].

Table 1. Attacks on hash functions using checksums.

Attack on checksum-based hash $\tilde{H}$	One-way C of size d	Easily invertible C of size d
Long message second preimage attack of [14]	$d < n$	$d < 2n - 2$
Herding attack of [13]	$d \leq n$	$d < 2n - 2$
Multicollision attack of [11]	$d \leq n$	d of any size

The long message second preimage and herding attacks on some specific checksum based hash functions are summarised in Tables 2 and 3. Our second preimage attacks on the MD2 and GOST hash functions are faster than their previous best known attacks for only the challenged target messages of sizes at least as large as indicated in Column 2 of Table 2. The other hash functions in Table 2 specify an upper bound on the length of the messages (at most $2^{64} - 1$ bits) to be hashed close to the values as indicated in Column 2. Table 3 summarises the first ever analytical results of the herding attack on these hash functions. Our second preimage and herding attacks require abundant memory as shown later in Section 6 of the paper.

Table 2. Second preimage attacks on specific hash functions.

Hash function	Our complexity	Known complexity	Ideal complexity
MD2	2^{71} for a target of 2^{64} 128-bit blocks	$2^{97.6}$ [15]	2^{128}
GOST	2^{191} for a target of 2^{65} 256-bit blocks	2^{192} [19]	2^{256}
$\mathsf{SHA256}(m\|\mathsf{SHA1}(m))$	2^{202} for a target of 2^{54} 512-bit blocks	2^{256} [8]	2^{256}
3C-SHA-256	2^{202} for a target of 2^{54} 512-bit blocks	2^{256} [8]	2^{256}
MAELSTROM-0	2^{202} for a target of 2^{54} 512-bit blocks	2^{256} [9]	2^{256}

Table 3. Herding attacks on specific hash functions.

Hash function	Our complexity	Ideal complexity
MD2	2^{87}	2^{128}
GOST	2^{172}	2^{256}
$\mathsf{SHA256}(m\|\mathsf{SHA1}(m))$	2^{172}	2^{256}
3C-SHA-256	2^{172}	2^{256}
MAELSTROM-0	2^{172}	2^{256}

1.1 Related work

Coppersmith [2] presented a short-cut collision attack on a 128-bit hash function based on the DES algorithm proposed by Quisquater and Girault [27] which processes two supplementary checksum blocks each computed independently using XOR and modular addition of message blocks. In unpublished work, Mironov and Narayanan (personal communication at Crypto'06, August, 2006) developed a different technique to defeat XOR checksums in hash functions; this technique is less flexible than ours, and does not work for long-message second preimage attacks. In [11], Joux provides a technique for finding 2^k collisions for a **MD** hash function for only about k times as much work as is required for a single collision, and uses this technique to attack cascade hashes. Nandi and Stinson [22] have shown the applicability of multicollision attacks to a variant of **MD** in which each message block is processed multiple times; Hoch and Shamir [10] extended the results of [22] showing that generalised sequential hash functions with any fixed repetition of message blocks do not resist multicollision attacks.

The MD2 hash function [12] uses a non-linear checksum function. Vulnerabilities in MD2 have been exposed through collision attacks on its compression function [15, 29] and (second) preimage attacks on the full hash function [15, 21] by exploiting weaknesses in its compression function. The technique of Dunkelman and Preneel [5] to herd cascaded hash functions is more generic than our herding attack on the cascaded hashes, but it cannot be used to carry out the second preimage attack of [14] on these hash functions. Mendel, Pramstaller and Rechberger [19] presented a short-cut (second) preimage attack on the 256-bit GOST hash function for a target message of at least 257 message blocks in 2^{192} evaluations of the compression function.

1.2 Guide to the paper

In Section 2, we give an overview of cryptographic hash functions. In Section 3, we discuss hash functions using checksums. In Sections 4 and 5, we introduce cryptanalytic tools that we later use in Section 6 to perform generic attacks on the checksum based hash functions. In Section 7, we apply our attacks to the cascaded hash functions. In Section 8, we discuss the application of cryptanalytic collision attacks on the hash functions to carry out generic attacks on the checksum based hash functions. In Section 9, we compare the approach of Mironov-Narayanan to defeat XOR checksums in hash functions with our approach. In Section 10, we conclude the paper with some open questions.

2 Cryptographic hash functions

Cryptographic hash functions process an arbitrary length message into a fixed length hash value. Let $H : \{0,1\}^* \rightarrow \{0,1\}^n$ be an n-bit hash function. Three well-known attacks on H are:

- **Collision attack:** Find m, m^*, such that $m \neq m^*$ and $H(m) = H(m^*)$.
- **Preimage attack:** Given $Y = H(m)$, find m^* such that $H(m^*) = Y$.
- **Second preimage attack:** Given m and $H(m)$, find m^* such that $m \neq m^*$ and $H(m^*) = H(m)$.

For an ideal H, the complexities of the collision and (second) preimage attacks are $2^{n/2}$ and 2^n respectively. These complexities are measured in terms of evaluations of H. These attacks are generic in the sense that they apply to any hash function independently of how it is designed.

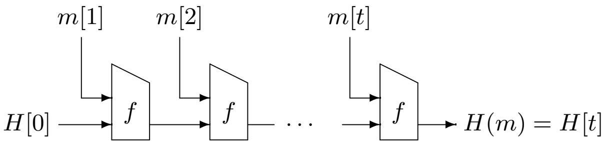

Fig. 1. The **MD** construction.

2.1 Merkle-Damgård hash functions

Most iterated hash functions used in practice such as MD5 [28], SHA-1 [24] and SHA-256 [23, 25] follow the **MD** hash function framework. An n-bit **MD** hash function H, as illustrated in Figure 1, works as follows: Assume that H can process a message m with a maximum length of 2^N bits. The message is padded to a length which is a multiple of b in bits. The padding includes the binary encoding of the length of the original message in the last N bits. This technique of including the length of the message in the padding is known as **MD** strengthening [16]. The message m is then partitioned into t message blocks $m[i]$, $1 \leq i \leq t$, each b bits long. Each block is processed using a fixed input length compression function $f : \{0,1\}^n \times \{0,1\}^b \to \{0,1\}^n$. The compression function iteration is defined by

$$H[i] = f(H[i-1], m[i]),$$

where $H[0]$ is the initial value (IV) of H. The values $H[i]$, $0 < i < t$, are called *intermediate hash values*. The value $H[t]$ is the hash value $H(m)$. The hash construction H preserves the collision resistance of the compression function f due to the inclusion of the **MD** strengthening. It requires about $2^{n/2}$ evaluations of f to find a collision and 2^n evaluations of f to find (second) preimages for an ideal n-bit compression function f.

The following attacks apply to an n-bit **MD** hash function H and are generic to the iterated hash functions even if their compression functions are ideal:

1. **2^k-collision attack where $k \geq 1$** [11]: Find a 2^k-set of messages $\{m[1], m[2], \ldots, m[2^k]\}$ such that $m[i] \neq m[j]$ whenever $i \neq j$, and $H(m[1]) = \ldots = H(m[2^k])$. This attack takes about $k2^{n/2}$ evaluations of f.
2. **Long message second preimage attack** [14]: Given a message m of approximately 2^k message blocks and $H(m)$, find m^* such that $m \neq m^*$ and $H(m^*) = H(m)$. This attack requires about $k2^{n/2} + 2^{n-k}$ evaluations of f.
3. **Herding attack** [13]: Construct a binary tree structure for H with 2^k random hash values at the leaves, and the hash value H_{T} at the root. For each node in the tree there is a message block that maps the hash value at that node to the hash value at the parent. Commit to H_{t}. Later, when some relevant information m' is available, construct a message m using m', the precomputed tree structure and some online computations such that $H(m) = H_{\mathrm{t}}$. It takes about $2^{n/2+k/2+2}$ offline evaluations of f to compute the tree structure and 2^{n-k} evaluations of f to construct m.

