

Constructing an Ideal Hash Function from Weak Ideal Compression Functions

Moses Liskov

Computer Science Department
The College of William and Mary
Williamsburg, Virginia, USA
`mliskov@cs.wm.edu`

Abstract. We introduce the notion of a *weak* ideal compression function, which is vulnerable to strong forms of attack, but is otherwise random. We show that such weak ideal compression functions can be used to create secure hash functions, thereby giving a design that can be used to eliminate attacks caused by undesirable properties of compression functions.

We prove that the construction we give, which we call the “zipper hash,” is *ideal* in the sense that the overall hash function is indistinguishable from a random oracle when implemented with these weak ideal building blocks.

The zipper hash function is relatively simple, requiring two compression function evaluations per block of input, but it is not streamable. We also show how to create an ideal (strong) compression function from ideal weak compression functions, which can be used in the standard iterated way to make a streamable hash function.

Keywords: Hash function, compression function, Merkle-Damgård, ideal primitives, non-streamable hash functions, zipper hash.

1 Introduction

The design of hash functions is a long-studied problem that has become recently more relevant because of significant attacks against commonly-used hash functions [22,20,21,19,1]. It is much easier to create *collision functions*, which take input of a particular size and produce output of a reduced size, than a full hash function directly. It is common practice to follow the basic concept of the Merkle-Damgård construction [6,14]: composing a compression function with itself, each time incorporating a block of the message, until the entire message is processed. If f is the compression function and x is an input divisible into l blocks of the appropriate size, then

$$H(x) = f(x_l, f(x_{l-1}, \dots, f(x_1, IV))) \dots$$

is the basic iterated hash function. There are two main ways in which this basic method has evolved: first of all, to handle messages of arbitrary length, a message may have to be padded so that the block size divides the length. In addition, the length of the initial message is included in the padding: this, along with fixing

an IV , is called Merkle-Damgård strengthening. Second, a finalization function g is often used after all the message blocks have been processed. Among other properties, this allows the output size of the compression function to be different from the output size of the hash function.

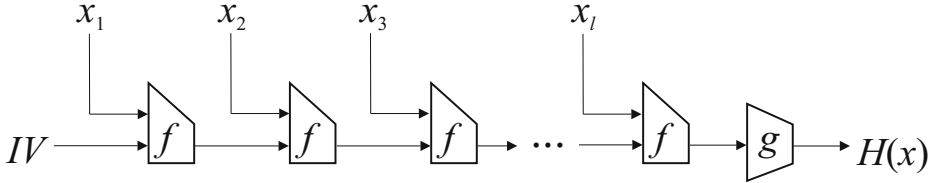


Fig. 1. The modern iterated hash function

The iterated hash function construction is elegant and natural, and is additionally attractive in that it is *streamable*, that is, a message may be hashed piece by piece with a small, finite amount of memory. Furthermore, this construction is known to be collision-resistant as long as the underlying compression function is collision-resistant [6,14]. However, there are reasons to question the iterated hash function design now.

An underlying theme in the recent high-profile attacks on hash functions has been the use of weaknesses in the compression function to build up an effective attack against the overall hash function. Furthermore, many attacks have been published recently that accomplish interesting black-box attacks against iterated hash functions once compression-function weaknesses have been found.

Here we summarize some known black-box attacks against iterated hash functions. Let n be the length of the output of a hash function H .

- **Second collision attack.** The basic attack goal here is to find a second collision on H once we have found a first collision on H . In a well-known attack, this is trivial for basic iterated hash functions: if $H(x) = H(y)$ then for all strings z , $H(x||z) = H(y||z)$ is another collision. Merkle-Damgård strengthening does not solve this problem completely, since the attack still works if $|x| = |y|$ and z contains the correct padding. [16,13]
- **Joux multicollision attack** [10]. It is easier than expected to find multicollisions: that is, a set of many distinct inputs that all hash to the same value. For a generic hash function, finding a t -way collision should require hashing an expected $2^{n \cdot (t-1)/t}$ messages. However, Joux showed that finding a t -way collision can also be done by making $(\log_2 t)2^{k/2}$ compression function queries, where k is the output size of the compression function. Essentially, the attack is to find one-block collisions for the compression function that can be chained together (by a brute force birthday attack). Once we have r such collisions, we can generate a 2^r -way collision by choosing one input for each colliding pair.
- **Fixed-point attack** [12,7]. The goal here is to come up with a second preimage for one of a set of known messages. If the target set is of size 2^t ,

it is easy to see that a second preimage can be found in a generic attack in time 2^{n-t} . This attack improves upon this by finding *expandable messages* based on fixed points for the collision function. Among other examples, the compression function in any Davies-Meyer block cipher-based hash function (such as the SHA family as well as MD4 and MD5) is susceptible to fixed-point attacks[12]. This allows an attack where, after hashing 2^t message *blocks*, a second preimage can be found in time $t2^{n/2} + 1 + 2^{n-t+1}$. Fixed points are used to circumvent Merkle-Damgård strengthening; with fixed points, one can build “expandable messages,” which let us recover a second preimage of the correct length.

- **The “herding” attack** [11]. This is an attack against the use of a hash function for commitments. The idea is to find a 2^t -way collision at a value $H(x)$, and then find a preimage of a commitment $H(x)$ that starts with an arbitrary z by trying random values y until $H(z||y)$ is one of the 2^t -way collisions.

In order to combat attacks like the Joux attack and the Kelsey-Kohno herding attack, Lucks proposed that the internal state of an iterated hash function should be larger than the output, thus preventing the usefulness of finding compression function collisions by brute force [13]. Lucks proposed *double-pipe hash* as a way to implement this, using two parallel compression function computations per block of message, in order to increase the size of the internal state. Lucks proved that, assuming the underlying compression function was ideal (i.e., a random oracle), the double-pipe hash compression function yields a collision-resistant hash function.

