Second Preimages on n-bit Hash Functions for Much Less than 2^n Work

John Kelsey[1] and Bruce Schneier[2]

[1] National Institute of Standards and Technology, `john.kelsey@nist.gov`
[2] Counterpane Internet Security, Inc., `schneier@counterpane.com`

Abstract. We provide a second preimage attack on all n-bit iterated hash functions with Damgard-Merkle strengthening and n-bit intermediate states, allowing a second preimage to be found for a 2^k-message-block message with about $k \times 2^{n/2+1} + 2^{n-k+1}$ work. Using SHA1 as an example, our attack can find a second preimage for a 2^{60} byte message in 2^{106} work, rather than the previously expected 2^{160} work. We also provide slightly cheaper ways to find multicollisions than the method of Joux[J04]. Both of these results are based on *expandable messages*–patterns for producing messages of varying length, which all collide on the intermediate hash result immediately after processing the message.[3] We also provide algorithms for finding expandable messages for a hash function, using only a small multiple of the work done to find a single collision in the hash function.

1 Introduction

The security goal for an n-bit hash function is that collisions require about $2^{n/2}$ work, while preimages and second preimages require about 2^n work. In this paper, we demonstrate that the standard way of constructing iterated hash functions (the Damgard-Merkle construction) cannot meet this goal: For a message of 2^k message blocks, we provide a second preimage attack requiring about $k \times 2^{n/2+1} + 2^{n-k+1}$ work. For some widely used compression functions, such as the SHA family[SHA02], the attack is even cheaper, requiring only $3 \times 2^{n/2+1} + 2^{n-k+1}$ work. Our attacks are made possible by the notion of *expandable messages*–patterns of messages of different lengths which all yield the same intermediate hash value after processing them. These expandable messages do not directly yield collisions on the whole hash function because of the length padding done at the end of modern hash functions, and in any event, they are no easier to find than collisions. However, they allow second preimages and multicollisions to be found much more cheaply than had previously been expected.

The remainder of the paper is organized as follows: First, we discuss basic hash function constructions and security requirements. Next, we

[3] We have heard of a previous paper, possibly never published, that independently introduced the idea of expandable messages based on fixed points. We would appreciate any pointers to this paper so that we can cite it. Thanks!

demonstrate two ways to find "expandable messages." We then demonstrate how these expandable messages can be used to violate the second preimage resistance of essentially all currently specified cryptographic hash functions with less than 2^n work. Finally, we demonstrate an even more efficient (albeit much less elegant) way to find multicollisions than the method of Joux. We end with a discussion of how this affects our understanding of iterated hash function security.

2 Hash Function Basics

In 1989, Merkle and Damgard [M89,D89] independently invented the basic construction used for essentially all modern cryptographic hash functions. Here, we describe this construction, and its normal security claims.

A hash function with an n-bit output is expected to have three minimal security properties. (In practice, a number of other properties are expected, as well.)

1. Collision-resistance: An attacker should not be able to find a pair of messages $M \neq M'$ such that $\text{hash}(M) = \text{hash}(M')$ with less than about $2^{n/2}$ work.
2. Preimage-resistance: An attacker given a possible output value for the hash Y should not be able to find an input X so that $Y = \text{hash}(X)$ with less than about 2^n work.
3. Second preimage-resistance: An attacker given one message M should not be able to find a second message, M' to satisfy $\text{hash}(M) = \text{hash}(M')$ with less than about 2^n work.

A collision attack on an n-bit hash function with less than $2^{n/2}$ work, or a preimage or second preimage attack with less than 2^n work, is formally a break of the hash function. Whether the break poses a practical threat to systems using the hash function depends on specifics of the attack.

Following the Damgard-Merkle construction, an iterated hash function is built from a fixed-length component called a compression function, which takes an n-bit input chaining value and an m-bit message block, and derives a new n-bit output chaining value. In this paper, $F(H, M)$ is used to represent the application of this compression function on hash chaining variable H and message block M.

