Linear-XOR and Additive Checksums Don't Protect Damgård-Merkle Hashes from Generic Attacks

Praveen Gauravaram[1]* and John Kelsey[2]

[1] Technical University of Denmark (DTU), Denmark
Queensland University of Technology (QUT), Australia.
p.gauravaram@mat.dtu.dk
[2] National Institute of Standards and Technology (NIST), USA
john.kelsey@nist.gov

Abstract. We consider the security of Damgård-Merkle variants which compute linear-XOR or additive checksums over message blocks, intermediate hash values, or both, and process these checksums in computing the final hash value. We show that these Damgård-Merkle variants gain almost no security against generic attacks such as the long-message second preimage attacks of [10, 21] and the herding attack of [9].

1 Introduction

The Damgård-Merkle construction [3, 14] (**DM** construction in the rest of this article) provides a blueprint for building a cryptographic hash function, given a fixed-length input compression function; this blueprint is followed for nearly all widely-used hash functions. However, the past few years have seen two kinds of surprising results on hash functions, that have led to a flurry of research:

1. *Generic attacks* apply to the **DM** construction directly, and make few or no assumptions about the compression function. These attacks involve attacking a t-bit hash function with more than $2^{t/2}$ work, in order to violate some property other than collision resistance. Examples of generic attacks are Joux multicollision [8], long-message second preimage attacks [10, 21] and herding attack [9].
2. *Cryptanalytic attacks* apply to the compression function of the hash function. However, turning an attack on the compression function into an attack on the whole hash function requires properties of the **DM** construction. Examples of cryptanalytic attacks that involve the

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

construction as well as the compression function include multi-block collisions on MD5, SHA-0 and SHA-1 [24–26].

These results have stimulated interest in new constructions for hash functions, that prevent the generic attacks, provide some additional protection against cryptanalytic attacks or both. The recent call for submissions for a new hash function standard by NIST [18] has further stimulated interest in alternatives to **DM**.

In this paper, we consider a family of variants of **DM**, in which a linear-XOR checksum or additive checksum is computed over the message blocks, intermediate states of the hash function, or both, and is then included in the computation of the final hash value. In a linear-XOR checksum, each checksum bit is the result of XORing together some subset of the bits of the message, intermediate hash states, or both. In an additive checksum, the full checksum is the result of adding together some or all of the message blocks, intermediate hash values, or both, modulo some N. In both cases, the final checksum value is processed as a final, additional block in computing the hash value.

Such **DM** variants can be seen as a special case of a cascade hash. Generic attacks such as the long-message second preimage attack or the herding attack appear at first to be blocked by the existence of this checksum. (For example, see [6] for the analysis of **3C** and MAELSTROM-0 against second preimage and herding attacks.)

Unfortunately, these **DM** variants turn out to provide very little protection against such generic attacks. We develop techniques, based on the multicollision result of Joux [8], which allow us to carry out the generic attacks described above, despite the existence of the checksum. More generally, our techniques permit the construction of a *checksum control sequence*, or CCS, which can be used to control the value of the checksum without altering the rest of the hash computation.

To summarize our results:

1. The generic multicollision, second preimage and herding attacks on **DM** hash functions can be applied to linear-XOR/additive checksum variants of **DM** at very little additional cost, using our techniques.
2. Our techniques are flexible enough to be used in many other situations. Some cryptanalytic attacks on the compression function of a hash, which the linear-XOR/additive checksum appears to stop from becoming attacks on the full hash function, can be carried out on the full hash function at a relatively little additional cost using our tech-

niques. Future generic attacks will almost certainly be able to use our techniques to control checksums at very low cost

1.1 Related Work

In unpublished work, Mironov and Narayan [15] developed a different technique to defeat linear-XOR checksums in hash functions; this technique is less flexible than ours, and does not work for long-message second preimage attacks. However, it is quite powerful, and can be combined with our technique in attacking hash functions with complicated checksums. In [8], Joux provides a technique for finding 2^k collisions for a **DM** 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. The linear-XOR and additive checksum variants of **DM** we consider in this paper can be seen as a special (weak) case of a cascade hash.

Multi-block collisions are an example of a cryptanalytic attack on a compression function, which must deal with the surrounding hash construction. Lucks [13] and Tuma and Joscak [22] have independently found that if there is a multi-block collision for a hash function with structured differences, concatenation of such a collision will produce a collision on **3C**, a specific hash construction which computes checksum using XOR operation as the mixing function. (**3C** does not prevent Joux multicollision attack over 1-block messages [6, 20].)