At the core of Lucks’ paper, however, was an even more important idea: that we should attempt to design hash functions that remain secure even when the compression functions on which they are based can be attacked.

We seek to improve on the work of Lucks in two ways. First, following the work of Coron, Dodis, Malimaud, and Puniya [4], we will prove that our construction is not only collision-resistant, but in fact indistinguishable from a random oracle, assuming the building blocks are ideal. Coron et al. show that the basic Merkle-Damgård construction is not ideal in the sense that even with an ideal compression function, it is impossible to prove that the hash function is indistinguishable from a random oracle. However, with an ideal finalization function (among other alternate modifications), iterated hash functions can be shown to be indistinguishable from a random oracle when implemented with ideal components. Assuming individual components to be ideal has been established as a reasonable model for the analysis of hash functions for some time [2]. The work of Coron et al. has set a higher standard for hash functions analyzed on the basis of ideal primitives, and we aspire to that standard.

Second, Lucks only attempts to make a hash function resilient to brute-force collision attacks against the compression function. It would be better to make a hash function resilient to actual *flaws* in the compression function as well. Therefore, we will weaken our assumptions about the underlying compression function as much as possible. We will still consider an ideal form of a compression function, but we will explicitly allow attacks against it, in order to model a weak but minimally secure compression function.

1.1 Our Results

In this paper, we formalize the notion of a weak ideal compression function, and show that such compression functions can be used to make stronger ideal primitives. Namely, we give a construction we call the “zipper hash” that makes an ideal hash function from weak ideal compression functions. The zipper hash is a very simple and elegant design; it requires $2l$ compression function evaluations for an l -block input. (Additionally, this concept of a weak ideal primitive may be of independent interest.)

Then, we go on to use weak ideal compression functions to make an ideal compression function. This construction is based on the zipper hash, and requires four compression function evaluations to run. We show that the Lucks double-pipe compression function is *not* an ideal construction, but offer a simple modification of it that is ideal. Thus, the compression function we consider comparable requires eight underlying compression function evaluations per block of input.

Finally, we analyze the efficiency of our schemes. We go on to make a case for considering non-streamable hash functions like our zipper hash in practice. We note that streamable hash functions (with constant-size state) always follow the essential Merkle-Damgård structure, so to avoid general attacks against iterated hash functions, one must consider non-streamable hash functions.

2 Notation and Definitions

2.1 Hash Functions and Compression Functions

Before we explore these issues, we must give a basic introduction to the concept of hash functions and compression functions. An n -bit *hash function family* is a family of functions $H : \mathcal{K} \times \{0, 1\}^* \rightarrow \{0, 1\}^n$ where \mathcal{K} represents the set of “keys” from which one is chosen at random. Note that hash functions must be defined as families: any specific hash function $H : \{0, 1\}^* \rightarrow \{0, 1\}^n$ cannot be *totally* collision-resistant, because a collision $H(x) = H(x')$ exists, and the algorithm that merely outputs (x, x') would always find it. Thus, we imagine that the hash function we use is randomly drawn from a larger family, and the “key” represents the individual member of the family. Note that we do not think of the key as secret: indeed, once the representative is chosen, the key will be known to all.

Compression functions must also be defined in terms of families. An (m, k) -bit *compression function family* is a function $f : \mathcal{K}_f \times \{0, 1\}^m \times \{0, 1\}^k \rightarrow \{0, 1\}^k$. Again, here, \mathcal{K}_f represents the set of keys for the compression function.

2.2 Ideal Hash Functions and Compression Functions

Typically, an ideal n -bit hash function is thought of as a random function $H : \{0, 1\}^* \rightarrow \{0, 1\}^n$. Here, there is no notion of key; the idea of choosing a random key for the hash function is abstracted away, represented as part of the randomness in the oracle.

An ideal (m, k) -bit compression function, similarly, is a random function $f : \{0, 1\}^m \times \{0, 1\}^k \rightarrow \{0, 1\}^k$.

2.3 Ideal Weak Compression Functions

In our construction we do not want to go so far as to assume that the compression functions are random oracles, as this would imply that they are collision resistant, and immune to all forms of attack. Instead, we will model our ideal compression function as a random oracle with additional *attack oracles* that provide results of successful attacks, and yet *still* give answers consistent with a random oracle.

This can be implemented in a variety of ways, depending on what the attack oracle does. We imagine that there is an oracle for the compression function f , so that on a new query (x, y) , a random output value z is returned. The following list describes the attack oracles for a variety of compression function security levels.

- **Ideal compression function.** No attack oracle, only the f oracle.
- **Collision-tractable compression function:** On invoking the attack oracle with no input, the oracle returns random values (x, x', y, y', z) such that $f(x, y) = z = f(x', y')$ where $(x, y) \neq (x', y')$.¹
- **Second preimage-tractable compression function:** On invoking the attack oracle on input (x, y) , the oracle returns a random pair of values (x', y') such that $f(x', y') = f(x, y)$.
- **Preimage-tractable compression function:** On invoking the attack oracle on input z , the oracle returns a random pair of values (x, y) such that $f(x, y) = z$.
- **Partially-specified preimage-tractable compression function:** On invoking the attack oracle on input (x, z) , the oracle returns a random value y such that $f(x, y) = z$.
- **Two-way partially-specified preimage-tractable compression function:** There are two attack oracles. On querying the first (called f^{-1}) on input (x, z) , the oracle returns a random value y such that $f(x, y) = z$. On querying the second (called f^*) on input (y, z) , the oracle returns a random value x such that $f(x, y) = z$.

This last form of ideal compression function we will name for convenience a *weak ideal compression function*. It should be clear that we can implement any form of compression function higher on the list with a weak ideal compression function (for instance, to implement the attack oracle for a preimage-tractable compression function, on input z , we pick a random x and query our first attack oracle on (x, z) to obtain y , then return (x, y)).