In order to hash a full message, the following steps are carried out:

1. The input string is padded to ensure that it is an integer multiple of m bits in length, and that the length of the original, unpadded message appears in the last block of the padded message.
2. The hash chaining value $h[i]$ is started at some fixed IV, $h[-1]$, for the hash function, and updated for each successive message block $M[i]$ as
$$h[i] = F(h[i-1], M[i])$$
3. The value of $h[i]$ after processing the last block of the padded message is the hash output value.

This construction gives a reduction proof: If an attacker can find a collision in the whole hash, then he can likewise find one in the compression function. The inclusion of the length at the end of the message is important for this reduction proof, and is also important for preventing a number of attacks, including long-message attacks [MvOV96].

Besides the claimed security bounds, there are two concepts from this brief discussion that are important for the rest of this paper:

1. A message made up of many blocks, $M[0, 1, 2, ..., 2^k - 1]$, has a corresponding sequence of intermediate hash values, $h[0, 1, 2, ..., 2^k - 1]$.
2. The padding of the final block includes the length, and thus prevents collisions between messages of different lengths in the intermediate hash states from yielding collisions in the full hash function.

3 Finding Expandable Messages

An expandable message is a kind of multicollision, in which the colliding messages have different lengths, and the message hashes collide in the *input* to the last compression function computation, before the length of the message is processed. Consider a starting hash value $h[-1]$. Then an "expandable message" from $h[-1]$ is a pattern for generating messages of different lengths, all of which yield the same intermediate hash value when they are processed by the hash, starting from $h[-1]$, without the final padding block with the message length being included. In the remainder of the paper, an expandable message that can take on any length between a and b message blocks, inclusive, will be called an (a, b)-expandable message.

3.1 A Generic Technique: Multicollisions of Different Lengths

Finding an expandable message for any compression function with n-bit intermediate hash values takes only a little more work than finding a collision in the hash function. This technique is closely related to the technique for finding k-collisions in iterated hash functions from Joux.

In Joux's technique, a sequence of single-message-block collisions is found, and then pasted together to provide a large number of different messages of equal length that lead to the same hash value. In our technique, a sequence of collisions between messages of *different* lengths is found, and pasted together to provide a set of messages that can take on a wide range of different lengths without changing the resulting intermediate hash value—an expandable message.

Finding a Collision on Two Messages of Different Lengths
Finding an expandable message requires the ability to find many pairs of messages of different specified lengths that have the same resulting intermediate hash value. Finding such a pair is not fundamentally different than finding a pair of equal-length messages that collide: The attacker

who wants a collision between a one-block message and an α-block message constructs about $2^{n/2}$ messages of length 1, and about the same number of length α, and looks for a collision. For efficiency, the attacker chooses a set of α-block messages whose hashes can be computed about as efficiently as the same number of single-block messages.

ALGORITHM: FindCollision(α, h_{in})

Finding a collision pair with lengths 1 and α, starting from h_{in}.

Variables:

1. α=desired length of second message.
2. A, B = lists of intermediate hash values.
3. q = a fixed "dummy" message used for getting the desired length.
4. h_{in} = the input hash value for the collision.
5. h_{tmp} = intermediate hash value used in the attack.
6. $M(i)$ = the ith distinct message block used in the attack.
7. n = width of hash function output in bits.

Steps:

1. Compute the starting hash for the α-block message by processing $\alpha - 1$ dummy message blocks:
   - $h_{tmp} = h_{in}$.
   - For $i = 0$ to $\alpha - 2$:
     - $h_{tmp} = F(h_{tmp}, q)$
2. Build lists A and B as follows:
   - for $i = 0$ to $2^{n/2} - 1$:
     - $A[i] = F(h_{in}, M(i))$
     - $B[i] = F(h_{tmp}, M(i))$
3. Find i, j such that $A[i] = B[j]$
4. Return colliding messages $(M(i), q||q||...||q||M(j))$, and the resulting intermediate hash $F(h_{in}, M(i))$.

