

On hash functions using checksums

Praveen Gauravaram · John Kelsey ·
Lars R. Knudsen · Søren S. Thomsen

Published online: 31 December 2009
© Springer-Verlag 2009

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. Our herding and multicollision attacks on the hash functions based on generic checksum functions (e.g., one-way) are a special case of the attacks on the cascaded iterated hash functions previously analysed by Dunkelman and Preneel and are not better than their attacks. On hash functions with easily invertible checksums, our multicollision and herding attacks (if the hash value is short as

in MD2) are more efficient than those of Dunkelman and Preneel.

Keywords Iterated hash functions · Checksums · Generic attacks

1 Introduction

1.1 Background

Cryptographic hash functions are important tools of cryptography used in many applications for secure and efficient information processing. 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 [3,21] (**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 input length collision-resistant compression function, a variable input length 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 Dean [4] and Kelsey and Schneier [14], and it also has other undesired weaknesses in the form of multicollisions [11] and herding attacks [13].

P. Gauravaram (✉) · L. R. Knudsen · S. S. Thomsen
Department of Mathematics, Technical University of Denmark
(DTU), Matematiktorvet, Building S303,
2800 Kgs.Lyngby, Denmark
e-mail: p.gauravaram@mat.dtu.dk

L. R. Knudsen
e-mail: lars.r.knudsen@mat.dtu.dk

S. S. Thomsen
e-mail: crypto@znoren.dk

J. Kelsey
Computer Security Division, National Institute of Standards
and Technology, 100 Bureau Drive, Stop 1070,
Gaithersburg, MD 20899-1070, USA
e-mail: john.kelsey@nist.gov

1.2 Our work

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. 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 significantly degrading its efficiency. Hence, it is desirable that a checksum function is at least as fast as the compression function itself. Applications may even require checksum to be much faster than the 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 [28], which uses an additive checksum, and the MD2 hash function of Rivest [29], 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 inaccurate, 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 n and d 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 [31], a simple non-linear function as in MD2 [12] or some complex one-way function such as the SHA-1 [25, 26] compression function. An example construction using SHA-1 as the checksum function for the SHA-256 hash function [25, 26] is defined by $\text{SHA256}(m \| \text{SHA1}(m))$, which we shall return to later. Similarly, instantiation of the 3C construction using SHA-256 compression function is called 3C-SHA-256.

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 the hash functions in the cascade are combined and processed in the end. That is, by using the compression function of one of the hash functions in the cascaded hash function as a checksum function, a cascaded construction can be turned into the checksum-based hash \tilde{H} . Hence, our attacks on the construction \tilde{H} also apply to the cascade of two independent hash functions H and G defined by $H(m) \| G(m)$ [27]. Similarly, our attacks easily extend to complex looking cascaded structures of form $H(m) \| G(m \| H(m))$ [11, Sect. 4.3].

1.3 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 [28] which processes two supplementary checksum blocks each computed independently using XOR and modular addition of message blocks. Joux [11] provided a technique to find 2^k -collisions for a **MD** hash function for only about k times as much work as is required for a collision, and uses this technique to attack cascade hashes. Gauravaram and Kelsey [7] analysed the security of the variants of **MD** hash functions that compute linear-XOR or additive checksums over message blocks, intermediate hash values, or both, and process these checksums in computing the final hash value. Dunkelman and Preneel [5] analysed the security of general iterated hash functions called “concatenated schemes” (also called cascaded constructions). Nandi and Stinson [23] 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 [23] 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, 30] and (second) preimage attacks on the full hash function [15, 16, 22] by exploiting weaknesses in its compression function. Mendel, Pramstaller and Rechberger [20] 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.