In fact, this form of weak compression function is susceptible to every form of (black-box) attack we are aware of.² An ideal weak compression function cannot

¹ That is, x, x', y, y' , and z are generated at random; if known values of f do not prohibit the property $f(x, y) = z = f(x', y')$, then those outputs are given, otherwise new ones are selected until known values of f do not cause a problem. Once the attack oracle returns a query, it affects how f will respond to (x, y) or (x', y') .

² Of course, we cannot capture non-black-box attacks when we try to view our primitives as ideal.

be used simply in an iterated way to make a hash function. For instance, if the padding function appends padding that depends only on the length of the input, we can find a collision by creating a random m -bit message x , computing $z = f(x, IV)$, and then querying the attack oracle $f^*(IV, z)$ to get a random x' such that $f(x', IV) = z$. Then, since the padding changes x and x' in the same way (because they are the same length), $H(x)$ and $H(x')$ will be the same, as they collide after one block, and the remaining blocks are the same.

Nonetheless, there is cryptographic strength implied in this notion of an ideal weak compression function, because despite the attacks we explicitly allow against it, we still imagine that the results of such attacks will be random and out of the control of the adversary.

Note that we are being quite generous with our attack oracles here. For an actual compression function, there is no guarantee that (for instance) a y such that $f(x, y) = z$ even exists, let alone many such y .³

2.4 Ideal Hash Functions and Compression Functions Based on Weak Ideal Compression Functions

Following Coron et al. [4], and paraphrasing closely from their paper, we will use the following methodology to prove that our constructions are sound. Let C be a Turing machine with access to an oracle: C will represent the *construction* and its oracle(s) will represent the ideal primitive the construction is made from.

Let Γ represent the oracle(s) for the underlying ideal primitive(s), and let Δ represent the oracle(s) for the ideal version of the primitive we try to construct with C .

We say that C is (t_A, t_S, q, ϵ) -indifferentiable from Δ if there is a simulator S such that for all distinguishers A ,

$$|Pr[A^{C, \Gamma} = 1] - Pr[A^{\Delta, S} = 1]| < \epsilon,$$

where (1) S answers as many different types of oracle queries as Γ provides, and S has oracle access to Δ and runs in time at most t_S , and (2) A runs in time at most t_A and makes at most q queries of its various oracles. We say that C is computationally indifferentiable from Δ if for all security parameters α it holds that C is $(t_A(\alpha), t_S(\alpha), q(\alpha), \epsilon(\alpha))$ -indifferentiable from Δ , where t_A and t_S are polynomial in α , where $q(\alpha) \leq t_A(\alpha)$, and where ϵ is negligible in α . We say that C is statistically indifferentiable if for all security parameters α it holds that C is $(t_A(\alpha), t_S(\alpha), q(\alpha), \epsilon(\alpha))$ -indifferentiable from Δ , where t_S and q are polynomial in α , and where ϵ is negligible in α .⁴

³ It may be more reasonable to think of our ideal compression function as a random quasigroup: that is, for every (x, z) there is a unique random y such that $f(x, y) = z$, and similarly, for every (y, z) there is a unique random x . However, we proceed under the more general attack oracle.

⁴ In other words, we no longer restrict the running time of the adversary, but we still restrict the number of queries.

3 The Zipper Hash Construction

The zipper hash is a general hash function construction. To build an n -bit hash function, we need two independent (m, k) -bit compression functions f_0 and f_1 , as well as a padding function P , an initialization vector IV , and a finalization function $g : \{0, 1\}^k \rightarrow \{0, 1\}^n$. On input x , P is guaranteed to return a value such that $x||P(x)$ is a string that can be broken down into m -bit blocks, and for all $x \neq x'$, $x||P(x) \neq x'||P(x')$. Given all these pieces, the zipper hash function works as follows:

1. Let x_1, \dots, x_l be m -bit strings such that $x_1||\dots||x_l = x||P(x)$.
2. H_1 is computed as $f_0(x_1, IV)$, and H_2, \dots, H_l are computed iteratively as $H_i = f_0(x_i, H_{i-1})$.
3. H'_1 is computed as $f_1(x_l, H_l)$, and H'_2, \dots, H'_l are computed iteratively as $H'_i = f_1(x_{l-i+1}, H'_{i-1})$.
4. Output $H(x) = g(H'_l)$.

This construction is called the zipper hash as its structure is reminiscent of a zipper. See figure 2. Note that although we require two independent compression functions, we can implement two independent weak ideal compression functions with a single one; see Appendix A.

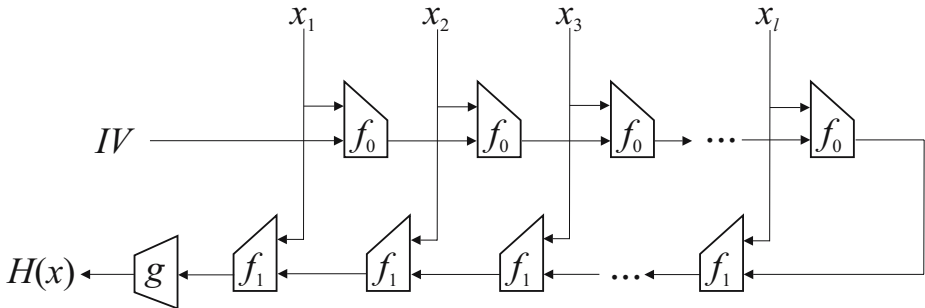


Fig. 2. The zipper hash function

4 Security

Let C be the Turing machine that implements the zipper hash. We will prove the following theorem:

Theorem 1. C is statistically indifferentiable from an ideal hash function Δ , using two ideal weak compression functions represented by Γ , where g is the identity function.