Work: $\alpha - 1 + 2^{n/2+1}$ compression function calls

Building a Full $(k, k + 2^k - 1)$-expandable message We can use the above algorithm to construct expandable messages that cover a huge range of possible lengths. We first find a colliding pair of messages, where one is of one block, and the other of $2^{k-1} + 1$ blocks. Next, we find a collision pair of length either 1 or $2^{k-2} + 1$, then 1 or $2^{k-3} + 1$, and so on, until we reach a collision pair of length 1 or length 2.

ALGORITHM: MakeExpandableMessage(h_{in}, k)

Make a $(k, k + 2^k - 1)$-expandable message.

Variables:

1. h_{tmp} is the current intermediate hash value.
2. C is a list of pairs of messages of different lengths; $C[i][0]$ is the first message of pair i, while $C[i][1]$ is that pair's second message.

Steps:

1. Let $h_{tmp} = h_{in}$.
2. For $i = 0$ to $k - 1$:
   - (m_0, m_1, h_{tmp}) = FindCollision($2^i + 1, h_{tmp}$)
   - $C[k - i - 1][0] = m_0$

- $C[k-i-1][1] = m_1$

3. Return the list of message pairs C.

Work: $k \times 2^{n/2+1} + 2^k \approx k \times 2^{n/2+1}$ compression function calls.

At the end of this process, we have an $k \times 2$ array of messages, for which we have done approximately $2^k + k \times 2^{n/2+1}$ compression function computations, and with which we can build a message consisting of between k and $k + 2^k - 1$ blocks, inclusive, without changing the result of hashing the message until the final padding block.

Producing a Message of Desired Length Finally, there is a simple algorithm for producing a message of desired length from an expanded message.

ALGORITHM: ProduceMessage(C, k, L)

Produce a message of length L, if possible, from the expandable message specified by (C, k).

Variables:

1. L = desired message length.
2. k = parameter specifying that C contains a $(k, k + 2^k - 1)$-expandable message.
3. C = a $k \times 2$ array of message fragments of different lengths.
4. M = the message to be constructed.
5. T = a temporary variable holding the remaining length to be added.
6. i = an integer counter.

Steps:

1. Start with an empty message $M = \emptyset$.
2. If $L > 2^k + k - 1$ or $L < k$, return an error condition.
3. Let $T = L - k$.
4. Concatenate message fragments from the expandable message together until we get the desired message length. Note that this is very similar to writing T in binary.
   - $i = 0$
   - While $T > 0$:
     - If $T > 2^{k-1-i}$, then:
       - $M = M||C[i][1]$
       - $T = T - 2^{k-1-i}$
     - Else:
       - $M = M||C[i][0]$
     - i = i + 1
5. Return M.

Work: Negligible (about k table lookups and string copying operations).

The result of this is a message of the desired length, with the same hash result before the final padding block is processed as all the other messages that can be produced from this expandable message.

3.2 More Efficient Expandable Messages with Fixed Points

There is a more efficient technique for building expandable messages when fixed points can easily be found in the compression function. For a compression function $h[i] = F(h[i-1], M[i])$, a fixed point is a pair $(h[i-1], M[i])$ such that $h[i-1] = F(h[i-1], M[i])$. Compression functions based on the Davies-Meyer construction[MvOV96], such as the SHA family [SHA02], MD4, MD5 [R92], and Tiger [AB96], have easily found fixed points. Similarly, Snefru[M90] has easily found fixed points. Techniques for finding these fixed points appear in an appendix; these techniques produce a pair $(h[i-1], M[i])$, but allow no control over the value of $h[i]$.

We can construct an expandable message using fixed points for about twice as much work as is required to find a collision in the hash function.

ALGORITHM: MakeFixedPointExpandableMessage($h[in]$)

Make an expandable message from initial hash value $h[in]$, using a fixed point finding algorithm.