Nandi and Stinson [17] have shown the applicability of multicollision attacks to a variant of **DM** in which each message block is processed multiple times; Hoch and Shamir [7] extended the results of [17] showing that generalized sequential hash functions with any fixed repetition of message blocks do not resist multicollision attacks. The MD2 hash function which uses a non-linear checksum was shown to not satisfy preimage and collision resistance properties [11, 16]. Coppersmith [2] has shown a collision attack on a DES based hash function which uses two supplementary checksum blocks computed using XOR and modular addition of the message blocks. Dunkelman and Preneel [4] applied herding attack of [9] to cascade hashes; their technique can be seen as an upper bound on the difficulty of herding **DM** variants with checksums no longer than the hash outputs.

1.2 Impact

The main impact of our result is that new hash function constructions that incorporate linear-XOR/additive checksums as a defense against

generic attacks do not provide much additional security. Designers who wish to thwart these attacks need to look elsewhere for defenses. We can apply our techniques to specific hash functions and hashing constructions that have been proposed in the literature or are in practical use. They include **3C**, GOST, MAELSTROM-0 and F-Hash. Both our techniques and the generic attacks which they make possible require the ability to (at least) find many collisions for the underlying compression function of the hash, and so probably represent only an academic threat on most hash functions at present.

1.3 Guide to the Paper

This paper is organised as follows: First, we provide the descriptions of hash functions with linear checksums analysed in this paper. Next, we demonstrate new cryptanlytical techniques to defeat linear-XOR/additive checksums in these designs. We then provide a generic algorithm to carry out second preimage and herding attacks on these designs with an illustration on **3C**. We then demonstrate multi-block collision attacks on these designs. We then compare our cryptanalysis with that of [15]. Finally, we conclude the paper with some open problems.

2 The DM construction and DM with linear checksums

2.1 The DM Construction

The **DM** iterative structure [3, 14] shown in Figure 1 has been a popular framework used in the design of standard hash functions MD5, SHA-1, SHA-224/256 and SHA-384/512.

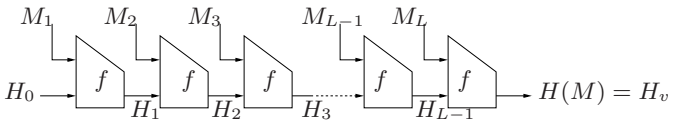

Fig. 1. The Damgård-Merkle construction

The message M, with $|M| \leq 2^l - 1$ bits, to be processed using a **DM** hash function H is always padded by appending it with a 1 bit followed by 0 bits until the padded message is l bits short of a full block of b bits. The last l bits are filled in with the binary encoded representation of the length

of true message M. This compound message is an integer multiple of b bits and is represented with b-bit data blocks as $M = M_1, M_2, \ldots M_L$. Each block M_i is processed using a fixed-length input compression function f as given by $H_i = f(H_{i-1}, M_i)$ where H_i from $i = 1$ to $L - 1$ are the intermediate states and H_0 is the fixed initial state of H. The final state $H_v = f(H_{L-1}, M_L)$ is the hash value of M.

2.2 Linear-XOR/additive checksum variants of DM

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

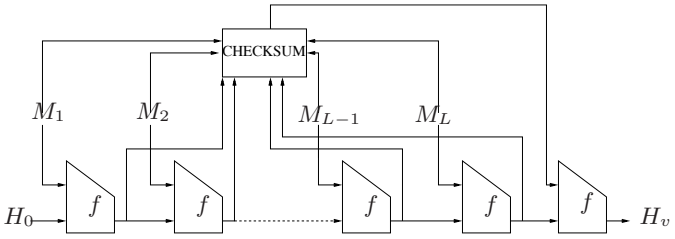

Fig. 2. Hash function structure with a linear-XOR/additive checksum

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

A 3-chain variant of **3C** called **3CM** is used as a chaining scheme in the MAELSTROM-0 hash function [5]. At every iteration of f in the iterative chain of **3CM**, the t-bit value in the third chain is updated using an LFSR. This result is then XORed with the data in the iterative chain at that iteration. All the intermediate states in the iterative chain of

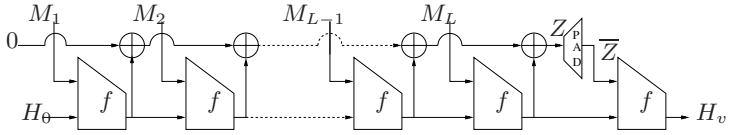

Fig. 3. The **3C**-hash function

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 [12], 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 [19]. The compression function f of GOST is iterated in the **DM** mode and a mod 2^{256} additive checksum is computed by adding all the 256-bit message blocks in an accumulation chain. We generalise our analysis of GOST by assuming that its f function has a block length of b bits and hash value of t bits.