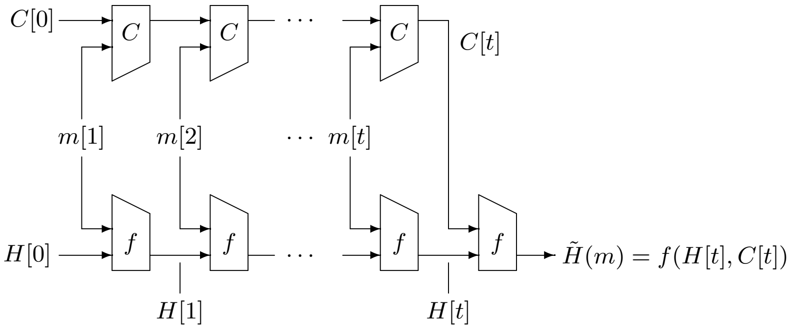

Fig. 2. The **MD** construction with checksum.

3 Hash functions using checksums

Hash functions that use checksums process an arbitrary length message using two chains: a *hash chain* where a compression function $f : \{0,1\}^n \times \{0,1\}^b \to \{0,1\}^n$ is iterated, and a *checksum chain* where a checksum function $C : \{0,1\}^d \times \{0,1\}^b \to \{0,1\}^d$ is iterated, as illustrated in Figure 2. Typical values of d are b or n. If $d < b$, the checksum will be padded to a length of b bits. The checksum function iteratively updates a checksum state using message blocks, intermediate hash values or both as input, in much the same way as the compression function updates an intermediate hash value. In the end, when the entire message has been processed in both the chains, the checksum is processed as an additional message block in the hash chain.

For an easy presentation of analysis later in the paper, we consider schemes that compute checksums using only message blocks. In fact, this implies no loss of generalisation if one assumes that the checksum function knows the IV of the hash chain, since the checksum function may then perform the exact same computations as those done in the hash chain. As a result, our analysis can be easily modified to the case of checksum computation using intermediate hash values, or both message blocks and intermediate hash values. Wherever applicable, we remark these extensions in this paper. In addition, for ease of exposition, we limit the discussion to checksums of size at most one message block – hence, $d \leq b$. Our attacks described later in the paper also work when $d > b$ (but the complexity of the attack often depends on the concrete value of d).

Let H be an n-bit hash function iterated over the compression function f. Now an n-bit checksum-based hash function may be described as

$$\tilde{H}(m) = H(m \| C(m)) = f(H(m), C(m)).$$

We repeat that this is a slight abuse of notation due to padding, but we ignore this difference in the following. In the description of the attacks, we do not care about padding, except that in the second preimage attack we make sure that the second preimage produced has the right length (same as the first preimage). It is a simple matter to account for padding in our attacks.

3.1 XOR/additive checksum variants of MD

A number of variant constructions have been proposed, that augment the **MD** construction by computing some kind of XOR/additive checksum on the message bits and/or intermediate hash values, and providing the XOR/additive checksum as a final block for the hash function. In the following sections we define some XOR/additive checksum variants of **MD**that have appeared in the literature.

3C hash function and its variants. The 3C construction maintains twice the size of the hash value for its intermediate states using a hash chain and a checksum chain as shown in Figure 3. In its hash chain, a compression function f with a block size b is iterated in the **MD** mode. In its checksum chain, the checksum $C[t]$ is computed by XORing all the intermediate states each of size n bits. The construction assumes that $b > n$. At any iteration i, the checksum value is $\bigoplus_{j=1}^{i} H_j$. The hash value H_{t} is computed by processing $C[t]$ padded with 0 bits to make the final data block $\mathrm{pad}(C[t])$ using the last compression function.

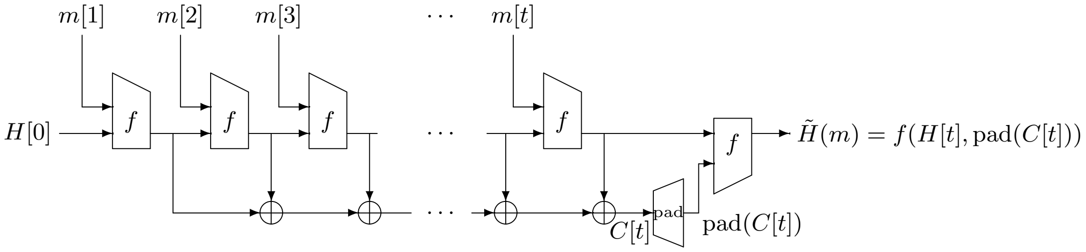

Fig. 3. The 3C-hash function.

A 3-chain variant of 3C called 3CM is used as a chaining scheme in the Maelstrom-0 hash function [6]. At every iteration of f in the hash chain of 3CM, the n-bit value in the third chain is updated using an LFSR. This result is then XORed with the data in the hash chain at that iteration. All the intermediate hash values in the hash chain of 3CM are XORed in the 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 [17], 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.

GOST hash function. GOST is a 256-bit hash function specified in the Russian standard GOST R 34.11 [30]. The compression function f of GOST is iterated in the **MD** mode and a mod 2^{256} additive checksum is computed by adding all the 256-bit message blocks in the checksum chain.

An arbitrary length message m to be processed using GOST is split into b-bit blocks $m[1], \ldots, m[t-1]$. If the last block $m[t-1]$ 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 $m[t]$ as shown in Figure 4. At any iteration i, the intermediate hash value in the hash chain and checksum chains are $H[i] = f(H[i-1], m[i])$ where $1 \leq i \leq t$ and $m[1] + m[2] \ldots + m[i] \bmod 2^b$ respectively where $1 \leq i \leq t-1$. The hash value of m is $H[v] = f(H[t], C[t-1])$ where $C[t-1] = m[1] + m[2] \ldots + m[t-1] \bmod 2^b$.

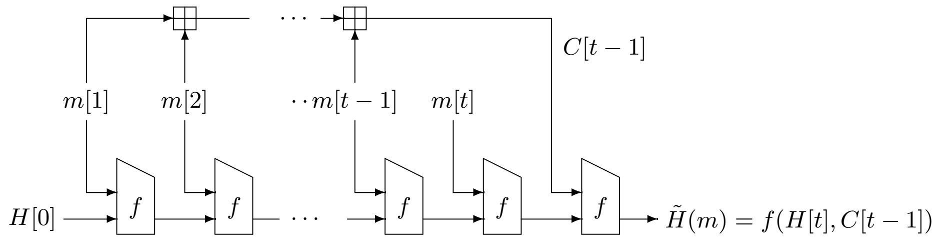

Fig. 4. The GOST hash function.

3.2 Our example one-way checksum variant of MD

We now define a 256-bit checksum-based hash function which will be used as a running example throughout this paper. We stress that this is *not* a proposal.

Definition 1. *The checksum-based hash function* $\tilde{\mathbf{H}}$ *is defined as*

$$\tilde{\mathbf{H}}(m) = \mathsf{SHA256}(m \| \mathsf{SHA1}(m)).$$

This construction may be seen as a checksum-based extension of the SHA-256 hash function, where the checksum function is the SHA-1 compression function. We assume that the compression functions of SHA-256 and SHA-1 take similar amounts of time to evaluate. The hash functions SHA-256 and SHA-1 hash messages of a maximum length of $2^{64}-1$ bits [23, 25], which is almost close to 2^{55} 512-bit blocks. Hence, while illustrating our attacks, we often consider that these hash functions hash messages of about 2^{55} 512-bit blocks.