This will prove that the zipper hash, with g being the identity function, is indistinguishable from a random oracle. If g is not the identity function, then the

overall zipper hash will be indistinguishable from $g \circ \Delta$. If, for instance, g has the property that it produces a random output on a random input, this will also be an ideal hash.

To briefly sketch the proof, the simulator answers oracle queries for the weak ideal compression functions randomly, except when a query is the last one needed to compute the hash function on some value, in which case the simulator assumes that the query was in the forward direction for the last compression function evaluation, and queries Δ and gives this value. It is nontrivial to show that the simulator can always determine when a query amounts to the last one needed to compute the hash function, but with careful record-keeping, we can do it in polynomial time.

We will then make the assumption that no unexpected coincidences occur: that is, for instance, if (u, v) is given as a query to an oracle of Γ , that the randomly generated answer w is not equal to any w that has been involved in a query before, nor is it equal to IV . We describe an event **Bad**, the event that this assumption fails. We then prove (1) that if **Bad** never happens, the simulator will simulate Γ perfectly, and (2) that **Bad** only happens with negligible probability over the course of an attack.

4.1 Record Keeping

In order to simulate Γ (the weak ideal compression functions) with access only to Δ , we use the natural approach: we answer queries to Γ 's oracles randomly as long as it follows the constraints: (1) for each (x, y) pair, there is only one value z such that $f_0(x, y) = z$, and only one value z' such that $f_1(x, y) = z'$, and (2) for any l m -bit values x_1, \dots, x_l , f_0 and f_1 have to be such that $\Delta(x_1 || \dots || x_l) = C(\Gamma)$.

Meeting the first constraint is easy; we simply do the following on each query. When we receive a query $f(x, y)$ ⁵, we check to see if we have defined an answer $z = f(x, y)$; if so, we return z , and if not, we generate a random z and note that $z = f(x, y)$, and return z . When we receive an attack query $f^{-1}(x, z)$, we pick a random y until we find one such that we have not defined an answer $z' = f(x, y)$ for $z \neq z'$, and return that y , and note that $z = f(x, y)$; we do similarly for an attack query on $f^*(y, z)$.

However, the most difficult part of record keeping is that we must be aware of when a query imposes a constraint based on Δ . In order to do this, we will attempt to keep track of all “partial chains.” A partial chain is a sequence of x values x_1, \dots, x_l , and two y -values y, y' such that $f_1(x_1, f_1(x_2, \dots, f_1(x_l, f_0(x_l, \dots, f_0(x_1, y') \dots))) \dots)) = y$. If a partial chain is such that $y' = IV$ then y must be equal to $\Delta(x_1 || \dots || x_l)$. However, it may be computationally infeasible to keep track of *all* partial chains that arise. Instead, we will keep track of only those that arise in expected ways, and we will prove later that we will actually find all partial chains as long as no unexpected coincidences occur.

⁵ In this proof, when we refer to an f query, we mean either an f_0 or f_1 query. We use this convention similarly when referring to f^* or f^{-1} queries.

For ease of notation, when we discover a partial chain, we will make a note of it, which we denote $\text{Chain}(x, y', y)$. Effectively, this note means that if the initialization vector were y' , then $H(x)$ would output y .

Forward queries. We show how to keep track of this for one type of query at a time, starting with forward queries. Without loss of generality, we assume that the query is on a new input pair (x, y) . If the query is an f_0 query, we will not attempt to find whether any partial chains have been formed. For f_1 queries, we will check if any partial chains have been formed using this query at the end. If so, we check if any of these partial chains are formed starting at IV , and if so, we use Δ to find the value we should set to be $f_1(x, y)$. If not, we pick $f_1(x, y)$ at random. If a query forms two or more distinct partial chains starting at IV , the simulator gives up and halts. If the simulator doesn't halt, it will make notes of all partial chains that have been formed with the current query at the end.

If the query is $f_0(x, y)$ then we can check if this completes a single-block partial chain. If there is a y' such that $f_0(x, y') = y$ then the value we return will form the chain $\text{Chain}(x, y', f_1(x, y))$. If there is an x' and a y'' such that $\text{Chain}(x', y'', y)$ and also there is a y' such that $f_0(x, y') = y''$ then the value we return will form the chain $\text{Chain}(x||x', y', f_1(x, y))$.

Backward queries. Next, we consider “backward” queries, that is, a query $f^{-1}(x, z)$. Similarly to forward queries, if the query is an f_1^{-1} query, we will not attempt to find whether any partial chains have been formed. For f_0^{-1} queries, however, we will check if any partial chains have been formed using this query at the beginning. If so, it may be that a partial chain has been formed starting at IV , but we can do nothing to set the appropriate value to one matching Δ in this case: it is too late, and the simulator will halt. However, this will not happen unless an unexpected coincidence occurs. Thus, once we have found all partial chains that will be formed from the current query, we pick a random answer to it and note the chains that are formed.

The result of a query $f_0^{-1}(x, z)$ will form a single-block chain if it is already known that $f_1(x, z) = y$ for some value y . In this case, we may note $\text{Chain}(x, f_0^{-1}(x, z), y)$. The result of $f_0^{-1}(x, z)$ will form a longer chain if it is already noted that $\text{Chain}(x', z, y')$ for some y' , and also $f_1(x, y') = y$ is known for some y , in which case we may note $\text{Chain}(x||x', f_0^{-1}(x, z), y)$.

Squeeze queries. Finally, we consider “squeeze” queries, that is, a query $f^*(y, z)$. Though squeeze queries may form chains, we do not check for them. If a chain is accidentally formed through a squeeze query, the simulator's behavior may become bad later, but this only happens if an unexpected coincidence occurs.