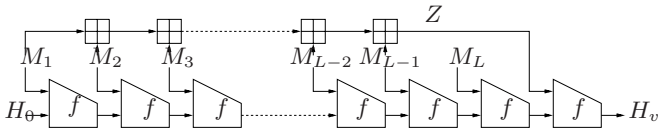

Fig. 4. GOST hash function

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

3 New techniques to defeat linear-XOR checksums

3.1 Extending Joux multicollisions on DM to multiple blocks

Let $C(s, n)$ be a collision finding algorithm on the compression function, where s denotes the state at which the collision attack is applied and n, the number of message blocks present in each of the colliding messages. On a t-bit hash function, a brute force $C(s, n)$ requires about $2^{t/2}$ hash function computations to find a collision with 0.5 probability whereas a cryptanalytic $C(s, n)$ requires less work than that. The Joux multicollision attack [8] finds a sequence of k collisions on a t-bit **DM** hash, to produce a 2^k collision with work only k times the work of a single collision search. For a brute-force collision search, this requires $k \times 2^{t/2}$ evaluations of the compression function. While it is natural to think of constructing such a multicollision from a sequence of single-message-block collisions, it is no more expensive to use the brute-force collision search to find a sequence of multi-message-block collisions.

3.2 Checksum control sequences

We define checksum control sequence (CCS) as a data structure which lets us to control the checksum value of the **DM** variant, *without altering the rest of the hash computation*. We construct the CCS by building a Joux multicollision 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 iterative chain, of the **DM** variant hash.

For example, a 2^k collision on the underlying **DM** construction of **3C**, in which the sequence of individual collisions is each two message blocks long, is shown in Figure 3. This multicollision 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], [23]. 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 $H_1^0 \oplus H_2$ or $H_1^1 \oplus H_2$ from the first 2-block collision and either $H_3^0 \oplus H_4$ or $H_3^1 \oplus H_4$ from the second 2-block collision of the CCS to control 2 bits of the checksum without changing the hash value after the CCS.

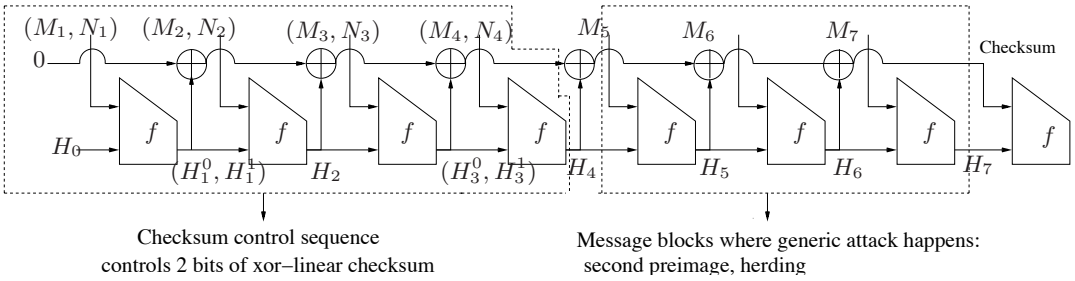

Fig. 5. Using CCS to control 2 bits of the checksum

3.3 Defeating linear-XOR checksum in hash functions

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

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

Steps:

1. Build a CCS for **3C** by constructing a 2^t 2-block collision on its underlying **DM** using a brute force $C(s, 2)$.
2. Each of the parts of the CCS gives one choice e_i^0 or e_i^1 for $i = 1, 2, \ldots, t$ to determine some random t-bit value that either is or is not XORed into the final checksum value at the end of the CCS. Now $e_i^0 = H_{2i-1}^0 \oplus H_{2i}^0$ and $e_i^1 = H_{2i-1}^1 \oplus H_{2i}^1$ for $i = 1, 2, \ldots, t$.
3. For any t-bit string $a = a[1], a[2], \ldots, a[t]$, let $e^a = e_1^{a[1]}, \ldots, e_t^{a[t]}$.
4. Find a such that $e_1^{a[1]} \oplus e_2^{a[2]} \oplus \ldots \oplus \ldots e_t^{a[t]} = D$. We solve the equation: $\bigoplus_{i=1}^{t} e_i^1 \times a[i] \oplus e_i^0 \times (1 - a[i]) = D$.
5. Each bit position of $e_i^{a[i]}$ gives one equation and turn the above into t equations, one for each bit. Let $\overline{a}[i] = 1 - a[i]$.