Table 1 Attacks on hash functions using checksums

Attack on the 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

Table 2 Second preimage attacks on specific hash functions

Hash function	Our complexity	Previous best known complexity	Ideal complexity
MD2	2^{71}	2^{73} [16]	2^{128}
GOST	2^{137}	2^{192} [20]	2^{256}
SHA256($m \parallel \text{SHA1}(m)$)	2^{202}	2^{256} [8]	2^{256}
3C-SHA-256	2^{202}	2^{256} [8]	2^{256}
MAELSTROM-0	2^{202}	2^{256} [9]	2^{256}

1.4 Comparison with prior work

The attack techniques presented in this paper when combined with the long-message second preimage attack of [14] improve the second preimage attack of Joux [11] on the cascaded 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. Our second preimage attacks on the GOST and MD2 hash functions improve significantly over the previous attacks of [20] and [15, 16, 22], respectively.

Gauravaram and Kelsey [7] have shown that the variants of **MD** hash functions that compute linear-XOR or additive checksums gain almost no security against the long-message second preimage attacks of [4, 14] and the herding attack of [13]. We analyse more general cases of the checksum function (e.g., non-linear and one-way functions) applied to the **MD** hash functions by extending the methods of [7]. Dunkelman and Preneel [5] analysed the security of more general iterated hash functions called “concatenated schemes” (also called cascaded constructions). Dunkelman and Preneel have proposed herding and multicollision attacks on this class of hash functions and left open the security of these hash functions against the second preimage attacks. The checksum-based hash functions analysed in our paper are the special cases of the concatenated iterated hash functions studied in [5]. Our second preimage attack on the cascaded hash functions solves the open question raised by Dunkelman and Preneel [5]. The herding and multicollision attacks of [5] also apply to the hash functions based on checksums. The complexity of our herding attack on the cascaded and checksum-based hash functions is not faster than the attack of [5] (herding attack on MD2 is an exception). In addition, our multicollision attack on the hash functions based on generic checksum function (e.g., one-way checksum function) has similar complexity to the attack of [5]. On hash functions with easily invertible checksums, our multicollision attack

is more efficient than that of [5]. A clear and explicit comparison of our results with those of Joux [11] for the second preimage attacks and with those of Dunkelman and Preneel [5] for the herding and multicollision attacks is provided later in the paper.

We summarise the state of the art of hash functions using checksums in Table 1 by combining our results with those of Dunkelman and Preneel [5]. Table 1 shows the sizes of the checksum function compared with the hash value sizes for which there are attacks better than brute force depending on whether checksum function is easily invertible or one-way.

1.5 Implications of our work

Our results have implications to the security of the 128-bit MD2 hash function [12] designed by Rivest, and the Russian standard GOST hash function [31]. 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^{73} based on a recent preimage attack [16]. 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 with the previous short-cut attack of 2^{192} [20]. The long-message second preimage 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 the 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 analytical results of the herding attack on these hash functions. Our second preimage and herding attacks require abundant memory as shown later in Sect. 6 of the paper.

Table 3 Herding attacks on specific hash functions

Hash function	Our complexity	Application of the attack of [5]	Ideal complexity
MD2	2^{87}	2^{92}	2^{128}
GOST	2^{172}	2^{172}	2^{256}
SHA256($m \parallel \text{SHA1}(m)$)	2^{172}	2^{172}	2^{256}
3C-SHA-256	2^{172}	2^{172}	2^{256}
MAELSTROM-0	2^{172}	2^{172}	2^{256}

1.6 Guide to the paper

In Sect. 2, we give an overview of cryptographic hash functions. In Sect. 3, we discuss hash functions using checksums. In Sects. 4 and 5, we introduce cryptanalytic tools that we later use in Sect. 6 to perform generic attacks on the checksum-based hash functions. In Sect. 6.4, we apply our attacks to the cascaded hash functions. In Sect. 7, we compare our results with those of Dunkelman and Preneel [5]. In Sect. 8, we discuss the application of cryptanalytic collision attacks on hash functions to carry out generic attacks on the checksum-based hash functions. In Sect. 9, we conclude the paper with some open questions.