4.2 The Bad Event

We will prove that our simulator fools the adversary by proving that the distribution of the adversary's output in the real system (where S is not involved) is

identical to the distribution of the adversary’s output in the ideal system, conditioned on a certain “bad” event not happening. The bad event *Bad* represents the event that a previously-used value is generated as the random answer to a later query. To be precise, let us imagine that (x_i, y_i, z_i) are all the triples of values such that $f_0(x_i, y_i) = z_i$ has been established in a query, and that (x'_i, y'_i, z'_i) are all the triples of values such that $f_1(x'_i, y'_i) = z'_i$ has been previously established. Then *Bad* occurs on the next query if:

1. The latest query is an $f(x, y)$ query that returns a value z equal to y_i, z_i, y'_i or z'_i for some i , or $z = IV$.
2. The latest query is an $f^{-1}(x, z)$ query that returns a value y equal to y_i, z_i, y'_i , or z'_i for some i , or $y = IV$.
3. The latest query is an $f^*(y, z)$ query that returns some value x equal to x_i or x'_i for some i .

Lemma 1. *If Bad does not happen when we simulate, the simulator will not halt during a query.*

Recall that the simulator will only halt in one situation: if a forward f_1 query completes more than one partial chain that start at *IV*. Specifically, this happens when a forward query $f_1(x, y)$ is such that for some $x' \neq x''$ and for some y'_0 and y'_1 , we know $\text{Chain}(x', y'_0, y)$ and $\text{Chain}(x'', y'_1, y)$, and $f_0(x, IV) = y'_0$ and $f_0(x, IV) = y'_1$. Therefore we can conclude that $y'_0 = y'_1$. In order for this to happen, we must have noted both $\text{Chain}(x', y', y)$ and $\text{Chain}(x'', y', y)$ for some $x' \neq x''$.

Remark 1. If we note $\text{Chain}(x, y', y)$ then, when we note it, either y' or y is a newly-generated random query answer. This is clear from our description of S above.

Remark 2. First, we prove that if there is some pair of notes $\text{Chain}(x_0, y', y)$ and $\text{Chain}(x_1, y', y)$ where the first block of x_0 is not the same as the first block of x_1 , then *Bad* must have happened. Assume, without loss of generality, that $\text{Chain}(x_0, y', y)$ was not noted later than $\text{Chain}(x_1, y', y)$. Because of the way we notice chains, we note $\text{Chain}(x_1, y', y)$ only when computing either a forward or backward query with x as the input value, where x is the first block of x_1 . Since x is not the first block of x_0 , we do not note $\text{Chain}(x_0, y', y)$ at this time, so it must have been noted previously. However, because of remark 1, when we note $\text{Chain}(x_1, y', y)$ either y' or y must be a newly-generated random query answer, so it can only be equal to the previously-known value of y if *Bad* occurs on this query.

Remark 3. Next, we note that if x_0 and x_1 consist of at least one block, and there is some y such that $\text{Chain}(x||x_0, y', y)$ and $\text{Chain}(x||x_1, y', y)$, where x is a single block, then either (1) there is some w and some w' such that $\text{Chain}(x_0, w', w)$ and $\text{Chain}(x_1, w', w)$ are already known, or (2) *Bad* has happened.

Assuming that both $\text{Chain}(x||x_0, y', y)$ and $\text{Chain}(x||x_1, y', y)$ were discovered simultaneously (if not, the previous argument shows that **Bad** happened), there are two cases:

- If both were discovered on a forward query $f_1(x, w)$, it must have been that both $\text{Chain}(x_0, w', w)$ was known, and that $w' = f_0(x, y')$ for some w and w' . Furthermore, it must also be true that $\text{Chain}(x_1, w'', w)$ was known, and that $w'' = f_0(x, y')$. But then, $w'' = f_0(x, y') = w'$, so the first condition holds.
- If both were discovered on a backward query $f_0^{-1}(x, w')$, then it must have been that $\text{Chain}(x_0, w', w)$ was known for some w , and that $f_1(x, w) = y$. We must also have noted $\text{Chain}(x_1, w', w'')$ for some w'' such that $f_1(x, w'') = y$. If $w = w''$ then the first condition holds. If not, then from remark 1, whichever of $f_1(x, w) = y$, $f_1(x, w'') = y$, $\text{Chain}(x_0, w', w)$, or $\text{Chain}(x_1, w', w'')$ was discovered last would have triggered **Bad**.

Remark 4. If there is some note $\text{Chain}(x, y', y)$ and $\text{Chain}(x||x_1, y', y)$ where x is a single block, then **Bad** has happened. Again, we may assume that $\text{Chain}(x, y', y)$ and $\text{Chain}(x||x_1, y', y)$ were discovered simultaneously. There are two cases:

- If both were discovered on a forward query $f_1(x, y'')$, then it must have been known in advance be that $f_0(x, y') = y''$, and that $\text{Chain}(x_1, y'', y'')$. However, $\text{Chain}(x_1, y'', y'')$ is impossible unless **Bad** happens, in view of remark 1.
- Similarly, if both were discovered on a backward query $f_0^{-1}(x, z)$, then it must have been known in advance that $f_1(x, z) = y$ and that $\text{Chain}(x_1, z, z)$, which again guarantees that **Bad** has happened.

By remarks 2, 3, and 4, if $\text{Chain}(x, y', y)$ and $\text{Chain}(x', y', y)$ are known for $x \neq x'$ then **Bad** must have happened: if x is not a prefix of x' or vice versa, we can descend by remark 2, getting similar properties, until the first blocks of x and x' are unequal. If x is a prefix of x' or vice versa, we can descend by remark 2 until we fall in to the case covered by remark 3. Therefore, the simulator will never halt prematurely unless **Bad** has happened.