6. The resulting system is: $\bigoplus_{i=1}^{t} e_i^1[j] \times a[i] \oplus e_i^0[j] \times \overline{a}[i] = D[j]$ $(j = 1, \ldots, t)$. Here, there are t linear equations in t unknowns that need to be solved for the solution $a[1], a[2], \ldots, a[t]$ which lets us determine the blocks in the CCS that form the prefix giving the checksum D.

Work: It requires $t(2^{t/2} + 1)$ evaluations of the compression function to construct the CCS and at most $t^3 + t^2$ bit-XOR operations to solve a system of $t \times t$ equations using Gaussian elimination [1, Appendix A], [23].

Remark 1. Similarly, linear-XOR checksums can be defeated in F-Hash and **3CM**. If a linear-XOR checksum is computed using both the message blocks and intermediate states, linear equations due to XOR of the intermediate states and that of message blocks need to be solved.

4 New techniques to defeat additive checksums

Consider an additive checksum mod 2^k computed using messages for a **DM** hash function. It is possible to build a checksum control sequence as above, but both its construction and its use require some different techniques.

4.1 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 work required to construct a 2^k Joux multicollision. Using the CCS to control the checksum then requires negligible work.

In this algorithm, we construct a 2^k Joux multicollision, in which each successive 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 the power to add 1 to the checksum, the next allows us to add 2, the next 4, and so on until the checksum is entirely controlled[1].

ALGORITHM: Defeat additive checksum on GOST
Steps for Constructing the CCS:

1. Let $h =$ the initial value of the hash function
2. For $i = 0$ to $k - 1$:

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

(a) Let A, B be random blocks.
(b) For $j = 0$ to $2^{t/2} - 1$:
    i. $X[j] = A + j, B - j$
    ii. $X^*[j] = A + j + 2^i, B - j$
    iii. $Y[j] =$ hash of $X[j]$ starting from h
    iv. $Y^*[j] =$ hash of $X^*[j]$ starting from h
3. Search for a collision between list Y and Y^*. Let $u, v =$ values satisfying $Y[u] = Y^*[v]$.
4. $CCS[i] = X[u], X^*[u]$
5. $h = Y[u]$

Steps for Using the CCS:

Using the CCS is very simple; we determine the checksum we would get by choosing $X[0], X[1], X[2], ...,$ and then determine what we would need to add to that value to get the desired checksum value. We then use our control over the CCS to add the desired value.

1. Let $T =$ the checksum that is desired.
2. Let $Q =$ the checksum obtained by choosing $X[0], X[1], X[2], ..., X[k-1]$ as the message blocks of the CCS.
3. Let $D = T - Q$.
4. $M =$ an empty message (which will end up with the message blocks chosen from the CCS for this desired checksum).
5. For $i = k - 1$ down to 0:
    (a) If $D > 2^i$ Then:
        i. $M = M||X^*[i]$
        ii. $D = D - 2^i$
    (b) Else:
        i. $M = M||X[i]$

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

Work: Constructing the CCS requires k successive brute-force collision searches, each requiring $2^{t/2}$ work. For the specific parameters of the GOST hash, this is 256 successive 2^{129} collision searches, and so requires about 2^{137} work total. (The same CCS could be used for many different messages.) Controlling the checksum with the CCS requires negligible work.

4.2 Building a CCS with Random Message Blocks

If the message blocks are not under our control, or if hash chaining values or other values not under our direct control are used as inputs for the additive checksum, then our attack becomes much less efficient. However, we can still construct a CCS which will be efficient to use, by carrying out an algorithm which is based loosely on Joux's collision attack on cascade hashes.

The idea behind this algorithm is to construct k successive Joux multicollisions, each of $2^{k/2}$ possible message strings. Then, we carry out a collision search on the first $2^{k/2}$-multicollision for a pair of strings that will cause a difference of 1 in the additive checksum, a search on the second $2^{k/2}$-multicollision for a pair that will cause a difference of 2, and so on, until we have the ability to completely control the checksum without affecting the rest of the hash computation.

An algorithm to defeat additive checksum on a t-bit GOST hash function structure H shown in Figure 4 is given below:

ALGORITHM: Defeating checksum in GOST
Variables:

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

Steps:

1. For $i = 1$ to b:
   - For $j = 1$ to $(b/2) + 1$:
     - Find M_j^i and N_j^i such that $f(H_{j-1}^i, M_j^i) = f(H_{j-1}^i, N_j^i) = H_j^i$ where $H_0^1 = H_0$. That is, build a $(b/2) + 1$-block multicollision where each block yields a collision on the iterative chain and there are $2^{(b/2)+1}$ different $(b/2) + 1$-block sequences of blocks all hashing to the same intermediate state $H_{(b/2)+1}^i$ on the iterative chain.
   - Find a pair of paths from the different $(b/2) + 1$-block sequences whose additive checksum differs by 2^{i-1} as follows:
     - T = empty table.