2 Cryptographic hash functions

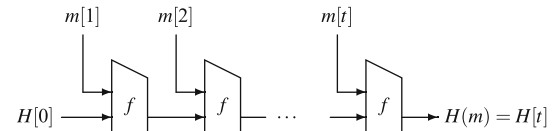
Cryptographic hash functions process an arbitrary length message into a fixed length hash value. Let $H : \{0, 1\}^* \rightarrow \{0, 1\}^n$ be an n -bit hash function. Informally, three well-known attacks on H are:

- **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.

2.1 Merkle-Damgård hash functions

Most iterated hash functions used in practice such as MD5 [29], SHA-1 [24] and SHA-256 [25, 26] follow the MD hash function framework. An n -bit MD hash function H , as illustrated in Fig. 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.

**Fig. 1** The MD construction

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** [17]. 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 $f : \{0, 1\}^n \times \{0, 1\}^b \rightarrow \{0, 1\}^n$. 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. For an ideal n -bit hash function H , 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 $\{m[1], m[2], \dots, m[2^k]\}$ such that $m[i] \neq m[j]$ whenever $i \neq j$, and $H(m[1]) = \dots = H(m[2^k])$. 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 $H(m) = H_t$. 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 .

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 $f : \{0, 1\}^n \times \{0, 1\}^b \rightarrow \{0, 1\}^n$ is iterated, and a *checksum chain* where a checksum function $C : \{0, 1\}^d \times \{0, 1\}^b \rightarrow \{0, 1\}^d$ is iterated, as illustrated in Fig. 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 generality 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.

3.1.1 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 Fig. 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 $\text{pad}(C[t])$ using the last compression function.

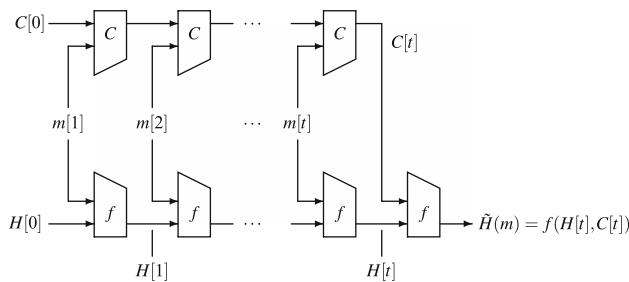


Fig. 2 The MD construction with checksum

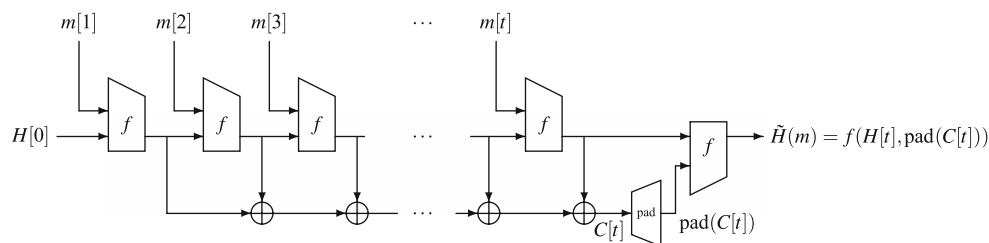
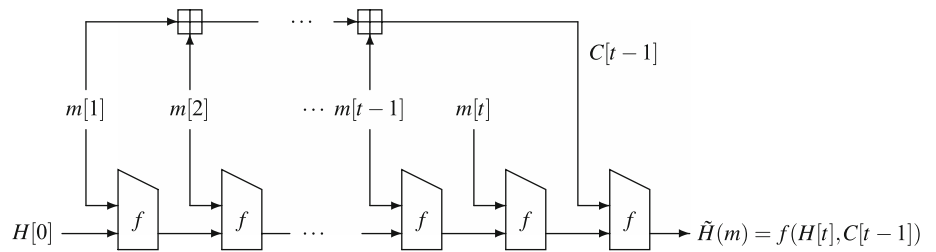


Fig. 3 The 3C-hash function

Fig. 4 The GOST 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 [18], 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.