Lemma 2. *If a query is ever made to S that would complete a partial chain, we note it unless **Bad** happens.*

Suppose a query is made to S that would complete a partial chain. There are three cases to consider:

Case i: A partial chain is completed on a forward query. If the link determined by $f(x, y)$ is used anywhere other than at the end, it can only be used there if the value generated for $f(x, y)$ triggers the **Bad** event. If the link determined by $f(x, y)$ only completes chains by adding on to the end, it must be a query to f_1 , and then there are two cases: either the partial chain is one block long, which we explicitly check for, or the partial chain is longer, in which case, a shorter, compatible partial chain is already known. In either case, we note the newly completed partial chain.

Case ii: A partial chain is completed on a backward query $f^{-1}(x, z)$. Similarly, if the result of this query is used anywhere other than at the beginning, it can only be used there if the result triggers the **Bad** event. Again, if the result can be used at the beginning, it must be a f_0^{-1} query, and our algorithm for the simulator is correct.

Case iii: A partial chain is completed on a “squeeze” query $f^*(y, z)$. In this case, the chain could only be completed if something is already known about f_0 or f_1 on input x where x is the result of this query. If this were the case, the result of this query would trigger the **Bad** event.

Lemma 3. *If a query is ever made to S that would complete a partial chain starting at IV , we note it, and respond correctly, unless **Bad** happens.*

The proof of this lemma is very similar to the proof of lemma 3. Note that by lemma 3, if a query is made to S that completes *any* partial chain, and **Bad** has not happened, we note it. Therefore, we need only consider two cases:

Case i: The partial chain $\text{Chain}(x, IV, y)$ is noted on a forward query to f_1 . In this case, we obtain y by querying $\Delta(x)$, so our answer is correct.

Case ii: The partial chain $\text{Chain}(x, IV, y)$ is noted on a backward query to f_0^{-1} . If this is the case, **Bad** must have happened, because this can only happen if the result of the final f_0^{-1} query was IV .

Lemma 4. *The probability that **Bad** happens is negligible.*

Note that initially, before any queries are made, **Bad** has not happened. If **Bad** has not happened after the first q queries, then the probability that it happens on the $q + 1$ st query is at most $(2q + 1) \cdot \max(2^{-m}, 2^{-k})$. This is because there are at most $(2q + 1)$ answers (all the previous y and z values, plus IV) that would make **Bad** happen, out of 2^m or 2^k possible random answers, depending on the type of query. Therefore, if the adversary makes a total of q queries, the probability that **Bad** happens is at most $\Omega(q^2 2^{-r})$, where $r = \min(m, k)$.

We note that the running time of S is polynomial in the number of queries, but is independent from the running time of the adversary. This completes the proof of Theorem 1.

4.3 Security Against Standard Attacks

In this section we discuss the applicability of our security proof to the standard attacks against hash functions. What we have proven, essentially, is that an adversary with a limited number of queries cannot distinguish between the zipper hash implemented with weak ideal compression function and a random oracle. Specifically, if the number of queries the adversary can make is significantly less than $2^{\min(m, k)/2}$, the **Bad** event remains extremely unlikely, and the proof is successful.

Provided the adversary makes fewer than this many queries, the only attacks an adversary could succeed in are attacks that could be performed against an ideal hash function. Hence, so long as this query limit is respected, the adversary

should not be able to find collisions, preimages, second preimages, et cetera. However, our proof does not imply that the adversary cannot perform these attacks more efficiently on the zipper hash than on an ideal hash function if the adversary exceeds this query limit. For instance, to find a preimage of an ideal hash function takes $O(2^k)$ queries, where k is the output size, whereas we cannot guarantee security against that many queries. As another example, our construction does not provide security against multicollisions: in fact, it fits a known framework in which an extension of the Joux attack is possible [15,9].

Nonetheless, keep in mind that the queries the adversary is allowed to make in attacking the zipper hash include attack queries, which are modeled as if they are trivial, but may in fact require significant effort.

5 Zipper Hash-Based Compression Function

The most natural criticism of the zipper hash in practice is that it is no longer streamable, as iterated hash functions are. However, we can easily use the zipper hash construction to create an ideal compression function rather than a full ideal hash function, which will allow us to use one of the modified iterated constructions of Coron et al. [4] and create a streamable, ideal hash function from weak ideal compression functions.

Now that we have proven that the zipper hash is indistinguishable from a random oracle, if we assume that we have an (m, m) -bit underlying compression function, we can make an (m, m) -bit ideal compression function very simply: let $f(x, y) = H(x||y)$. That is, we use a full zipper hash computation on the two-block message $x||y$ as our compression function. By a simple restriction on our theorem, this is indistinguishable from a random oracle from $\{0, 1\}^m \times \{0, 1\}^m \rightarrow \{0, 1\}^m$, and is therefore an ideal compression function.

5.1 Amortizing Streamability vs. Efficiency

The full zipper hash requires 2 underlying compression function queries per input block. If we use the zipper hash in the natural way as a compression function, the resulting iterated construction (after double-piping) requires 4 underlying compression function queries per input block. However, we can trade streamability for efficiency here, by using the zipper hash function on more blocks of input at once.

For instance, we can make a $(3m, m)$ -bit compression function by computing $f(x_1||x_2||x_3, y) = H(x_1||x_2||x_3||y)$. This requires 8 queries for 3 input blocks, which is a significant savings compared to 12 queries for 3 input blocks. However, by having the compression function require more input, we are sacrificing streamability: we must now buffer 3 input blocks instead of one before we can apply the compression function. In general, if we use b blocks at once, our efficiency will be $\frac{2(b+1)}{b} = 2 + \frac{2}{b}$ queries per input block.