3.3 Some definitions

At every iteration i in the hash computation of $\tilde{H}(m)$, the checksum function C updates a checksum state as given by $C[i] = C(C[i-1], m[i])$. We measure the number of evaluations of the checksum function with a time parameter T. We use the following definitions later in the paper:

Definition 2. *Inverting* C*: The process of computing* $C[i-1]$ *given* $C[i]$ *and* $m[i]$ *in time* T*.*

Definition 3. *Linking* C*: The process of computing* $m[i]$ *given* $C[i]$ *and* $C[i-1]$ *in time* T*.*

For example, the XOR checksum function in 3C, 3CM and F-Hash and the additive checksum function in GOST are invertible and linkable in time $T = 1$. The non-linear checksum function of MD2 is also invertible and linkable in time $T = 1$ [15, 21]. In addition, the invertible non-linear checksum of MD2 can be computed in time equivalent to about $1/52 \approx 2^{-5.7}$ compression function evaluations. We shall use these facts later in the paper. We shall also often assume that C and f take a similar amount of time to evaluate.

4 Checksum control sequences

We will use a cryptanalytic tool which we call as *checksum control sequence* (CCS) to extend a generic attack on an iterated hash function H to the checksum-based hash construction $\tilde{H} = H(m \| C(m))$. We define CCS as a data structure which lets us control the checksum value of $\tilde{H}$, *without altering the rest of the hash computation.* We construct the CCS by building a Joux multicollision [11] of the correct size using a brute-force collision search. It is important to note that the CCS is not itself a single string which is hashed; instead, it is a data structure which permits us to construct one of a very large number of possible strings, each of which has some effect on the checksum, but leaves the remainder of the hash computation unchanged. That is, the choice of a piece of the message from the CCS affects the checksum chain, but not the hash chain, of the $\tilde{H}$ hash. Thus, CCS enables us to produce a message with any desired checksum, such that the checksum has, in effect, been defeated and allows the generic attack to work on the hash function $\tilde{H}$.

The CCS has two algorithms associated with it; a CCS construction algorithm and a CCS searching algorithm. Our CCS algorithms are generic and can defeat checksum values computed using a wide range of checksum functions. We also present very specific CCS algorithms that can be used to defeat only simple checksums such as linear checksums. The complexities to construct and use the specific CCS algorithms to defeat the linear checksums are sometimes better than the generic CCS algorithms to defeat linear checksums; for example, defeating XOR checksums as in 3C using our specific CCS algorithm is faster than the generic one.

4.1 Constructing the CCS

Given some intermediate hash value H_{iv}, the CCS for $\tilde{H}$ is constructed as follows.
Algorithm.

1. Produce a 2^d-collision over 1-block messages on the hash chain of $\tilde{H}$ with H_{iv} as its IV using the multicollision method of [11] ignoring its impact on the checksum chain[3].
2. The checksum computed over at least one of the 2^d messages in the multicollision for H is expected to be equal to some (at this point unknown) target checksum value C_{t}. All 2^d messages in the CCS produce the same intermediate hash value H_{CCS}.

Complexity. Constructing the CCS takes about $d \times 2^{n/2}$ evaluations of the compression function.

Remark 1. If the checksum in $\tilde{H}$ is computed using intermediate hash values then a 2^d-collision over multi-block messages (e.g., 2-block messages) on its hash chain has to be produced to construct the CCS.

4.2 Searching the CCS

Once a CCS is constructed, a generic attack may be carried out on H from the end of the CCS. To extend this generic attack onto the construction $\tilde{H}$, a message in the CCS with the right checksum C_{t} must be found by searching the CCS. We provide the following two techniques to find such a message depending on the difficulty of inverting the checksum function C.

[3] The checksum function need not be evaluated while the CCS is being constructed and it is sufficient to evaluate the checksum function while searching the CCS.

Exploiting the tree structure of messages. If C is not easily invertible, e.g. SHA-1 in $\tilde{\mathbf{H}}$, we can exploit the tree structure of the messages in the CCS to find the right message. This takes about 2^{d+1} evaluations of the checksum function C. This is a direct application of Joux' collision attack on the cascaded hash functions [11].

Meet-in-the-middle attack. If C is easily invertible, as the C in MD2, GOST and 3C, one may perform a meet-in-the-middle attack on the message blocks in the CCS as follows:
Algorithm. Let the initial checksum be C_{iv}, and the target checksum be C_{t}.

1. Compute all $2^{d/2}$ intermediate checksum values from the first $d/2$ message blocks in the CCS, starting from C_{iv}. Store all these values in the list L_1.
2. Invert the checksum function C from C_{t}, using the last $d/2$ message blocks in the CCS. This produces $2^{d/2}$ intermediate checksum values. Store all these values in the list L_2.
3. Find a collision between L_1 and L_2. This collision corresponds to a message in the CCS which produces the checksum C_{t}.

Complexity. This attack takes about $2^{d/2+1}$ evaluations of the checksum function C and requires enough memory to store $2^{d/2+1}$ d-bit checksum values.

To generalise, if the checksum function is invertible in time $T = 2^{\tau}$, then searching the CCS for the right message can be done in time about $2^{(d+\tau)/2+1}$, by placing the "middle" in the meet-in-the-middle attack at the $(d/2 + \tau/2)$-th block (note that $0 \leq \tau \leq d$). The attack requires around $2^{(d-\tau)/2+1}$ memory, i.e., a maximum of $2^{d/2+1}$ when $\tau = 0$ (as above) and negligible memory when $\tau = d$.

5 CCS algorithms to defeat linear checksums

Specific CCS algorithms can be constructed and used to defeat XOR and additive checksums. The XOR and additive checksums can be controlled efficiently using specific algorithms than the generic algorithms described in Section 4.2.

For example, a 2^k-collision on the underlying **MD** construction of 3C, in which the sequence of individual collisions, each two message blocks long, gives us a choice of 2^k different sequences of message blocks that might appear at the beginning of this message. When we want a particular k-bit checksum value, we 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, which can be done very efficiently using existing tools such as Gaussian elimination [1, Appendix A], [31]. This is shown in Figure 5 for $k = 2$ where we compute the CCS by finding a 2^2 collision using random 2-block messages. Then we have a choice to choose either $H[1] \oplus H[2]$ or $H^*[1] \oplus H[2]$ from the first 2-block collision and either $H[3] \oplus H[4]$ or $H^*[3] \oplus H[4]$ from the second 2-block collision of the CCS to control 2 bits of the checksum without changing the hash value after the CCS.

5.1 Defeating XOR checksums in hash functions

We now explain how to defeat the XOR checksum of 3C. Let C_{t} be the desired value of the checksum.
Algorithm.