3.1.2 GOST hash function

GOST is a 256-bit hash function specified in the Russian standard GOST R 34.11 [31]. 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 $m[1], \dots, m[t-1]$. 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 Fig. 4. At any iteration i , the intermediate hash value in the hash chain and checksum chains is $H[i] = f(H[i-1], m[i])$, where $1 \leq i \leq t$ and $m[1] + m[2] + \dots + m[i] \bmod 2^b$, respectively, where $1 \leq i \leq t-1$. The hash value of m is $H[v] = f(H[t], C[t-1])$, where $C[t-1] = m[1] + m[2] + \dots + m[t-1] \bmod 2^b$.

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{H} is defined as

$$\tilde{H}(m) = \text{SHA256}(m \parallel \text{SHA1}(m)).$$

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 shall not make use of any short-cut attacks on these hash functions in this paper. 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 [25, 26], which is 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} 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 definition later in the paper:

Definition 2 *Inverting C* is the process of computing $C[i-1]$ given $C[i]$ and $m[i]$ in time T . *Linking C* is 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 $T = 1$ [22]. 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 *checksum control sequence* (CCS) to extend a generic attack¹ on an iterated hash function H to the checksum-based hash construction $\tilde{H} = H(m \parallel C(m))$. 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

¹ Recall that these generic attacks were described in 2.1.

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 a specific CCS to defeat 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.²
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 multiblock messages (e.g., two-block messages) on its hash chain has to be produced to construct the CCS.

² 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.

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 .

4.2.1 Exploiting the structure of messages

If C is not easily invertible, e.g., SHA-1 in \tilde{H} , we can exploit the 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].

4.2.2 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

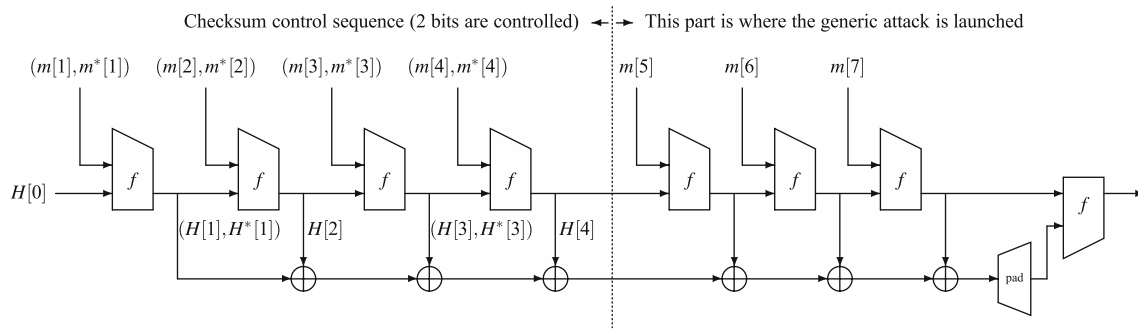


Fig. 5 Using the CCS to control 2 bits of the checksum (3C)

checksums can be controlled more efficiently using specific algorithms than the generic algorithms described in Sect. 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], [32]. This is shown in Fig. 5 for $k = 2$ where we compute the CCS by finding a 2^2 collision using random two-block messages. Then, we have a choice to choose either $H[1] \oplus H[2]$ or $H^*[1] \oplus H[2]$ from the first two-block collision and either $H[3] \oplus H[4]$ or $H^*[3] \oplus H[4]$ from the second two-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_1 be the desired value of the checksum.

Algorithm.

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 $e_1^{a[1]} \oplus e_2^{a[2]} \oplus \dots \oplus e_n^{a[n]} = \bigoplus_{i=1}^n (e_i^0 \times (1 - a[i]) \oplus e_i^1 \times a[i]) = C_1$. This equation can be seen as a set of n binary equations treating each bit of C_1 individually. That yields n linear

equations in n unknowns, which can be solved using Gaussian elimination.³

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.

5.2.1 Building a CCS with free choice 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

³ With exactly n equations in n unknowns, there will often be no solution. However, increasing the number of choices within the CCS by a small amount will make it extremely likely that a solution can be found.

next 2^2 , and so on up to 2^{d-1} . This means that the checksum is entirely controlled⁴. 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 $M[1] \| M[2] \| \dots \| M[2d]$ 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 $a = C_t - C_0 \bmod 2^d$, 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 into the appropriate place in the message, 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.

