Constructing Secure Hash Functions from Weak Compression Functions: The Case for Non-Streamable Hash Functions

Moses Liskov

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

Abstract. In a recent paper, Lucks espoused a "failure-friendly" approach to hash function design [12]. We expand on this idea in two main ways. First of all, we consider 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 many unusual properties of compression functions.
Furthermore, 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 ideal building blocks.
The zipper hash function is relatively efficient, requiring two compression function evaluations per block of input, but it is not streamable. We also show how to create an ideal compression function from ideal weak compression functions, which can be used in the standard iterated way to make a streamable hash function. However, a comparison of these two constructions, as well as consideration of certain recent attacks against iterated hash functions, lead us to the conclusion that non-streamable hash functions may be worth considering.

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 [20, 18, 19, 17, 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, 13]: 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

$$H(x) = f(f(\dots f(IV, x_1), x_2), \dots, x_l))$$

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 so that distinct messages remain distinct after 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.

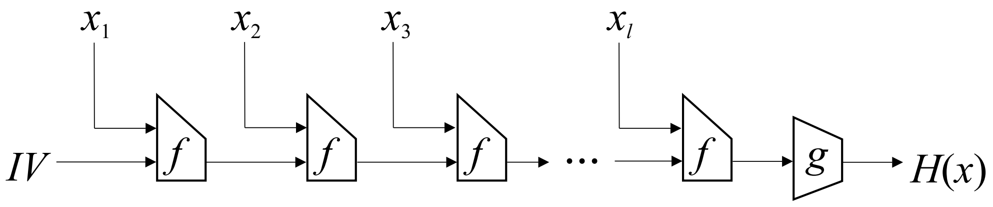

Figure 1: The modern iterated hash function

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, 13]. 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 $|x| = |y|$. [14, 12]
- **Joux multicollision attack** [9]. 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** [11, 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 suscpetible to fixed-point attacks[11]. This allows an attack where, after hashing 2^t mesage *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** [10]. This is an attack against the use of a hash function for commitments. The idea is to find a 2^t-way collision (using the Joux attack) 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 [12]. Lucks proposed *double-pipe hash* as a way to implement this, effectively using two compression function computations per block of message, in order to increase the size of the internal state securely. 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 of all, 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.

Second, following the work of Coron, Dodis, Malimaud, and Puniya [4], we wish to find ways of strengthening hash functions to a high standard. Coron et al show that the basic Merkle-Damgård construction is not ideal in the sense that even with an ideal compression function, we cannot prove 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 indistiguishable 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], and since we will use this model, we should attain strong results, as Coron et al do.

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 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 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 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 implmented 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)).

[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').

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 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.

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. 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.

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 $\mathit{\Gamma}$ represent the oracle(s) for the underlying ideal primitive(s), and let $\mathit{\Delta}$ 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 $\mathit{\Delta}$ if there is a simulator S such that for all distinguishers A,

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

where (1) S answers as many different types of oracle queries as $\mathit{\Gamma}$ provides, and S has oracle access to $\mathit{\Delta}$ 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 $\mathit{\Delta}$ if for all security parameters α it holds that C is $(t_A(\alpha), t_S(\alpha), q(\alpha), \epsilon(\alpha))$-indifferentiable from $\mathit{\Delta}$, where t_A and t_S are polynomial in α, where $q(\alpha) \leq t_A(\alpha)$, and where ϵ is negligible in α.

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 $x \circ P(x)$ is a string that can be broken down into m-bit blocks, and for all $x \neq x'$, $x \circ P(x) \neq x' \circ P(x')$. Given all these pieces, the zipper hash function works as follows:

1. Let $x_1, \ldots x_l$ be m-bit strings such that $x_1 \circ \ldots \circ x_l = x \circ P(x)$.
2. H_1 is computed as $f_0(x_1, IV)$, and $H_2, \ldots, H_l$ 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 $H'_2, \ldots, H'_l$ 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 reminscent of a zipper. See figure 2.