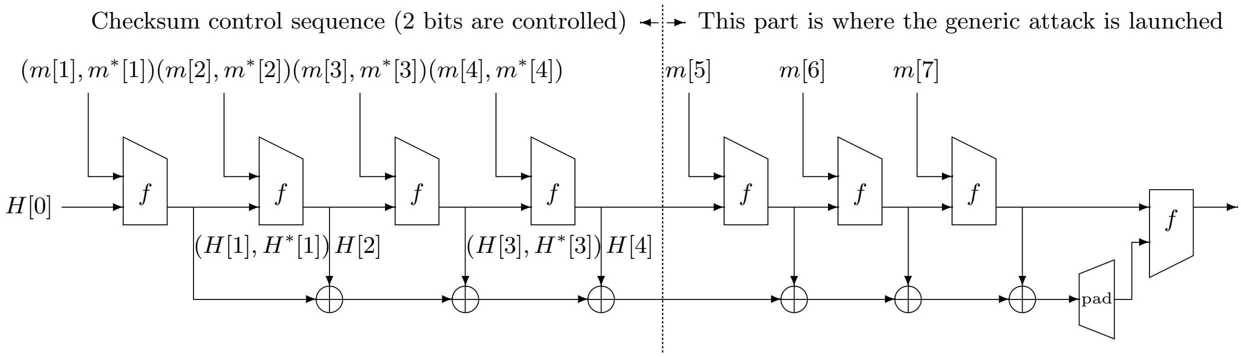
Checksum control sequence (2 bits are controlled)
This part is where the generic attack is launched
$(m[1], m^*[1])(m[2], m^*[2])(m[3], m^*[3])(m[4], m^*[4])$
$m[5]$
$m[6]$
$m[7]$
$H[0]$
f
$(H[1], H^*[1])$
$H[2]$
$(H[3], H^*[3])$
$H[4]$
pad

Fig. 5. Using the CCS to control 2 bits of the checksum (3C).

1. Build a CCS for 3C by constructing a 2^n-collision consisting of two-block ordinary collisions on the underlying **MD** construction. The $2n$ message block pairs in the CCS are denoted $(m[i], m^*[i])$, $1 \leq i \leq 2n$, and the corresponding intermediate hash values are denoted $H[i]$ and $H^*[i]$ (we have $H[i] \neq H^*[i]$ for i odd, and $H[i] = H^*[i]$ for i even).
2. Let $e_i^0 = H[2i-1] \oplus H[2i]$ and $e_i^1 = H^*[2i-1] \oplus H[2i]$, $1 \leq i \leq n$.
3. Find the n bits $a[i]$, $1 \leq i \leq n$ such that $e_1^{a[1]} \oplus e_2^{a[2]} \oplus \ldots \oplus e_n^{a[n]} = \bigoplus_{i=1}^{n} (e_i^0 \times (1-a[i]) \oplus e_i^1 \times a[i]) = C_{\mathrm{t}}$. This equation can be seen as a set of n binary equations treating each bit of C_{t} individually. That yields n linear equations in n unknowns, which can be solved using Gaussian elimination.

Complexity: It requires $n(2^{n/2+1})$ evaluations of the compression function to construct the CCS and around n^3 XORs to solve n equations using Gaussian elimination (memory requirements are negligible). In comparison, searching the CCS using the meet-in-the-middle attack requires $2^{n/2+1}$ XORs and about $2^{n/2+1}$ memory.

Remark 2. Similarly, XOR checksums can be defeated in F-Hash and 3CM. If an XOR checksum is computed using both message blocks and intermediate hash values, linear equations due to the XOR of the intermediate hash values and those of message blocks need to be solved.

5.2 Defeating additive checksums

Consider an additive checksum mod 2^d computed using messages for a **MD** hash function. It is possible to build a CCS as building it for XOR checksums, but both its construction and its use require some different techniques.

Building a CCS with Control of Message Blocks. When the collision finding algorithm is simply brute-force collision search, we can build a CCS for the complexity required to construct a 2^d-collision. Using the CCS to control the checksum then requires negligible work.

In this algorithm, we construct a 2^d-collision, in which each ordinary collision is two message blocks long. We choose the two-block messages in the collisions in such a way that the additive difference between the pair of two-block messages in each collision is a different power of two. The result is a CCS in which the first collision allows us to add 2^0 to the checksum, the next allows us to add 2^1, the next 2^2, and so on up to 2^{d-1}. This means that the checksum is entirely controlled[4]. In the following, arithmetic on message blocks is to be interpreted modulo 2^d.

Algorithm. Constructing the CCS.

```
1. Let $h = H[0]$.
2. For $i = 1$ to $d$:
   (a) Let $A, B$ be two random message blocks.
   (b) For $j = 1$ to $2^{n/2}$:
       i. $X[j] = A + j$
       ii. $X^*[j] = A + j + 2^{i-1}$
       iii. $Y[j] = f(f(h, X[j]), B - j)$
       iv. $Y^*[j] = f(f(h, X^*[j]), B - j)$
   (c) Search for a collision between the lists $Y$ and $Y^*$ (if the two lists are sorted during construction, then this search takes linear time). Let $u$ and $v$ be indices such that $Y[u] = Y^*[v]$.
   (d) Let $m[2i-1] = X[u]$ and $m[2i] = B - u$.
   (e) Let $m^*[2i-1] = X^*[v]$ and $m^*[2i] = B - v$.
   (f) Now $m^*[2i-1] + m^*[2i] - (m[2i-1] + m[2i]) = X^*[v] + B - v - (X[u] + B - u) = A + v + 2^{i-1} + B - v - (A + u + B - u) = 2^{i-1}$.
   (g) Let $h = Y[u]$.
```

Searching the CCS for the message $M[1] \| M[2] \| \cdots \| M[2d]$ that produces the right checksum is done as follows (assume the generic attack fixed the checksum C_{t}).

Algorithm. Searching the CCS.

```
1. Compute the checksum $C_0$ of the message blocks $m[i]$, $1 \leq i \leq 2d$.
2. Let $a = C_{\mathrm{t}} - C_0 \bmod 2^d$, and denote each individual bit of $a$ by $a[i]$, $1 \leq i \leq d$.
3. For $i = 1$ to $d$:
   (a) If $a[i] = 0$ let $M[2i-1] = m[2i-1]$ and $M[2i] = m[2i]$, otherwise let $M[2i-1] = m^*[2i-1]$ and $M[2i] = m^*[2i]$.
```

At the end of this process, M contains a sequence of $2d$ message blocks which, when put in the place of the CCS, will force the checksum to the desired value.

Complexity: Constructing the CCS requires about $d2^{n/2+2}$ work. Searching the CCS for the right message takes a negligible amount of time and memory. For the specific parameters of the GOST hash, the total work is about $256 \cdot 2^{130} = 2^{138}$. (The same CCS could be used for many different messages.) In comparison, searching the generically constructed CCS on GOST using meet-in-the-middle attack requires about 2^{129} addition operations.

6 Generic attacks on the checksum-based hash functions

In this section, we describe how to adapt generic attacks on the **MD** hash functions to the checksum-based hash function $\tilde{H} = H(m \| C(m))$. We illustrate every attack on $\tilde{\mathbf{H}}$ whose checksum function is invertible in time $T = 2^{160}$ and on MD2, GOST and 3C-SHA-256 whose checksum functions are invertible in time $T = 1$.

[4] A variant of this algorithm could be applied to many other checksums based on group operations.

6.1 Second preimage attack

Using the CCS, we can defeat the checksum in the hash construction $\tilde{H}$ to carry out the second preimage attack of [14]. The method is as follows: we first process the given target message with $\tilde{H}$ and store all the intermediate hash values and checksum values. To find the second preimage for $\tilde{H}$, we first construct an *expandable message* on the hash function H using a suitable method from [14], then construct the CCS from the end of the expandable message. We carry out the second preimage attack from the end of the CCS, and then we expand the expandable message to make up for all the blocks skipped by the attack. Finally, we search the CCS to find the message which results in the right checksum. Then we are able to produce a second preimage of the same length as the target message, and with the same checksum. In algorithmic form, the attack works as follows.