- for $j = 1$ to $2^{(b/2)+1}$
  - $C_j^i \equiv \sum_{k=1}^{(b/2)+1} X_k^i \bmod 2^b$ where X_k^i can be M_k^i or N_k^i.
  - Add to T: $(C_j^i,\ 0,\ X_1^i||X_2^i||\ldots X_{(b/2)+1}^i)$
  - Add to T: $(C_j^i + 2^{i-1},\ 1,\ X_1^i||X_2^i||\ldots X_{(b/2)+1}^i)$.
- Search T to find colliding paths between the entries with 0 and 1 in the second column of T. Let these paths of $(b/2)+1$ sequence of blocks be e_i^1 and e_i^0 where $e_i^1 \equiv e_i^0 + 2^{i-1} \bmod 2^b$.

– $chunk[i] = (e_i^0, e_i^1)$.

2. Construct CCS by concatenating individual chunks each containing a pair of $(b/2)+1$ blocks that hash to the same intermediate state on the iterative chain. The CCS is $chunk[1] \mid\mid chunk[2] \ldots \mid\mid chunk[b]$.
3. The checksum at the end of the 2^b $(b/2)+1$-block collision can be forced to the desired checksum by choosing either of the sequences e_i^0 or e_i^1 from the CCS which is practically free to use and adding blocks in each sequence over modulo 2^b.

Work: Defeating additive checksum in GOST equals the work to construct b $2^{(b/2)+1}$ 1-block collisions plus the work to find a chunk in each $2^{(b/2)+1}$ 1-block collision. It is $b \times ((b/2)+1) \times 2^{t/2}$ evaluations of f and a time and space of $b \times 2^{b/2+1}$ for a collision search to find b chunks. For GOST, it is 2^{143} evaluations of f and a time and space of about 2^{137}.

Similarly, additive checksum mod 2^k for a **DM** hash using intermediate states can be defeated by constructing a CCS with a 2^k Joux multicollision over 2-block messages. For a **DM** hash with additive checksum mod 2^k computed using both the message blocks and intermediate states, a $2^{(k/2)+1}$ Joux multicollision using 2-block messages is performed to find a pair of messages (resp. intermediate states) within the multicollision whose additive checksum differs by any desired value. This can be done by generating all possible $2^{(k/2)+1}$ checksum values due to messages (resp. intermediate states) from the multicollision, and doing a modified collision search for a pair of messages (resp. intermediate states) whose additive difference is the desired value.

5 Generic attacks

The fundamental approach used to perform the generic attacks on all the hash functions with linear checksums is similar. Hence, we discuss it here only for **3C**. Broadly, it consists of the following steps:

1. Construct a CCS and combine it with whatever other structures such as expandable message, diamond structure (or vice versa for some attacks) for the generic attack to work.

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

To find a 2^k-2-block collision on **3C**, first find a 2^k-2-block collision on the iterative chain of **3C** and construct CCS from this end. By defeating each possible 2^k checksum value to a fixed checksum, we can get a 2^k-collision for **3C**. Constructing and using the CCS does not imply random gibberish in the messages produced; using Yuval's trick [27], a brute-force search for the multicollision used in the CCS can produce collision pairs in which each possible message is a plausible-looking one. This is possible when the CCSs to defeat the checksums are constructed from individual collisions as in (Dear Fred/Freddie,)(Enclosed please find/I have sent you) (a check for $100.00/a little something) and so on, where we can choose either side of the slash for the next part of the sentence. In that case, any choice for the CCS used to defeat the checksum will be a meaningful message.

5.1 Long-message second preimage attack on 3C

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

1. M_{target} : the target long message.
2. M_{link} : linking block connecting the intermediate state at the end of the *expandable message* to an intermediate state of the target message.
3. H_{exp} : the intermediate state at the end of the *expandable message*.
4. H_t : the intermediate hash value at the end of the CCS.
5. M_{sec} : the second preimage for H of the same length as M_{target}.
6. M_{pref} : the checksum control prefix obtained from the CCS.

Steps:

1. Compute the intermediate hash values for M_{target} using H:
   - H_0 and h_0 are the initial states of the iterative and accumulation chains respectively.
   - M_i is the i^{th} message block of M_{target}.
   - $H_i = f(H_{i-1}, M_i)$ and $h_i = H_i \oplus h_{i-1}$ are the i^{th} intermediate states on the iterative and accumulation chains respectively.