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

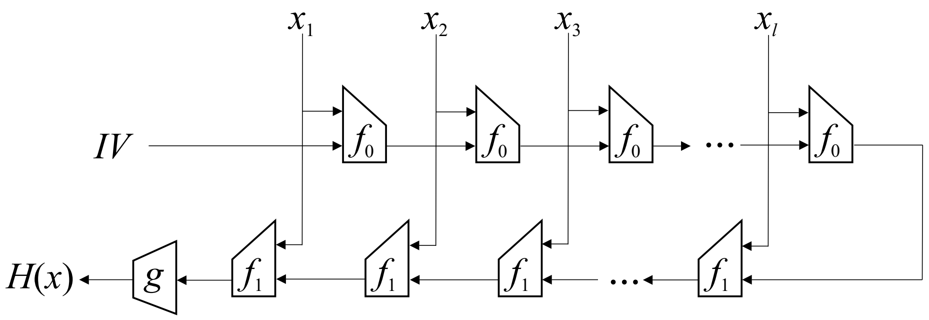

Figure 2: The zipper hash function

4 Proof of security

Let C be the Turing machine that implements the zipper hash. We will prove that C is computationally indifferentiable from an ideal hash function $\varDelta$, using two ideal weak compression functions represented by $\varGamma$, where g is the identity function.[3]

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 $\varDelta$ 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 $\varGamma$, 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 $\varGamma$ perfectly, and (2) that Bad only happens with negligible probability over the course of an attack.

4.1 Record keeping

In order to simulate $\varGamma$ (the weak ideal compression functions) with access only to $\varDelta$, we use the natural approach: we answer queries to $\varGamma$'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 $x_1, \ldots, x_l$, f_0 and f_1 have to be such that $\varDelta(x_1 \circ \ldots \circ x_l) = C(\varGamma)$.

Meeting the first constraint is easy; we simply do the following on each query. When we receive a query $f(x, y)$[4], 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 $f^*(y, z)$.

[3] This will essentially prove that querying the zipper hash on a new message is indistinguishable from querying g on a randomly-chosen value. If g has the property that it produces a random output on a random input, the overall construction will remain ideal.

[4] 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.

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 $x_1, \ldots, x_l$, and two y-values y, y' such that $f_0(x_1, f_0(x_2, \ldots, f_0(x_l, f_1(x_l, \ldots, f_1(x_1, y') \ldots)) \ldots)) = y$. If a partial chain is such that $y' = IV$ then y must be equal to $\Delta(x_1 \circ \ldots \circ x_l)$. 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.

For ease of notation, when we discover a partial chain, we will make a note of it, which we denote $\mathsf{Chain}(x, y', y)$. 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 $f_1(x, y)$ 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 $\mathsf{Chain}(x, y', f_1(x, y))$. If there is an x' and a y'' such that $\mathsf{Chain}(x', y'', y)$ and also there is a y' such that $f_0(x, y') = y''$ then the value we return will form the chain $\mathsf{Chain}(x \circ x', y', f_1(x, y))$.

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

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.

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 $\mathsf{Chain}(x', y'_0, y)$ and $\mathsf{Chain}(x'', y'_1, y)$, 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 $\mathsf{Chain}(x', y', y)$ and $\mathsf{Chain}(x'', y', y)$ for some $x' \neq x''$.

Remark 1. If we note $\mathsf{Chain}(x, y', y)$ 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 $\mathsf{Chain}(x_0, y', y)$ and $\mathsf{Chain}(x_1, y', y)$ 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 $\mathsf{Chain}(x_0, y', y)$ was not noted later than $\mathsf{Chain}(x_1, y', y)$. Because of the way we notice chains, we note $\mathsf{Chain}(x_1, y', y)$ 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 $\mathsf{Chain}(x_0, y', y)$ at this time, so it must have been noted previously. However, because of remark 1, when we note $\mathsf{Chain}(x_1, y', y)$ 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 $\mathsf{Chain}(x \circ x_0, y', y)$ and $\mathsf{Chain}(x \circ x_1, y', y)$, where x is a single block, then either (1) there is some w and some w' such that $\mathsf{Chain}(x_0, w', w)$ and $\mathsf{Chain}(x_1, w', w)$ are already known, or (2) Bad has happened.

Assuming that both $\mathsf{Chain}(x \circ x_0, y', y)$ and $\mathsf{Chain}(x \circ x_1, y', y)$ 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 $\mathsf{Chain}(x_0, w', w)$ was known, and that $w' = f_0(x, y')$ for some w and w'. Furthermore, it must also be true that $\mathsf{Chain}(x_1, w'', w)$ 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 $\mathsf{Chain}(x_0, w', w)$ was known for some w, and that $f_1(x, w) = y$. We must also have noted $\mathsf{Chain}(x_1, w', w'')$ 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 $f_1(x, w) = y, f_1(x, w'') = y, \mathsf{Chain}(x_0, w', w)$, or $\mathsf{Chain}(x_1, w', w'')$ was discovered last would have triggered Bad.