Algorithm. Assume $m = m[1] \| \cdots \| m[t]$ is the given target message of length $t \approx 2^k$ message blocks. Assume that C is invertible in time $T = 2^\tau$.

1. Hash m and store the intermediate hash values $H[i]$ and checksum values $C[i]$ for $1 \leq i \leq t$.
2. Starting from $H[0]$ as defined by the hash function $\tilde{H}$, build an expandable message of size from k up to 2^k blocks. Let the resulting intermediate hash value be H_{exp}. Let $m_{\text{exp}}(\ell)$ be the expandable message of length ℓ blocks.
3. Construct a CCS of size 2^d, starting from H_{exp}. Let the intermediate hash value at the end of the CCS be H_{CCS}.
4. Find a linking message block m_{link}, such that $f(H_{\text{CCS}}, m_{\text{link}})$ equals one of the values $H[i]$ of the target message, where $i > k + d$. Let this i be i'. Find the corresponding checksum $C[i']$.
5. Compute the intermediate checksum of $m_{\text{exp}}(\ell)$, where $\ell = i' - d - 1$. Let this checksum be C^*. Search the CCS for a message which, starting from C^*, has checksum $C[i']$. Let this message be m_{CCS}.
6. Finally, produce the second preimage as

$$m^* = m_{\text{exp}}(\ell) \| m_{\text{CCS}} \| m_{\text{link}} \| m[i'+1] \| \cdots \| m[t].$$

Complexity. It takes about $k2^{n/2}$ evaluations of the compression function f to construct the expandable message of size from k up to 2^k blocks using the generic expandable message algorithm [14]. If fixed points can be easily found for f, then the expandable message may be produced in time about $2^{n/2}$ [4, 14]. Constructing the CCS takes an expected time of $d2^{n/2}$. It takes 2^{n-k} evaluations of the compression function f to find the linking block. The time required to search the CCS is at most 2^d. The total complexity is then about $k2^{n/2} + d2^{n/2} + 2^{n-k} \approx 2^{n-k}$ evaluations of the compression function f and at most 2^d evaluations of the checksum function.

The expandable message and the CCS can be pre-computed and stored before the long target message is given. The expandable message of size at most 2^k blocks constructed using the generic method of [14] requires memory of size about 2^k. If the expandable message is constructed using fixed points [4,14] in the compression function then memory requirements are about $2^{n/2+1}$. Storing a CCS of size 2^d requires negligible memory. Memory required in order to search the CCS, assuming that the checksum function can be inverted in time 2^τ, is $2^{(d-\tau)/2+1}$ due to the meet-in-the-middle attack as described in Section 4.2 (negligible if the techniques of Section 5 can be used to search the CCS). Storing the long target message and the intermediate hash and chaining values requires about 2^k memory.

Example 1. $\tilde{\mathbf{H}}$ can hash messages of length up to about 2^{55} blocks. Hence, the complexity of the above attack given a target message of this length is about $2^{256-55} = 2^{201}$ compression function evaluations of SHA-256. Note that the 2^{160} evaluations of the SHA-1 checksum function required to defeat the 160-bit checksum of $\tilde{\mathbf{H}}$ are negligible in comparison to 2^{201} SHA-256 compression function evaluations. The attack requires storing about 2^{55} message blocks, intermediate hash values, and intermediate chaining values. This attack has the same complexity as the second preimage attack of [14] on SHA-256. Hence, using SHA-1 as a checksum function for SHA-256 provides no additional security than SHA-256 against the long message second preimage attack.

Example 2. The above attack is also applicable to MD2 [12] which uses a non-linear checksum function with a 128-bit state computed over message blocks. Unlike **MD** hash functions, MD2 does not use **MD**-strengthening, and it processes messages of any length. Given a target message of 2^{64} blocks, the above attack can be used to find a second preimage for MD2 in about 2^{71} evaluations of its compression function. Memory requirements of the attack are on the order of 2^{64}. This is an improvement over the previous best known second preimage attack on MD2 [15] based on the preimage attack with a complexity of $2^{97.6}$ compression function evaluations which requires about 2^{72} memory. In both attacks, the complexity due to evaluations of the checksum function is negligible compared to the total complexity.

Example 3. The 256-bit GOST hash function uses a mod 2^{256} additive checksum computed over message blocks and hashes messages up to 2^{256} bits. For a really long target message of 2^{128} 256-bit blocks, our second preimage attack on GOST requires 2^{138} evaluations of its compression function. Note that this complexity is due to the construction of the CCS. This is also equal to the complexity required to construct the CCS for GOST by controlling message blocks as demonstrated in Section 5.2. The attack requires about 2^{128} memory.

Similarly, the complexity to find a second preimage for 3C-SHA-256 for a given target message of 2^{55} blocks is 2^{201} evaluations of the SHA-256 compression function and about 2^{24} XOR operations when the CCS is constructed and searched using the algorithm in Section 5.1. Memory requirements of the attack are about 2^{55}.

6.2 Herding attack

We describe the herding attack on the checksum-based hash construction $\tilde{H}(m) = H(m \| C(m))$. The attack consists of two phases: a pre-computation phase and an on-line phase.

Algorithm.

Precomputation phase:

1. Choose arbitrarily 2^k intermediate hash values for $\tilde{H}$, and place them in the list L. Find 2^{k-1} pairwise collisions (under f) from these 2^k hash values.
2. Given the new 2^{k-1} intermediate hash values, find again 2^{k-2} pairwise collisions from these intermediate hash values.
3. Repeat this process until there is only a single hash value H_{dia} left. These collisions have the structure of a binary tree and resemble a diamond.
4. Choose some fixed checksum value C_{t} and compute the hash value $H_{\text{t}} = f(H_{\text{dia}}, C_{\text{t}})$. Publish H_{t}.

Online phase:

1. Form the message m_{pre} from the relevant information that recently became available.

2. Starting from $H[0]$ of $\tilde{H}$, compute the intermediate hash value H_{pre} of m_{pre}. Let C_{pre} be the corresponding checksum value.
3. Starting from H_{pre}, construct a CCS of size 2^d. Let the intermediate hash value at the end of the CCS be H_{CCS}.
4. Search for a message block m_{link} such that $f(H_{\text{CCS}}, m_{\text{link}})$ is equal to one of the hash values in the list L. Let this value be H_{link}.
5. There is always a path in the diamond structure which connects the value H_{link} to the target hash value H_{t}. Let this path be m_{dia}.
6. Search for a message m_{CCS} in the CCS which, when processed from C_{pre} using the checksum function C via m_{link} and m_{dia} equals the target checksum value C_{t}.
7. Now the message
$$m^* = m_{\text{pre}} \| m_{\text{CCS}} \| m_{\text{link}} \| m_{\text{dia}}$$
has the hash value H_{t}.

Note that in order to account for the **MD** strengthening, the length of the final message must be fixed already in the precomputation phase.

Complexity. The precomputation phase has complexity roughly $2^{n/2+k/2+2}$ evaluations of the compression function. The complexity of the online phase is the following. Constructing the CCS requires about $d2^{n/2}$ evaluations of f. Searching the CCS and through the linking block and diamond to find a message which equals the published hash value takes time at most $(k+1)2^{\tau} + 2^{(d+\tau)/2}$. Finding the linking message takes time about 2^{n-k}. With $k \approx n/3$, the two phases have similar complexity when the checksum is not taken into account. The checksum may add to this complexity if the checksum function is large or one-way. As an example, with $d \ll 2n/3$, the additional complexity is negligible, even if C is one-way. If C is efficiently invertible, then the additional complexity is negligible even with $d < 4n/3$. Memory requirements are about $2^{k+1} + 2^{(d-\tau)/2+1}$ (or 2^{k+1} if the techniques of Section 5 can be used to search the CCS).