- The intermediate states on the iterative and accumulation chains are organised in some searchable structure for the attack, such as hash table. The hash values $H_1, \ldots, H_d$ and those obtained in the processing of t 2-block messages are excluded from the hash table.

2. Build a CCS for H by constructing a 2^t 2-block collision starting from H_0. Let H_t be the multicollision value and h_t be the corresponding checksum value which is random.
3. Construct a $(d, d + 2^d - 1)$-*expandable message* M_{exp} with H_t as the starting state using either of the *expandable message* construction methods [10]. Append M_{exp} to the CCS and process it to obtain H_{exp}.
4. Find M_{link} such that $f(H_{exp}, M_{link})$ collides with one of the intermediate states on the iterative chain stored in the hash table while processing M_{target}. Let this matching value of the target message be H_u and the corresponding state in the accumulation chain be h_u where $d + 2t + 1 \leq u \leq 2^d + d + 2t + 1$.
5. Use the CCS built in step 2 to find the checksum control prefix M_{pref} which adjusts the state in the accumulation chain at that point to the desired value h_u of M_{target}. This is equivalent to adjusting the checksum value at the end of the CCS.
6. Expand the *expandable message* to a message M^* of $u-1$ blocks long.
7. Return the second preimage $M_{sec} = M_{pref}||M^*||M_{link}||M_{u+1} \ldots M_{2^d+d+1+2t}$ of the same length as M_{target} such that $H(M_{sec}) = H(M_{target})$.

Work: The work to find a second preimage on **3C** equals the work to construct the CCS plus the work to solve a system of $t \times t$ linear equations plus the work to do the second preimage attack on **DM**. Note that constructing and using the CCS is very fast compared to the rest of the attack.
Illustration: Using generic-expandable message algorithm [10], the work to find a second preimage for **3C-SHA-256** for a target message of $2^{54} + 54 + 512 + 1$ blocks is $2^{136} + 54 \times 2^{129} + 2^{203}$ SHA-256 compression function evaluations and $2^{24} + 2^{16}$ bit-XOR operations assuming abundant memory.

5.2 Herding attack on 3C

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

1. Construct a 2^d hash value wide diamond structure for H and output the hash value H_v as the chosen target which is computed using any of the possible 2^{d-1} checksum values or some value chosen arbitrarily. Let h_c be that checksum value.

2. Build a CCS for H using a 2^t collision over 2-block messages. Let H_t be the intermediate state due to this multicollision on H.
3. When challenged with the prefix message P, process P using H_t. Let $H(H_t, P) = H_p$.
4. Find a linking message block M_{link} such that $H(H_p, M_{link})$ collides with one of the 2^d outermost intermediate states on the iterative chain in the diamond structure. If it is matched against all of the $2^{d+1}-2$ intermediate states in the diamond structure then a $(1, d+1)$-*expandable message* must be produced at the end of the diamond structure to ensure that the final herded message is always a fixed length.
5. Use the CCS computed in step 2 to force the checksum of the herded message P to h_c. Let M_{pref} be the checksum control prefix.
6. Finally, output the message $M = M_{pref}||P||M_{link}||M_d$ where M_d are the message blocks in the diamond structure that connect $H(H_p, M_{link})$ to the chosen target H_v. Now $H_v = H(M)$.

Work: The work to herd **3C** equals the work to build the CCS plus the work to solve the system of equations plus the work to herd **DM** [9]. This equals about $t \times 2^{t/2} + 2^{t/2+d/2+2} + d \times 2^{t/2+1} + 2^{t-d-1}$ evaluations of f and $t^3 + t^2$ bit-XOR operations assuming that all of the $2^{d+1} - 2$ intermediate states are used for searching in the diamond structure. Note that the work to build and use the CCS is negligible compared to the rest of the attack.
Illustration: Herding **3C-SHA-256** with $d = 84$ requires $2^{136} + 2^{172} + 84 \times 2^{129} + 2^{171}$ evaluations of SHA-256 compression function and $2^{24} + 2^{16}$ bit-XOR operations.