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

6 Generic attacks on 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 $\tilde{H} = H(m \| C(m))$. For each of these attacks, we illustrate the attack on \tilde{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 $T = 1$.

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

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 capable of taking on any size between k and 2^k blocks, inclusive. 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

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

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 precomputed 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 Sect. 4.2 (negligible if the techniques of Sect. 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{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{H} are negligible in comparison with 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 compared to 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} . In comparison, a recently published cryptanalytic second preimage attack [16] has time and memory complexity around 2^{73} . 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 Sect. 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 Sect. 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 precomputation phase and an online 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 $H_t = f(H_{\text{dia}}, C_t)$. 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 take 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 Sect. 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.

6.3.1 Application of Joux' collision attack on the cascaded construction

Since a checksum-based hash function may be viewed as a cascaded construction [27] 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 $\tilde{\mathbf{H}}$ 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

may be compared with the complexity of about $k \times 2^{128}$ for the attack on SHA-256.

6.3.2 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 $C_{iv} = C[0]$.

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 $C_{iv} = C^*$.

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 2^k -collision are each $2k$ blocks in length. Similarly, 2^k collisions can be constructed on the GOST hash function in time about $k \times 2^{129}$ compression function evaluations and $k \times 2^{129}$ addition operations. The attack requires a memory of approximately 2^{128} .

Remark 4 If the checksum is computed using intermediate hash values as in 3C-SHA-256, 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.

Remark 5 Mironov and Narayanan (personal communication at CRYPTO'06, August 2006) observed that on the MD hash functions that use XOR checksums using message blocks, a 2^k -collision attack can be constructed by repeating the same message block twice in every individual collision. This trick can be seen as a special case of our multicollision attack on the MD hashes that use easily linkable checksum function. We note that their trick does not work on the iterated hash functions that use simple non-linear checksums as in MD2.

6.4 Application of our attacks to cascaded hash functions

Since the second preimage, herding and multicollision attacks described in the previous sections span both the hash chain and the checksum chain, they can also be applied to some cascaded constructions [27]. 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.

7 Comparison of our results with those of Joux [11] and Dunkelman and Preneel [5]

7.1 Comparison with the results of Joux [11]

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. In the following, 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 $\text{SHA256}(m) \| \text{SHA1}(m)$ 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 $\text{SHA1}(m) \| \text{MD5}(m)$ 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.

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
SHA256(m) SHA1(m SHA256(m))	2^{256}	2^{201}	2^{416}
SHA256(m) SHA1(m)	2^{256}	2^{201}	2^{416}
SHA1(m) MD5(m)	2^{160}	2^{128}	2^{288}
SHA1(m) RIPEMD160(m)	2^{161}	2^{160}	2^{320}

7.2 Comparison with the results of Dunkelman and Preneel [5]

Dunkelman and Preneel [5] analysed more general iterated hash functions called “concatenated” hashing schemes in [5] against herding and multicollision attacks. In this class of hash functions, the compression function can be expressed as two or “more” data paths wherein the first path is not affected by the other paths, the second path may be affected by the first path but not by the other remaining paths and so on. The checksum-based iterated hash functions analysed in our paper are special cases of the concatenated (multipath) iterated hash functions studied in [5]. Hence, the herding and multicollision attack of [5] also applies to the checksum-based hash functions.

7.2.1 Herding attack

Our herding attack on the cascaded or checksum-based hash functions is not faster than that of Dunkelman and Preneel [5] whose complexity in the pre-processing phase is $2^{n/2+k/2} + (d/2 - k/2)2^{d/2+k/2}$ and in the online phase is $2^{n-k} + (d - k)2^{d-k}$. When their attack is applied to $\tilde{\mathbf{H}}$ with $k = n/3 \approx 85$, the total complexity is 2^{172} . The complexity of our attack is $2^{167} + 2^{172}$. The reason for similar complexities is that the attacks are dominated by the complexity to find the message to link to the precomputed diamond structure. This comparison holds even when the checksum function is easily invertible or linkable (i.e., $T = 1$).