Example 4. When our herding attack is applied to $\tilde{\mathbf{H}}$ with $k = n/3 \approx 85$, the complexity of the attack is about $2^{160} + 2^{160} + 85 \times 2^{160} \approx 2^{167}$ evaluations of the SHA-1 compression function and 2^{172} evaluations of SHA-256 compression function. Since the number of evaluations of the checksum function is much lower than the number of evaluations of the compression function, the complexity of the attack on $\tilde{\mathbf{H}}$ is roughly the same as the complexity of the herding attack on SHA-256 [13]. The attack requires about 2^{86} memory.

Example 5. The herding attack may also be applied to MD2. Since the MD2 checksum function is invertible in time $T = 1$, and $d = n$, the additional work due to the checksum is negligible. Hence, for a diamond width of $k = 42$, the herding attack on MD2 takes about 2^{87} evaluations of the compression function in both the precomputation and online phases. The attack requires around 2^{65} memory.

Example 6. Similarly, herding 3C-SHA-256 with $k = 84$ requires about $2^{136} + 2^{172} + 84 \times 2^{129} + 2^{171} \approx 2^{172}$ evaluations of the SHA-256 compression function and about 2^{24} XOR operations. About 2^{85} memory is required. Similarly, the herding attack on the GOST hash function with $k = 84$ requires about 2^{172} evaluations of its compression function and memory around 2^{85}.

Remark 3. Our second preimage and herding attacks can also be applied to hash functions that use checksums computed over intermediate hash values or both message blocks and intermediate hash values by building a CCS so that each individual collision is two blocks long. Thus, our techniques can be easily modified to attack non-linear checksum variants of the constructions 3C, F-Hash and Maelstrom-0 that use intermediate hash values to compute checksums.

6.3 Multicollisions in checksum-based hash functions

The CCS used in the second preimage and herding attacks can also be used to construct multicollisions on hash functions using checksums. In this section, we show some better algorithms to find multicollisions in these hash functions without the need for constructing and searching the CCS.

Application of Joux' collision attack on the cascaded construction. Since a checksum-based hash function may be viewed as a cascaded construction [26] followed by a "merging" of the two chains, Joux's collision attack [11] on the cascaded construction applies to all checksum-based hash functions. Such a collision can be used to construct multicollisions. A 2^k-collision attack on $\tilde{H}$ can be carried out as follows.
Algorithm. Let $C_{\text{iv}} = C[0]$.

1. For i from 1 to k do:
   (a) Starting from C_{iv}, find a $2^{n/2}$-collision on the checksum chain. Let the common checksum value be C^*.
   (b) Use brute force search to find a collision on the hash chain among the $2^{n/2}$ messages in the multicollision on the checksum chain.
   (c) Let $C_{\text{iv}} = C^*$.

Complexity. The complexity of the attack, when Joux's method to find multicollisions is used in Step 1a, is $k \times n/2 \times 2^{d/2} + k \times 2^{n/2}$. Memory requirements are around $2^{n/2}$.

Note that in the above 2^n-collision attack, we make no assumptions on the checksum function, nor on the compression function, except that we assume they take about the same time to evaluate. Hence, the roles of the two functions may be switched, so we may instead first find a 2^d-collision on the hash chain, and then brute force a collision on the checksum chain. The complexity is then $k \times d/2 \times 2^{n/2} + k \times 2^{d/2}$.

Example 7. The above attack on $\tilde{\mathbf{H}}$ has complexity about $k \times 2^{87} + k \times 2^{128} \approx k \times 2^{128}$ (and about 2^{128} memory). This is to be compared with the complexity of about $k \times 2^{128}$ for the attack on SHA-256.

Using an easily linkable checksum function. If it is easy to link the checksum function C in $\tilde{H}$ then multicollisions can be found for $\tilde{H}$ with complexity independent of d. Assume linking can be done in time $T = 2^\tau$. Then we can find a 2^k-collision as follows:
Algorithm. Let $C_{\text{iv}} = C[0]$.

1. For i from 1 to k do:
   (a) Choose $2^{n/2}$ arbitrary message blocks, and compute the intermediate checksum values of these blocks starting from C_{iv}.
   (b) Choose some arbitrary checksum value C^*, and find, for each of the checksum values computed in the previous step, the message block that produces C^*.

(c) We now have $2^{n/2}$ two-block messages. Find a collision among these in the hash chain.
(d) Let $C_{\text{iv}} = C^*$.

Complexity. Computing the checksum function $2^{n/2}$ times in the forward direction takes $2^{n/2}$ time. Linking the checksum function $2^{n/2}$ times takes time $T2^{n/2}$. The brute force collision attack on the hash chain takes time $2^{n/2+1}$, since we need to process two message blocks. The total complexity to find a collision is therefore about $(T+1)2^{n/2}$ evaluations of C and $2^{n/2+1}$ evaluations of the compression function f. We repeat this k times to obtain a 2^k-collision. The total complexity is $k(T+1)2^{n/2}$ evaluations of C and $k \times 2^{n/2+1}$ evaluations of f. Memory requirements are about $2^{n/2}$. The messages in the multicollision are of $2k$ blocks each.

Example 8. Since the checksum function of MD2 is linkable in time $T = 1$ and its checksum function can be evaluated in time equivalent to $2^{-5.7}$ compression function evaluations, the 2^k-collision attack on MD2 takes time about $k \times 2^{65} + k \times 2^{59.3} \approx k \times 2^{65}$ compression function evaluations. Memory requirements are approximately 2^{64}. The messages in the multicollision are each $2k$ blocks in length.

Remark 4. If the checksum is computed using intermediate hash values, then every 1-block collision on the hash chain would also result in a 1-block collision on the checksum chain. Hence a 2^k-collision can be constructed on such a checksum-based hash function in $k \times 2^{n/2}$ compression function evaluations.

7 Application of our attacks to the cascaded hash functions

Since the attacks described in the previous section span both the hash chain and the checksum chain, they can also be applied to some cascaded constructions [26]. Hence our attacks on the construction $\tilde{H}$ are also applicable to the cascaded constructions of form $H(m)\|G(m)$ and $H(m)\|G(m\|H(m))$ where H and G are two different iterated hash functions or two different variants of the same hash function. In [11], Joux showed that with respect to collision and (second) preimage attacks, a cascade of two hash functions is no more secure than the stronger hash function in the cascade. Below, we provide complexities of our long message second preimage attack on some cascaded hash functions for a given target message of 2^{55} blocks and compare them with the work of [11].

While the second preimage attack of [11] on $\mathsf{SHA256}(m)\|\mathsf{SHA1}(m)$ requires about 2^{256} evaluations of SHA-256 and 2^{160} evaluations of SHA-1, our attack requires about 2^{201} evaluations of SHA-256 and 2^{160} evaluations of SHA-1, effectively the same as the complexity of the long message second preimage attack of [14] on SHA-256 itself. In some cases, however, the weakest hash function affects the complexity. For example, while the second preimage attack of Joux [11] on $\mathsf{SHA1}(m)\|\mathsf{MD5}(m)$ requires 2^{160} evaluations of SHA-1 and 2^{128} evaluations of MD5, our attack needs 2^{105} evaluations of SHA-1 and 2^{128} evaluations of MD5. If H and G are of equal strengths, then our second preimage attack is slightly better than the attack of [11]. See Table 4 for some illustrations. It is important to note that the complexities of our second preimage attacks on the cascaded hashes correspond to those applied to impractically long messages of 2^{55} 512-bit blocks unlike those of [11]. Also, memory requirements are around $2^{n/2}$, where n is the output size of the largest hash function in the cascade.