6 On carrying out generic attacks using collision attacks

We note that it is difficult to construct the CCSs using cryptanalytic $C(s, n)$ such as the ones built on MD5 and SHA-1 [25,26] in order to defeat linear checksums to carry out generic attacks. For example, consider two 2-block colliding messages of format $(M_{2.i-1}, M_{2.i})$,$(N_{2.i-1}, N_{2.i})$ for $i = 1, \ldots, t$ on the underlying **MD** of **3C** based on near collisions due to the first blocks in each pair of the messages. Usually, the XOR differences of the nearly collided intermediate states are either fixed or very tightly constrained as in the collision attacks on MD5 and SHA-1 [25, 26]. It is difficult to construct a CCS due to the inability to control these fixed or constrained bits. Similarly, it is also difficult to build the CCSs using colliding blocks of format $(M_{2.i-1}, M_{2.i})$,$(N_{2.i-1}, M_{2.i})$. It is not possible to control the checksum due to 2-block collisions of the format $(M_{2.i-1}, M_{2.i})$,

$(M_{2.i-1}, N_{2.i})$ [24] as this format produces a zero XOR difference in the checksum after every 2-block collision.

Though we cannot perform generic attacks on this class of hash functions using structured collisions, we can find multi-block collisions by concatenating two structured collisions. Consider a collision finding algorithm $C(s, 1)$ with $s = H_0$ for the GOST hash function H. A call to $C(s, 1)$ results in a pair of b-bit message blocks (M_1, N_1) such that $M_1 \equiv N_1 + \Delta \bmod 2^b$ and $f(H_0, M_1) = f(H_0, N_1) = H_1$. Now call $C(s, 1)$ with $s = H_1$ which results in a pair of blocks (M_2, N_2) such that $N_2 \equiv M_2 + \Delta \bmod 2^b$ and $f(H_1, M_2) = f(H_1, N_2) = H_2$. That is, $H(H_0, M_1||M_2) = H(H_0, N_1||N_2)$. Consider $M_1 + M_2 \bmod 2^b = \Delta + N_1 + N_2 - \Delta \bmod 2^b = N_1 + N_2 \bmod 2^b$, a collision in the chain which computes additive checksum.

7 Comparison of our techniques with that of [15]

Independent to our work, Mironov and Narayanan [15] have found an alternative technique to defeat linear-XOR checksum computed using message blocks. We call this design GOST-x. While our approach to defeat the XOR checksum in GOST-x requires finding a 2^b collision using b random 1-block messages (M_i, N_i) for $i = 1$ to b, their technique considers repetition of the same message block twice for a collision. In contrast to the methods presented in this paper for solving system of linear equations for the whole message, their approach solves the system of linear equations once after processing every few message blocks. We note that this constrained choice of messages would result in a zero checksum at the end of the 2^b multicollision on this structure and thwarts the attempts to perform the second preimage attack on GOST-x. The reason is that the attacker loses the ability to control the checksum after finding the linking message block from the end of the expandable message which matches some intermediate state obtained in the long target message.

However, we note that their technique with a twist can be used to perform the herding attack on GOST-x. In this variant, the attacker chooses the messages for the diamond structure that all have the same effect on the linear-XOR checksum. These messages would result in a zero checksum at every 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 [15] is applied to defeat checksums in **3C**, **3CM**

and F-Hash, the 2^t 2-block collision finding algorithm used to construct the CCS must output the same pair of message blocks on the either side of the collision whenever it is called. This constraint is not there in our technique, and the approach of [15] is not quite as powerful. However, it could be quite capable of defeating linear-XOR checksums in many generic attacks. Because it is so different from our technique, some variant of this technique might be useful in cryptanalytic attacks for which our techniques do not work.

8 Concluding remarks

Our research leaves a number of questions open. Among these, the most interesting is, *how much security can be added by adding a checksum to* ***DM*** *hashes?* Our work provides a lower bound; for linear-XOR and additive checksums, very little security is added. Joux's results on cascade hashes [8] and more recent results of [4] provide an upper bound, since a checksum of this kind can be seen as a kind of cascade hash.

The other open question is on the properties that would ensure that a checksum would thwart generic attacks, and thus be no weaker than a cascade hash with a strong second hash function. The inability to construct a CCS for the checksum with less work than the generic attack is necessary but apparently not sufficient to achieve this goal, since we cannot rule out the possibility of other attacks on checksums of this kind, even without a CCS. The final open question is on how our techniques might be combined with cryptanalytic attacks on compression functions. It appears to be possible to combine the construction and use of a CCS with some kinds of cryptanalytic attacks, but this depends on fine details of the cryptanalysis and the checksum used.

Acknowledgments: We thank Gary Carter, Ed Dawson, Morris Dworkin, Jonathan Hoch, Barbara Guttman, Lars Knudsen, William Millan, Ilya Mironov, Heather Pearce, Adi Shamir, Tom Shrimpton, Martijn Stam, Jiri Tuma and David Wagner for comments on our work.