In some cases, the herding attack of Dunkelman and Preneel [5] is more efficient than ours; in particular when the two hash functions in the cascade have the same size. For example, to herd the cascaded hash function SHA1(m)||RIPEMD160(m) where $n = d = 160$ bits and for $k = 54$, the online phase of the attack of [5] requires approximately 2^{113} . Our attack requires more than 2^{160} . The reason is that the attack of [5] is dominated by the precomputed stage, whereas ours is dominated by the online stage which has the complexity $d \times 2^{n/2} + (k + 1)2^d + 2^d + 2^{n-k}$.

The exception to the above comparison is the application of the herding attack on the checksum-based hash functions with short hash values and whose checksum is easily invertible (i.e., $T = 1$) such as the MD2 hash function. As noted

before, on MD2 with $k = 42$, the complexity of our attack is about 2^{87} . The complexity of the attack of [5] is about 2^{92} .

7.2.2 Multicollision

The complexity to find 2^k -collisions in hash functions based on general checksum functions (e.g., one-way checksums) using the result of Dunkelman and Preneel [5] combined with Joux' attack [11] is approximately: $k \times d/2 \times 2^{n/2+2^{d/2}}$. Our attack complexity is $k \times (n/2 \times 2^{d/2} + 2^{n/2})$. If $d = n$, then the attacks will have the same complexity. Therefore, for the general checksum functions with large d (close to n), the complexity of our multicollision attack is approximately the same as that of the attack using the method of [5]. For hash functions based on an easily linkable checksum function, the complexity of our attack approximates to $k \times 2^{n/2}$ and hence is more efficient than that of [5]. For example, the complexity of our attack to find 2^k -collisions on MD2 is $k \times 2^{65}$, whereas the attack of [5] requires $k \times 2^{70}$.

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 [34,35] in order to defeat even linear checksums to carry out generic attacks. For example, consider 2 two-block colliding messages of the form $(m[2i - 1], m[2i]), (m^*[2i - 1], m^*[2i])$ for $i = 1, \dots, t$ 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 [34,35]. 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 often not possible to control the XOR checksum due to two-block collisions of the form $(m[2i - 1], m[2i]), (m[2i - 1], m^*[2i])$ [33] as this form produces a zero XOR difference in the checksum after every two-block collision. Similarly, cryptanalytic collision attacks on the compression functions often 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 multiblock 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 $m[1] \equiv m^*[1] + \Delta \pmod{2^b}$ 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 $m^*[2] \equiv m[2] + \Delta \pmod{2^b}$ 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 $m[1] + m[2] \pmod{2^b} = \Delta + m^*[1] + m^*[2] - \Delta \pmod{2^b} = m^*[1] + m^*[2] \pmod{2^b}$. This is a collision in the additive checksum chain.

9 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 [19].

It is also fair to note that a checksum may still be a good way of protecting against shortcut attacks. As an example, although there is a theoretical collision attack on the full MD2 hash function [16], the real complexity of the attack might be greater than the birthday complexity (due to large memory requirements). When the checksum is omitted, however, there is a very efficient collision attack on MD2 [30].

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 how to develop 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 [20] to a two-block collision attack as described in this paper using the collision attack on the compression function [20]. Another possible research area is to construct alternative simple and efficient extensions to the MD construction that provide protection against all generic attacks.

Acknowledgments The authors would like to thank Barbara Guttman and Morris Dworkin for useful comments and suggestions on this paper. Praveen Gauravaram is supported by a grant numbered 274-09-0096 from the Danish Council for Independent Research: Technology and Production Sciences (FTP) and Natural Sciences (FNU). Søren S. Thomsen is supported by a grant from the Villum Kann Rasmussen Foundation. Portion of the results presented in this paper were published in [7].