Similarly, it is straight-forward to apply our multicollision and herding attacks to the cascaded hash functions. Our multicollision attacks on the cascaded hash functions complement the collision attacks of [11] and [5] on these hash functions. The herding attack of Dunkelman and Preneel [5]

Table 4. Comparison of complexities to carry out the second preimage attack with those of Joux [11].

Hash Construction	Joux's work	Our work	Ideal case
$\mathsf{SHA256}(m)\|\|\mathsf{SHA1}(m\|\|\mathsf{SHA256}(m))$	2^{256}	2^{201}	2^{416}
$\mathsf{SHA256}(m)\|\|\mathsf{SHA1}(m)$	2^{256}	2^{201}	2^{416}
$\mathsf{SHA1}(m)\|\|\mathsf{MD5}(m)$	2^{160}	2^{128}	2^{288}
$\mathsf{SHA1}(m)\|\|\mathsf{RIPEMD160}(m)$	2^{161}	2^{160}	2^{320}

on the cascaded hash functions is more efficient than ours in some cases; especially when the two hash functions in the cascade are of the same size. For example, to herd the cascaded hash function $\mathsf{SHA1}(m)||\mathsf{RIPEMD160}(m)$, the on-line phase of the attack of [5] for $k = 54$ requires 2^{106} evaluations of SHA-1 and about 2^{113} evaluations of RIPEMD-160. Our attack requires 2^{106} evaluations of SHA-1 and about 2^{160} evaluations of RIPEMD-160.

8 On carrying out generic attacks using cryptanalytic collision attacks

We note that it is difficult to construct the CCS using cryptanalytic collision finding algorithms such as the ones built on MD5 and SHA-1 [33, 34] in order to defeat even linear checksums to carry out generic attacks. For example, consider two 2-block colliding messages of the form $(m[2i-1], m[2i])$,$(m^*[2i-1], m^*[2i])$ for $i = 1, \ldots, t$ on the underlying **MD** of 3C based on the near collisions due to the first blocks in each pair of the messages. Usually, the XOR differences of the nearly collided intermediate hash values are either fixed or very tightly constrained as in the collision attacks on MD5 and SHA-1 [33,34]. 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 CCS using colliding blocks of the form $(m[2i-1], m[2i])$,$(m^*[2i-1], m[2i])$. It is not possible to control the XOR checksum due to 2-block collisions of the form $(m[2i-1], m[2i])$, $(m[2i-1], m^*[2i])$ [32] as this form produces a zero XOR difference in the checksum after every 2-block collision. Similarly, cryptanalytic collision attacks on the compression functions do not help in carrying out generic attacks on the hashes that use non-linear checksums.

Though we cannot perform generic attacks on the hash functions that use linear checksums using structured collisions, we can find multi-block collisions by concatenating two structured collisions. Consider a one-block collision finding algorithm for the GOST hash function H. A call to this collision finder results in a pair of b-bit message blocks $(m[1], m^*[1])$ such that $m[1] \equiv m^*[1] + \Delta \bmod 2^b$ and $f(H[0], m[1]) = f(H[0], m^*[1]) = H[1]$. Now call the collision finding algorithm with $H[1]$ as the starting state, which results in a pair of blocks $(m[2], m^*[2])$ such that $m^*[2] \equiv m[2] + \Delta \bmod 2^b$ and $f(H[1], m[2]) = f(H[1], m^*[2]) = H[2]$. That is, $H(H[0], m[1]||m[2]) = H(H[0], m^*[1]||m^*[2])$. Consider $m[1] + m[2] \bmod 2^b = \Delta + m^*[1] + m^*[2] - \Delta \bmod 2^b = m^*[1] + m^*[2] \bmod 2^b$. This is a collision in the additive checksum chain.

9 Comparison of our technique with that of Mironov and Narayanan

Mironov and Narayanan (personal communication at Crypto'06, August, 2006) have found an alternative technique to defeat XOR checksums computed using message blocks. We call this design GOST-x. While our approach from Section 5 to defeat the XOR checksum in GOST-x requires finding a 2^b-collision using b random 1-block messages $(m[i], m^*[i])$ 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 systems 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 messages would result in a zero checksum at the end of the 2^b-collision 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 CCS which matches some intermediate hash value 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 XOR checksum. These messages would result in a zero checksum at every level 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 Mironov and Narayanan is applied to defeat checksums in 3C, 3CM and F-Hash, the 2^n 2-block collision finding algorithm used to construct the CCS must output the same pair of message blocks on either side of the collision whenever it is called. This constraint is not there in our technique, and the approach of Mironov and Narayanan is not quite as powerful. However, it could be quite capable of defeating 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 approach to defeat XOR checksums does not work.

10 Conclusion and open problems

Simple and fast checksums have been considered for a long time as possible methods of complicating many attacks on hash functions.

The attacks of this paper show that many such checksum-based hash functions are vulnerable to generic attacks on the iterated hash functions that do not depend on the intrinsic properties of the components. It was shown that such attacks are possible even in the cases where the checksum function is assumed to be ideal in the sense that inverting it can only be done in a brute-force manner. Also, even if the checksum function is as strong as the compression function, multicollision attacks are still applicable. Previous results [5] have shown that also herding attacks can be applied. These combined results show that the checksum function in a hash function must be very strong if it is to provide any additional security, and hence it is also likely to result in a severe reduction in efficiency. Thus, checksum functions do not seem to provide the best protection for hash functions against generic attacks. Better alternatives seem to be to continuously mix the outputs of two different functions, such as the double pipe scheme of Lucks [18].

It is also fair to note that a checksum may still be a good way of protecting against shortcut attacks. For instance, as far as we know, there is still no known shortcut collision attack on the MD2 hash function, but there are very efficient collision attacks on its compression function.

Our study on the security of checksum based hash functions leaves a number of questions open. Among these, the most interesting is, whether there are faster methods of circumventing checksums than the use of a checksum control sequence. Another question is on constructing efficient checksum control sequences to defeat simple non-linear checksums such as the ones in MD2 similar to the ones constructed to defeat linear checksums. It is also an interesting research problem to improve the 1024-block collision attack on the GOST hash function [19] to a 2-block collision attack as described in this paper using the collision attack on the compression function [19]. Another possible research area is to construct alternative simple and efficient extensions to the **MD** construction that provide protection against all generic attacks.