6 Efficiency

The zipper hash requires $2l$ compression function evaluations on an input of l blocks. There is one additional drawback in that the zipper hash is not *streamable*: we have to scan the message twice, so in principle, we cannot compute the zipper hash in fixed memory unless for some reason it is feasible to access the input a second time. This is an especially significant point as it is often desired that limited devices such as smart cards be able to compute hash functions with limited available memory. However, there are some points in favor of this approach anyway:

- In applications on non-limited devices, streamability is not mission-critical. It may be worthwhile to consider a non-streamable hash function like the zipper hash if it has attractive theoretical properties.
- The zipper hash can be implemented using existing machinery: essentially all that is required is two traditional Merkle-Damgård hash function evaluations.
- Theoretically, it is possible that weakly secure compression functions could be designed that may be more efficient than strong ones. If this is the case, then such compression functions used in the zipper hash constructions may actually yield a *more* efficient hash function.
- The zipper hash may remain secure even if the compression function is vulnerable to attack.
- Finally, *any* streamable hash function is essentially an iterated hash function based on a compression function. Therefore, some black-box attacks are known that apply to any streamable hash function. If such attacks are undesirable, it may be necessary to adopt a non-streamable approach.

This last point needs some explanation. We prove that all streamable hash functions (that is, all hash functions with constant-size state) are in fact iterated hash functions in Appendix B. The gist of the argument is that if a hash function can be streamed, this means it can be computed with a fixed amount of state, with a fixed maximum amount of memory into which the input is provided. Whatever method is used to combine the input with the current state to arrive at the next state can be thought of as the compression function. The initial state can be thought of as the initialization vector, and whatever method is used, once all input has been processed, to determine the output can be thought of as the finalization function.

7 Conclusion

This paper is new in two ways. First of all, this is the first paper that we are aware of to foray into positive results for non-streamable hash function design. Second, as far as we know, this is the first paper to explicitly model weakly-secure

primitives as ideal primitives with relevant attack oracles available. Here are some open problems we consider worth investigating:

- Are there attacks against the generic zipper hash design that are better than brute force?
- What other non-streamable hash function designs are possible, and what properties do they have? In particular, are there benefits to making three or more passes in a zipper-like construction?
- Is there a weaker version of an ideal compression function? If so, can we use it to build secure hash functions?
- Can this notion of a weak ideal primitive be used elsewhere?
- Can we make better constructions, or prove stronger security results, by representing our compression functions as ideal random quasigroups?

Acknowledgements

We would like to sincerely thank Bart Preneel [17] and Stefan Lucks [13] for their Asiacrypt 2005 presentations which inspired this research. We would also like to thank those with whom we had useful conversations concerning this project: Ron Rivest, Stefan Lucks, Stuart Haber, Zufikar Ramzan, Silvio Micali, Susan Hohenberger, and Barbara Liskov. We would also like to thank the program committee for their comments.

References

1. Biham, E., Chen, R., Joux, A., Carribault, P., Lemuet, C., Jalby, W.: Collisions of SHA-0 and reduced SHA-1. In: Cramer [5], pp. 36–57
2. Black, J., Rogaway, P., Shrimpton, T.: Black-box analysis of the block-cipher-based hash-function constructions from PGV. In: Yung, M. (ed.) CRYPTO 2002. LNCS, vol. 2442, pp. 320–335. Springer, Heidelberg (2002)
3. Brassard, G. (ed.): CRYPTO 1989. LNCS, vol. 435. Springer, Heidelberg (1990)
4. Coron, J., Dodis, Y., Malinaud, C., Punyia, P.: Merkle-Damgård revisited: how to construct a hash function. In: Shoup [18], pp. 430–448
5. Cramer, R. (ed.): EUROCRYPT 2005. LNCS, vol. 3494, pp. 22–26. Springer, Heidelberg (2005)
6. Damgård, I.: A design principle for hash functions. In: Brassard [3], pp. 416–427
7. Dean, R.: Formal aspects of mobile code security. Ph.D. Dissertation, Princeton University (1999)
8. Franklin, M. (ed.): CRYPTO 2004. LNCS, vol. 3152. Springer, Heidelberg (2004)
9. Hoch, J., Shamir, A.: Breaking the ICE - finding multicollisions in iterated concatenated and expanded (ICE) hash functions. In: Robshaw, M. (ed.) FSE 2006. LNCS, vol. 4047, Springer, Heidelberg (2006)
10. Joux, A.: Multicollisions in iterated hash functions, application to cascaded constructions. In: Franklin [8], pp. 306–316
11. Kelsey, J., Kohno, T.: Herding hash functions and the Nostradamus attack. Available on eprint: article 2005/281 (2005)

12. Kelsey, J., Schneier, B.: Second preimages on n -bit hash functions for much less than 2^n work. In: Cramer [5], pp. 474–490
13. Lucks, S.: A failure-friendly design principle for hash functions. In: Roy, B. (ed.) ASIACRYPT 2005. LNCS, vol. 3788, pp. 474–494. Springer, Heidelberg (2005)
14. Merkle, R.C.: A certified digital signature. In: Brassard [3], pp. 218–238
15. Nandi, M., Stinson, D.R.: Multicollision attacks on a class of hash functions. Available on IACR eprint archive, paper 2006-2055 (2006)
16. Preneel, B.: Analysis and design of cryptographic hash functions. Ph. D. thesis, updated version (2003)
17. Preneel, B.: Hash functions: past, present and future. In: Roy, B. (ed.) ASIACRYPT 2005. LNCS, vol. 3788, Springer, Heidelberg (2005)
18. Shoup, V. (ed.): CRYPTO 2005. LNCS, vol. 3621. Springer, Heidelberg (2005)
19. Wang, X., Lai, X., Feng, D., Chen, H., Yu, X.: Cryptanalysis of the hash functions MD4 and RIPEMD. In: Cramer [5], pp. 1–18
20. Wang, X., Yin, Y.L., Yu, H.: Finding collisions in the full SHA-1. In: Shoup [18], pp. 17–36
21. Wang, X., Yu, H.: How to break MD5 and other hash functions. In: Cramer [5], pp. 19–35
22. Wang, X., Yu, H., Yin, Y.L.: Efficient collision search attacks on SHA-0. In: Shoup [18], pp. 1–16

A Simulating Two Compression Functions

In the zipper hash construction, we require two independent compression functions. However, in practice, it is more sensible to use only one. If we have a single $(m + 1, k)$ -bit compression function f , we can define $f_0(x, y) = f(0||x, y)$ and $f_1(x, y) = f(1||x, y)$. This idea is natural, but we must prove that it is secure in an ideal sense.