References

1. Bellare, M., Micciancio, D.: A new paradigm for collision-free hashing: incrementality at reduced cost. In: Fumy, W. (ed.) *Advances in Cryptology—EUROCRYPT'97, Proceedings, Lecture Notes in Computer Science*, Vol. 1233, pp. 163–192 (1997)
2. Coppersmith, D.: Two broken hash functions. IBM research report RC 18397, IBM T. J. Watson Research Center, Yorktown Heights, N.Y., 10598, USA, October (1992)
3. Damgård, I.: A design principle for hash functions. In: Brassard, G. (ed.) *Advances in Cryptology—CRYPTO '89, Proceedings, Lecture Notes in Computer Science*, vol. 435 pp. 416–427. Springer (1990)
4. Dean, R.D.: Formal aspects of mobile code security. PhD thesis, Princeton University, January (1999)
5. Dunkelman, O., Preneel, B.: 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> (2009/11/27)
6. Filho, D., Barreto, P., Rijmen, V.: The MAELSTROM-0 hash function. Published at 6th Brazilian Symposium on Information and Computer System Security (2006)
7. Gauravaram, P., Kelsey, J.: Linear-XOR and additive checksums don't protect Damgård-Merkle hashes from generic attacks. In: Malkin, T. (ed.) *Topics in Cryptology—CT-RSA 2008, Lecture Notes in Computer Science*, vol. 4964, pp. 36–51. Springer, Berlin (2008)
8. Gauravaram, P., Millan, W., Dawson, E., Viswanathan, K.: Constructing secure hash functions by enhancing Merkle-Damgård construction. In: Batten, L.M., R. Safavi-Naini, L.M. (eds.) *Australasian Conference on Information Security and Privacy 2006, Proceedings, Lecture Notes in Computer Science*, vol. 4058, pp. 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 2009/11/27)
9. Gazzoni Filho, D.L., Barreto, P.S.L.M., Rijmen, V.: The maelstrom-0 hash function. Published at 6th Brazilian Symposium on Information and Computer System Security, August 28–September 1, Santos, Brazil (2006)