Remark 3. If there is some note $\mathsf{Chain}(x, y', y)$ and $\mathsf{Chain}(x \circ x_1, y', y)$ where x is a single block, then Bad has happened. Again, we may assume that $\mathsf{Chain}(x, y', y)$ and $\mathsf{Chain}(x \circ x_1, y', y)$ 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 $\mathsf{Chain}(x_1, y'', y'')$. However, $\mathsf{Chain}(x_1, y'', y'')$ 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 $\mathsf{Chain}(x_1, z, z)$, which again guarantees that Bad has happened.

By remarks 2, 3, and 4, if $\mathsf{Chain}(x, y', y)$ and $\mathsf{Chain}(x', y', y)$ are known for $x \neq x'$ then Bad must have happened: if x is not a prefix of x' 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 3. *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 4. *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 $\mathsf{Chain}(x, IV, y)$ 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 $\mathsf{Chain}(x, IV, y)$ 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 5. *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 $r = min(m,k)$.

This completes the proof that the zipper hash function is indistinguishable from a random oracle when it is made using two weak ideal compression functions.

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 indifferentiable 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 $f(x, y) = H(x \circ y)$. By a simple restriction on our theorem, this is indifferentiable from a random oracle from $\{0,1\}^m \times \{0,1\}^m \to \{0,1\}^m$, and is therefore an ideal compression function.

In order to make the iterated hash function from this compression function comparable to the zipper hash, however, we would have to follow the advice of Lucks [12] and somehow use a wide-pipe hash to avoid the Joux attack and related concerns that arise in an iterated construction. This is because the non-streamable zipper hash is not known to be vulnerable to these general black-box attacks.

Lucks' construction of a double-pipe compression function can be summed up as follows: if f is a $(2m, n)$-bit compression function, then $f'(x, y_1 \circ y_2) = f(x, y_1 \circ y_2) \circ f(x, y_2 \circ y_1)$ is a $(2m, 2n)$-bit compression function.

It is easy to see that this construction is not ideal: $f'(x, y \circ y) = z \circ z$ 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

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

Then 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.

In order to use this modified double-pipe construction, we just have to use the zipper hash with two different IV values, one for f_0 and one for f_1. The result is a compression function that requires eight weak ideal compression function queries per evaluation. See figure 3.

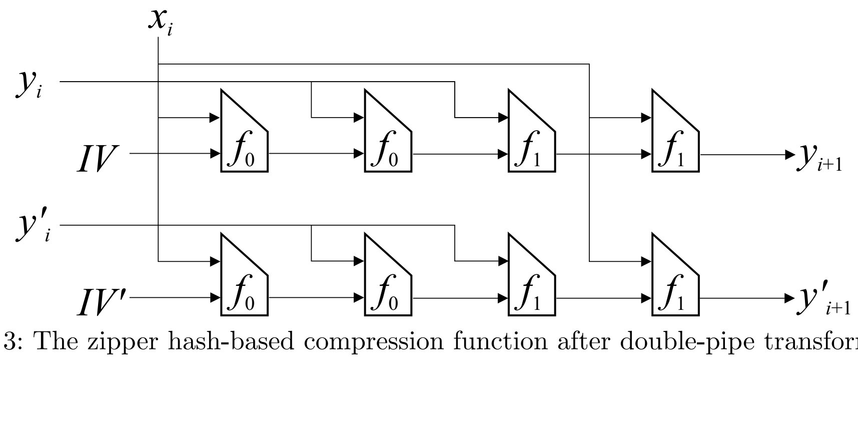

Figure 3: The zipper hash-based compression function after double-pipe transformation.

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 8 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 $f(x_1 \circ x_2 \circ x_3, y) = H(x_1 \circ x_2 \circ x_3 \circ y)$. This requires 8 queries for 3 input blocks (16 queries for 3 input bocks after applying the double-pipe transform), which is a significant savings compared to 24 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.

6 Efficiency

Note that the zipper hash requires $2l$ compression function evaluations on an input of l blocks. This is relatively efficient, especially when we consider the potential benefits to performance we may get by relaxing the security requirements on the compression function itself.