References

1. M. Bellare and D. Micciancio. A new paradigm for collision-free hashing: Incrementality at reduced cost. In *EUROCRYPT*, volume 1233 of *LNCS*, pages 163–192, 1997.
2. D. Coppersmith. Two Broken Hash Functions. Technical Report IBM Research Report RC-18397, IBM Research Center, October 1992.
3. I. Damgård. A Design Principle for Hash Functions. In *CRYPTO*, volume 435 of *LNCS*, pages 416–427. Springer-Verlag, 1989.

4. O. Dunkelman and B. Preneel. Generalizing the herding attack to concatenated hashing schemes. Presented at ECRYPT hash function workshop, 2007.
5. D.G. Filho, P. Barreto, and V. Rijmen. The MAELSTROM-0 Hash Function. Published at 6$^{\text{th}}$ Brazilian Symposium on Information and Computer System Security, 2006.
6. P. Gauravaram. *Cryptographic Hash Functions: Cryptanalysis, Design and Applications*. PhD thesis, Information Security Institute, QUT, June 2007.
7. J. Hoch and A. Shamir. Breaking the ICE: Finding Multicollisions in Iterated Concatenated and Expanded (ICE) Hash Functions. In *FSE*, volume 4047 of *LNCS*, pages 179–194. Springer, 2006.
8. A. Joux. Multicollisions in Iterated Hash Functions. Application to Cascaded Constructions. In Matt Franklin, editor, *CRYPTO*, volume 3152 of *LNCS*, pages 306–316. Springer, August 15–19 2004.
9. J. Kelsey and T. Kohno. Herding Hash Functions and the Nostradamus Attack. In *EUROCRYPT*, volume 4004 of *LNCS*, pages 183–200. Springer, 2006.
10. J. Kelsey and B. Schneier. Second Preimages on n-bit Hash Functions for Much Less than 2n̂ Work. In *EUROCRYPT*, volume 3494 of *LNCS*, pages 474–490. Springer, 2005.
11. L. Knudsen and J. Mathiassen. Preimage and Collision attacks on MD2. In *FSE*, volume 3557 of *LNCS*, pages 255–267. Springer, 2005.
12. D. Lei. F-HASH: Securing Hash Functions Using Feistel Chaining. Cryptology ePrint Archive, Report 2005/430, 2005.
13. S. Lucks. Hash Function Modes of Operation. ICE-EM RNSA 2006 Workshop at QUT,Australia., June 2006.
14. R. Merkle. One way Hash Functions and DES. In *CRYPTO*, volume 435 of *LNCS*, pages 428–446. Springer-Verlag, 1989.
15. I. Mironov and A. Narayanan. Personal communication, August 2006.
16. M. Muller. The MD2 Hash Function Is Not One-Way. In *ASIACRYPT*, volume 3329 of *LNCS*, pages 214–229. Springer, 2004.
17. M. Nandi and D. Stinson. Multicollision attacks on some generalized sequential hash functions. Cryptology ePrint Archive, Report 2006/055, 2006.
18. NIST. Cryptographic Hash Algorithm Competition, November 2007. See announcement at `http://www.csrc.nist.gov/groups/ST/hash/sha-3/index.html`.
19. Government Committee of the Russia for Standards. GOST R 34.11-94, 1994.
20. P.Gauravaram, W.Millan, E.Dawson, and K.Viswanathan. Constructing Secure Hash Functions by Enhancing Merkle-Damgård Construction. In *ACISP*, volume 4058 of *LNCS*, pages 407–420, 2006.
21. R.D.Dean. *Formal Aspects of Mobile Code Security*. PhD thesis, Princeton University, 1999.
22. J. Tuma and D. Joscak. Multi-block Collisions in Hash Functions based on 3C and 3C+ Enhancements of the Merkle-Damgård Construction. In *ICISC*, volume 4296 of *LNCS*, pages 257–266, 2006.
23. D. Wagner. A Generalized Birthday Problem. In *CRYPTO*, volume 2442 of *LNCS*, pages 288–303. Springer, 2002.
24. X. Wang, Y.L. Yin, and H. Yu. Efficient collision search attacks on SHA-0. In *CRYPTO*, volume 3621 of *LNCS*, pages 1–16. Springer, 2005.
25. X. Wang, Y.L. Yin, and H. Yu. Finding collisions in the full SHA-1. In *CRYPTO*, volume 3621 of *LNCS*, pages 17–36. Springer, 2005.
26. X. Wang and H. Yu. How to Break MD5 and Other Hash Functions. In *EUROCRYPT*, volume 3494 of *LNCS*, pages 19–35. Springer, 2005.
27. G. Yuval. How to swindle Rabin. *Cryptologia*, 3(3):187–189, July 1979.