Theorem 2. *If C is the TM that computes f_0 and f_1 given the oracle Γ for a weak ideal compression function f , then C is (perfectly) indifferentiable from Δ , where Δ is the oracle for two independent random weak ideal compression functions.*

Proof. First of all, we must describe how C answers attack queries. If C is given an attack query $f_b^{-1}(x, z)$, C makes an attack query $f^{-1}(b||x, z)$ to obtain the answer y . If, on the other hand, C is given an attack query $f_b^{-1}(y, z)$, C makes an attack query $f^{-1}(y, z)$ to obtain an answer $b' || x$ where b' is the first bit. If $b' = b$, C returns x ; if not, C makes an identical attack query, and repeats until it obtains an answer that does begin with b .

To simulate the oracle for the single weak ideal compression function is very easy. To answer a forward query $f(b||x, y)$, we simply return the result of the forward query $f_b(x, y)$. To answer an attack query $f^{-1}(b||x, z)$, we return the result of the attack query $f_b^{-1}(x, z)$. Finally, to answer an attack query $f^{-1}(y, z)$, we pick a random bit b and return the result of $f_b^{-1}(y, z)$.

The only effective difference between the simulation and the construction is that the construction may have made superfluous queries to $f^{-1}(y, z)$. However,

for all adversaries, the queries of the adversary are independent of the superfluous queries, and even if the adversary makes a query that is later restricted by one of those superfluous queries, the result is still random and distributed properly.

Another application of this is to show how to make the Lucks double-pipe construction ideally secure.

Lucks' construction of a double-pipe compression function can be summed up as follows: if f is a $(2m, n)$ -bit compression function, then

$f'(x, y_1 || y_2) = f(x, y_1 || y_2) || f(x, y_2 || y_1)$ is a $(2m, 2n)$ -bit compression function.

It is easy to see that this construction is not ideal: $f'(x, y || y) = z || z$ for some z , whereas this is unlikely to be the case for an ideal compression function f' . This flaw is easily avoided, however, by the following modification. Assume that f_0 and f_1 are two *independent* $(2m, n)$ -bit compression functions, and let

$$f'(x, y_1 || y_2) = f_0(x, y_1 || y_2) || f_1(x, y_2 || y_1).$$

Theorem 3. *f' is computationally indifferentiable from an ideal compression function.*

Proof. It is easy to see how to simulate the f_0 and f_1 random oracles in the presence of a single random oracle for f' : to calculate $f_0(x, y)$, for instance, we query $f'(x, y)$, and split the result into two halves, the first of which is $f_0(x, y)$, and the second of which is recorded as $f_1(x, y^{fl})$ where y^{fl} flips the first and second halves of y .

Naturally, we can apply Theorem 2 to show that f_0 and f_1 can be simulated with a single compression function. We do not even need the full strength of Theorem 2, since in this case the compression functions are meant to be ideal, rather than weak ideal.

B Universality of Merkle-Damgård

In this section, we prove that any streamable hash function can be viewed as an iterated hash function with a single compression function f , a fixed initialization vector IV , and a finalization function g .

In order to consider a hash function “streamable,” it must be the case that H can be computed by an algorithm M in such a way that (1) M has a fixed-size state, and (2) M receives its input in pieces, each piece being no larger than some maximum size m . If (1) does not hold, then potentially, M simply stores the entire input and computes the hash function only at the end. Furthermore, we assume that the function operates directly on the input message, that is, we assume that any padding is computed as part of the hash function. If this is the case, we can view H as an iterated hash function as follows.

Let k be such that any state of M can be written as a k -bit string. Let IV be the k -bit string corresponding to the starting state of M , appended with the empty string ϵ .

Let f be the $(m, k + m + 1)$ -bit compression function that works as follows. On input (x, y) , let the state s be the state corresponding to the first k bits of y , and let x' be the string encoded in the remaining $m + 1$ bits. Based on s , we determine the amount $m' \leq m$ of input that M expects in state s . Let s' be the state after M is run in state s with input the first m' bits of $x' || x$. If $|x' || x|$ is large enough that it contains the entire next block to process, we repeat this, processing another chunk of the input. If not, we let s_{out} be the state M ends in, and let x_{out} be whatever part of $x' || x$ has not been used, and output the $k + m + 1$ -bit string representation $\langle s_{out}, x_{out} \rangle$.

Let g be as follows. On input x , let s be the state corresponding to the first k bits of x , and let x' be the string encoded in the remaining bits. Run M , from state s , on input x' with the signal that no more input remains, and output the output of M .

It should be clear that the iterated hash function with compression function f , finalization function g , and initialization vector IV simply runs M on the input to the hash function. This shows that the streamable hash function H is in fact an iterated hash function.

B.1 Universal Vulnerability of Iterated Hash Functions

Since all hash functions that can reasonably be thought of as streamable are in fact iterated hash functions, black-box attacks against iterated hash functions are actually universal attacks against streamable hash functions. For instance, the second collision attack described in the introduction applies to *all* streamable hash functions. Also, the Joux attack applies to all streamable hash functions. The efficiency of the Joux attack, as well as its extensions, depends on the size of the internal state ($k+m+1$, with the construction above); if this size is large enough, the Joux attack is irrelevant, as Lucks has shown. However, note that a streamable hash function may be realizable as an iterated hash function more efficiently in terms of internal state size than the general construction above.