However, there is one 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, especially if it has attractive efficiency properties.
– *Any* streamable hash function is essentially an iterated hash function based on a compression function. The program for the streamable hash function can be thought of as the "key" of the compression function, while the incremental computation done by it can be thought of as the compression function itself, where the internal state (including the memory) is the chaining value, and where the computation of the final value is the finalization function. Thus, streamable hash functions are *inherently* iterated hash functions, and thus vulnerable to certain attacks, including all the attacks mentioned in the introduction. Therefore, if we want to avoid such attacks, we will need to consider non-streamable hash functions.
– The zipper hash can be implemented using existing machinery: essentially all that is required is two traditional Merkle-Damgård hash function evaluations, plus one more compression function evaluation and an XOR in the middle.
– At present, the zipper hash takes several times fewer compression function evaluations than the best known construction using similarly weak compression functions (our zipper hash-based compression construction). We make no claim that the zipper hash-based compression function construction is optimal, but nonetheless, there is a sizable gap at the moment. Note that the "most" streamable version of the zipper hash-based compression construction takes about 4 times as many evaluations. Though we can amortize some of the cost by using a multi-block compression function, for any constant number of blocks to process simultaneously, the streamable version is always at least twice as expensive, thanks to the necessity of the double-pipe transformation.

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 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?
- Is there a weaker version of an ideal compression function? And 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 by representing our compression functions as ideal random quasigroups?

Acknowledgements

We would like to sincerely thank Bart Preneel [15] and Stefan Lucks [12] 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, Zulfikar Ramzan, Silvio Micali, Susan Hohenberger, and Barbara Liskov. We would also like to thank the program committee in advance for their comments.

References

1. E. Biham, R. Chen, A. Joux, P. Carribault, C. Lemuet, and W. Jalby. Collisions of sha-0 and reduced sha-1. In Cramer [5], pages 36–57.
2. J. Black, P. Rogaway, and T. Shrimpton. Black-box analysis of the block-cipher-based hash-function constructions from pgv. In Moti Yung, editor, *Advances in Cryptology—CRYPTO 2002*, Lecture Notes in Computer Science, pages 320–335. Springer-Verlag, 2002.
3. G. Brassard, editor. *Advances in Cryptology—CRYPTO '89*, volume 435 of *Lecture Notes in Computer Science*. Springer-Verlag, 1990, 20–24 August 1989.
4. J. Coron, Y. Dodis, C. Malinaud, and P. Punyia. Merkle-damgård revisited:how to construct a hash function. In Franklin [8], pages 430–448.
5. Ronald Cramer, editor. *Advances in Cryptology—EUROCRYPT 2005*, Lecture Notes in Computer Science. Springer-Verlag, 22 – 26 May 2005.
6. I. Damgård. A design principle for hash functions. In Brassard [3], pages 416–427.
7. R. Dean. Formal aspects of mobile code security. Ph.D. Dissertation, Princeton University, 1999.
8. Matt Franklin, editor. *Advances in Cryptology: CRYPTO 2004*, volume 3152 of *Lecture Notes in Computer Science*. Springer-Verlag, 15–19 aug 2004.
9. A. Joux. Multicollisions in iterated hash functions, application to cascaded constructions. In Franklin [8], pages 306–316.
10. J. Kelsey and T. Kohno. Herding hash functions and the nostradamus attack. Available on eprint: article 2005/281, 2005.
11. J. Kelsey and B. Schneier. Second preimages on n-bit hash functions for much less than 2^n work. In Cramer [5], pages 474–490.
12. S. Lucks. A failure-friendly design principle for hash functions. In Bimal Roy, editor, *Advances in Cryptology—ASIACRYPT 2005*, volume 3788 of *Lecture Notes in Computer Science*, pages 474–494, Chennai, India, 4–8 December 2005. Springer-Verlag.
13. Ralph C. Merkle. A certified digital signature. In Brassard [3], pages 218–238.
14. B. Preneel. Analysis and design of cryptographic hash functions. Ph. D. thesis, updated version, 2003.

15. B. Preneel. Hash functions: past, present and future. Invited lecture, ASIACRYPT 2005, Chennai, India, 2005.
16. Victor Shoup, editor. *Advances in Cryptology: CRYPTO 2005*, volume 3621 of *Lecture Notes in Computer Science*. Springer-Verlag, August 2005.
17. X. Wang, X. Lai, D. Feng, H. Chen, and X. Yu. Cryptanalysis of the hash functions md4 and ripemd. In Cramer [5], pages 1–18.
18. X. Wang, Y. L. Yin, and H. Yu. Finding collisions in the full sha-1. In Shoup [16], pages 17–36.
19. X. Wang and H. Yu. How to break md5 and other hash functions. In Cramer [5], pages 19–35.
20. X. Wang, H. Yu, and Y. L. Yin. Efficient collision search attacks on sha-0. In Shoup [16], pages 1–16.