References

1. M. Bellare and D. Micciancio. A new paradigm for collision-free hashing: Incrementality at reduced cost. In W. Fumy, editor, *EUROCRYPT*, volume 1233 of *LNCS*, pages 163–192, 1997.
2. D. Coppersmith. Two broken hash functions. IBM Research Report RC 18397, IBM T. J. Watson Research Center, Yorktown Heights, N.Y., 10598, USA, October 1992.
3. I. Damgård. A Design Principle for Hash Functions. In G. Brassard, editor, *Advances in Cryptology – CRYPTO '89, Proceedings*, volume 435 of *Lecture Notes in Computer Science*, pages 416–427. Springer, 1990.
4. R. D. Dean. *Formal Aspects of Mobile Code Security.* PhD thesis, Princeton University, January 1999.
5. O. Dunkelman and B. Preneel. Generalizing the Herding Attack to Concatenated Hashing Schemes. Presented at ECRYPT hash function workshop, May 24–25, 2007, Barcelona, Spain. Available: `http://events.iaik.tugraz.at/HashWorkshop07/program.html` (2008/1/28).
6. D. G. Filho, P. Barreto, and V. Rijmen. The MAELSTROM-0 Hash Function. Published at 6th Brazilian Symposium on Information and Computer System Security, 2006.
7. P. Gauravaram and J. Kelsey. Linear-XOR and additive checksums don't protect damgård-merkle hashes from generic attacks. In T. Malkin, editor, *Topics in Cryptology - CT-RSA 2008*, volume 4964 of *Lecture Notes in Computer Science*, pages 36–51. Springer, 2008.
8. P. Gauravaram, W. Millan, E. Dawson, and K. Viswanathan. Constructing Secure Hash Functions by Enhancing Merkle-Damgård Construction. In L. M. Batten and R. Safavi-Naini, editors, *Australasian Conference on Information Security and Privacy 2006, Proceedings*, volume 4058 of *Lecture Notes in Computer Science*, pages 407–420. Springer, 2006. The full version of this paper is available at `http://www.isi.qut.edu.au/research/publications/technical/qut-isi-tr-2006-013.pdf`. (Accessed on 2008/9/15).
9. D. L. Gazzoni Filho, P. S. L. M. Barreto, and V. Rijmen. The Maelstrom-0 Hash Function. Published at 6th Brazilian Symposium on Information and Computer System Security, August 28–September 1, 2006, Santos, Brazil.
10. J. Hoch and A. Shamir. Breaking the ICE: Finding Multicollisions in Iterated Concatenated and Expanded (ICE) Hash Functions. Proceedings of 13th Annual Fast Software Encryption (FSE) International Conference, 2006.
11. A. Joux. Multicollisions in Iterated Hash Functions. Application to Cascaded Constructions. In M. K. Franklin, editor, *Advances in Cryptology – CRYPTO 2004, Proceedings*, volume 3152 of *Lecture Notes in Computer Science*, pages 306–316. Springer, 2004.
12. B. S. Kaliski Jr. The MD2 Message-Digest Algorithm, April 1992. RFC 1319. Available: `http://www.ietf.org/rfc/rfc1319.txt` (2008/1/28).
13. J. Kelsey and T. Kohno. Herding Hash Functions and the Nostradamus Attack. In S. Vaudenay, editor, *Advances in Cryptology – EUROCRYPT 2006, Proceedings*, volume 4004 of *Lecture Notes in Computer Science*, pages 183–200. Springer, 2006.
14. J. Kelsey and B. Schneier. Second Preimages on n-Bit Hash Functions for Much Less than 2^n Work. In R. Cramer, editor, *Advances in Cryptology – EUROCRYPT 2005, Proceedings*, volume 3494 of *Lecture Notes in Computer Science*, pages 474–490. Springer, 2005.
15. L. R. Knudsen and J. E. Mathiassen. Preimage and Collision Attacks on MD2. In H. Gilbert and H. Handschuh, editors, *Fast Software Encryption 2005, Proceedings*, volume 3557 of *Lecture Notes in Computer Science*, pages 255–267. Springer, 2005.
16. X. Lai and J. L. Massey. Hash Functions Based on Block Ciphers. In R. A. Rueppel, editor, *Advances in Cryptology – EUROCRYPT '92, Proceedings*, volume 658 of *Lecture Notes in Computer Science*, pages 55–70. Springer, 1993.
17. D. Lei. F-HASH: Securing Hash Functions Using Feistel Chaining. Cryptology ePrint Archive, Report 2005/430, 2005. Available: `http://eprint.iacr.org/2005/430.pdf` (2006/11/11).
18. S. Lucks. A Failure-Friendly Design Principle for Hash Functions. In B. K. Roy, editor, *Advances in Cryptology – ASIACRYPT 2005, Proceedings*, volume 3788 of *Lecture Notes in Computer Science*, pages 474–494. Springer, 2005.

19. F. Mendel, N. Pramstaller, C. Rechberger, M. Kontak, and J. Szmidt. Cryptanalysis of the GOST hash function. In D. Wagner, editor, *Advances in Cryptology - CRYPTO*, volume 5157 of *Lecture Notes in Computer Science*, pages 162–178. Springer, 2008.
20. R. C. Merkle. One Way Hash Functions and DES. In G. Brassard, editor, *Advances in Cryptology – CRYPTO '89, Proceedings*, volume 435 of *Lecture Notes in Computer Science*, pages 428–446. Springer, 1990.
21. F. Muller. The MD2 Hash Function Is Not One-Way. In P. J. Lee, editor, *Advances in Cryptology – ASIACRYPT 2004, Proceedings*, volume 3329 of *Lecture Notes in Computer Science*, pages 214–229. Springer, 2004.
22. M. Nandi and D. Stinson. Multicollision attacks on some generalized sequential hash functions. *IEEE Transactions on Information Theory*, 53(2):759–767, 2007.
23. National Institute for Standards and Technology. *Federal Information Processing Standard (FIPS PUB 180-3) Secure Hash Standard.* National Institute for Standards and Technology, 2007. Available at `http://csrc.nist.gov/publications/drafts/fips_180-3/draft_fips-180-3_June-08-2007.pdf` (Accessed on 7/22/2008).
24. National Institute of Standards and Technology. FIPS PUB 180-1, Secure Hash Standard, 17 April 1995.
25. National Institute of Standards and Technology. FIPS PUB 180-2, Secure Hash Standard, 1 August 2002.
26. B. Preneel. *Analysis and Design of Cryptographic Hash Functions.* PhD thesis, Katholieke Universiteit Leuven, February 1993.
27. J.-J. Quisquater and M. Girault. 2n-Bit Hash-Functions Using n-Bit Symmetric Block Cipher Algorithms. In J.-J. Quisquater and J. Vandewalle, editors, *Advances in Cryptology – EUROCRYPT '89, Proceedings*, volume 434 of *Lecture Notes in Computer Science*, pages 102–109. Springer, 1990.
28. R. L. Rivest. The MD5 Message-Digest Algorithm, April 1992. RFC 1321. Available: `http://www.ietf.org/rfc/rfc1321.txt` (2008/1/28).
29. N. Rogier and P. Chauvaud. MD2 Is not Secure without the Checksum Byte. *Designs, Codes and Cryptography*, 12(3):245–251, November 1997.
30. Rostekhregulirovaniye (Russia's Federal Agency for Technical Regulation and Metrology). GOST R 34.11-94: Information technology – Cryptographic data security – Hashing function, 1994.
31. D. Wagner. A Generalized Birthday Problem. In M. Yung, editor, *CRYPTO 2002*, volume 2442 of *LNCS*, pages 288–303. Springer, 2002.
32. X. Wang, Y. L. Yin, and H. Yu. Efficient collision search attacks on SHA-0. In V. Shoup, editor, *CRYPTO*, volume 3621 of *LNCS*, pages 1–16. Springer, 2005, 14–18 Aug. 2005.
33. X. Wang, Y. L. Yin, and H. Yu. Finding collisions in the full SHA-1. In V. Shoup, editor, *CRYPTO*, volume 3621 of *LNCS*, pages 17–36. Springer, 2005, 14–18 Aug. 2005.
34. X. Wang and H. Yu. How to Break MD5 and Other Hash Functions. In R. Cramer, editor, *EUROCRYPT*, volume 3494 of *LNCS*, pages 19–35. Springer, 2005.