10. Hoch, J., Shamir, A.: Breaking the ICE: finding multicollisions in iterated concatenated and expanded (ICE) hash functions. In: Robshaw, M. (ed.) *Fast Software Encryption 2006*, Lecture Notes in Computer Science, vol. 4047, pp. 179–194. Springer, Berlin (2006)
11. Joux, A.: Multicollisions in iterated hash functions. Application to cascaded constructions. In: Franklin, M.K. (ed.) *Advances in Cryptology—CRYPTO 2004*, Proceedings, Lecture Notes in Computer Science, vol. 3152, pp. 306–316. Springer (2004)
12. Kaliski B.S. Jr.: The MD2 message-digest algorithm, April 1992. RFC 1319. Available: <http://www.ietf.org/rfc/rfc1319.txt> (2009/11/27) (1992)
13. Kelsey, J., Kohno, T.: Herding hash functions and the nostradamus attack. In: Vaudenay, S. (ed.) *Advances in Cryptology—EUROCRYPT 2006*, Proceedings, Lecture Notes in Computer Science, vol. 4004, pp. 183–200. Springer (2006)
14. Kelsey, J., Schneier, B.: Second preimages on n -bit hash functions for much less than 2^n work. In: Cramer, R. (ed.) *Advances in Cryptology—EUROCRYPT 2005*, Proceedings, Lecture Notes in Computer Science, vol. 3494, pp. 474–490. Springer (2005)
15. Knudsen, L.R., Mathiassen J.E.: Preimage and Collision Attacks on MD2. In: Gilbert, H., Handschuh, H. (eds.) *Fast Software Encryption 2005*, Proceedings, volume 3557 of *Lecture Notes in Computer Science*, pp. 255–267. Springer (2005)
16. Knudsen, L.R., Mathiassen, J.E., Muller, F., Thomsen S.S.: Cryptanalysis of MD2. (2009). doi:[10.1007/s00145-009-9054-1](https://doi.org/10.1007/s00145-009-9054-1)
17. Lai, X., Massey, J.L.: Hash functions based on block ciphers. In: Rueppel, R.A. (ed.) *Advances in Cryptology—EUROCRYPT '92*, Proceedings, Lecture Notes in Computer Science, vol. 658, pp. 55–70. Springer (1993)
18. Lei, D.: F-HASH: securing hash functions using feistel chaining. Cryptology ePrint Archive, Report 2005/430, Available: <http://eprint.iacr.org/2005/430.pdf> (2009/11/27) (2005)
19. Lucks, S.: A failure-friendly design principle for hash functions. In: Roy, B.K. (ed.) *Advances in Cryptology—ASIACRYPT 2005*, Proceedings, Lecture Notes in Computer Science, vol. 3788, pp. 474–494. Springer (2005)
20. Mendel, F., Pramstaller, N., Rechberger, C., Kontak, M., Szmidi, J.: Cryptanalysis of the GOST hash function. In: Wagner, D. (ed.) *Advances in Cryptology—CRYPTO 2008*, Proceedings, Lecture Notes in Computer Science, vol. 5157, pp. 162–178. Springer (2008)
21. Merkle, R.C.: One way hash functions and DES. In: Brassard, G. (ed.) *Advances in Cryptology—CRYPTO'89*, Proceedings, Lecture Notes in Computer Science, vol. 435, pp. 428–446. Springer (1990)
22. Muller, F.: The MD2 hash function is not one-way. In: Lee, P.J. (ed.) *Advances in cryptology—ASIACRYPT 2004*, Proceedings, Lecture Notes in Computer Science vol. 3329, pp. 214–229. Springer (2004)
23. Nandi, M., Stinson, D.: Multicollision attacks on some generalized sequential hash functions. *IEEE Trans. Inf. Theory* **53**(2), 759–767 (2007)
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. National Institute of Standards and Technology: FIPS PUB 180-3, Secure Hash Standard, (June 2007)
27. Preneel, B.: *Analysis and design of cryptographic hash functions*. PhD thesis, Katholieke Universiteit Leuven, (February 1993)
28. Quisquater, J.-J., Girault, M.: $2n$ -Bit hash-functions using n -bit symmetric block cipher algorithms. In: Quisquater, J.-J., Vandewalle, J. (eds.) *Advances in Cryptology—EUROCRYPT '89*, Proceedings, Lecture Notes in Computer Science, vol. 434, pp. 102–109. Springer (1990)
29. Rivest, R.L.: The MD5 message-digest algorithm, April 1992. RFC 1321. Available: <http://www.ietf.org/rfc/rfc1321.txt> (2009/11/27) (1992)
30. Rogier, N., Chauvaud, P.: MD2 is not secure without the checksum byte. *Des. Codes Cryptogr.* **12**(3), 245–251 (1997)
31. Rostekhregulirovaniye (Russia's Federal Agency for Technical Regulation and Metrology): GOST R 34.11-94: Information technology—cryptographic data security—hashing function (1994)
32. Wagner, D.: A generalized birthday problem. In: Yung, M. (ed.) *CRYPTO 2002*, Proceedings, Lecture Notes in Computer Science, vol. 2442, pp. 288–303. Springer (2002)
33. Wang, X., Yin, Y., Yu, H.: Efficient collision search attacks on SHA-0. In: Shoup, V. (ed.) *CRYPTO 2005*, Proceedings, Lecture Notes in Computer Science, vol. 3621, pp. 1–16. Springer (2005)
34. Wang, X., Yin, Y., Yu, H.: Finding collisions in the full SHA-1. In: Shoup V. (ed.) *CRYPTO 2005*, Proceedings, Lecture Notes in Computer Science, vol. 3621, pp. 17–36. Springer (2005)
35. Wang, X., Yu, H.: How to break MD5 and other hash functions. In: Cramer, R. (ed.) *EUROCRYPT 2005*, Proceedings, Lecture Notes in Computer Science, vol. 3494, pp. 19–35. Springer (2005)

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.