Variables:

1. $h[in]$ = initial chaining value for the expandable messages.
2. FindRandomFixedPoint() = an algorithm returning a pair $(h[i], M[i])$ such that $h[i] = F(h[i], M[i])$.
3. A, C = two lists of hash values.
4. B, D = two lists of message blocks.
5. i, j = integers.
6. $M(i)$ = a function that produces a unique message block for each integer i less than 2^n.

Steps:

1. Construct a list of $2^{n/2}$ fixed points:
   - For $i = 0$ to $2^{n/2} - 1$:
     - h, m = FindRandomFixedPoint()
     - $A[i] = h$
     - $C[i] = m$
2. Construct a list of $2^{n/2}$ hash values we can reach from $h[-1]$:
   - For $i = 0$ to $2^{n/2} - 1$:
     - $h = F(h[in], M(i))$
     - $B[i] = h$
     - $D[i] = M(i)$
3. Find a match between lists A and B; let i, j satisfy $A[i] = B[j]$.
4. Return expandable message $(D[j], C[i])$.

Work: About $2^{n/2+1}$ compression function computations, assuming abundant memory.

If an n-bit hash function has a maximum of 2^k blocks in its messages, then this technique takes about $2^{n/2+1}$ work to discover $(1, 2^k)$-expandable messages. Producing a message of the desired length is trivial.

ALGORITHM: ProduceMessageFP(L, X, Y)

Produce a message of desired length from the fixed-point expandable messages.

Variables:
1. L is the desired length in message blocks; must be at least one and no more than the maximum number of message blocks supported by the hash.
2. X is the first message block in the expandable message.
3. Y is the second (repeatable) block in the expandable message.

Steps:
1. $M = X$.
2. For $i = 0$ to $L - 1$:
   - $M = M||Y$
3. Return M.

Work: Negligible work, about L steps.

3.3 Variants

The expandable messages found by both of these methods can start at any given hash chaining value. As a result, we can build expandable messages with many useful properties:

1. The expandable message can start with any desired prefix.
2. The expandable message can end with any desired suffix.
3. While both algorithms given here for finding expandable messages assume complete freedom over choice of message block, a variant of the generic method can be used even if the attacker is restricted to only two possible values for each message block[4].
4. The fixed-point method requires about $2^{n/2}$ possible values for each message block, but this is sufficiently flexible that for existing hash functions, it can typically be used with only ASCII text, legitimate sequences of Pentium opcodes, etc.
5. The trick from Joux can be used to make expandable messages that are also 2^k-collisions; that is, a set of 2^k different expandable messages for any length that the expandable message can accommodate.

4 Using Expandable Messages to Find Second Preimages

An n-bit hash function is supposed to resist second preimage attacks up to about 2^n work. That is, given one message M, the attacker ought to have to spend about 2^n work to find another message that has the same hash value as output.

4.1 The Long Message Attack

Here is a general (and previously known) way to violate Second-preimage resistence [MvOV96], which the Damgard-Merkle construction prevents from working: Start with an extremely long message: e.g., of 2^{55} blocks.

[4] This works in almost the same way as a natural extension of the Joux technique for finding multicollisions.

An attacker who wishes to find another message that hashes to the same value with SHA1 can do so by finding a message M such that, from the IV of the hash, $h[-1]$, $h^* = C(h[-1], M)$ yields a value h^* that matches one of the intermediate values of the hash function in processing the long message. Since the message has about 2^{55} such intermediate values, the attacker expects to need to try only about 2^{105} message blocks to get a match. That is, the attacker has 2^{55} available target values, so each message block he tries has about a 2^{-105} chance of yielding the same hash output as one of the 2^{55} intermediate hash values of the target message. He thus has a shorter message, which has the same hash output *up until the final block is processed.*

The length padding at the end of the Damgard-Merkle construction foils this attack. Note that in the above situation, the attacker has a message that is shorter than 2^{55}-block target message, which leads to the same intermediate hash value . But now, the last block has a different length field, and so the attack fails—the attacker can find something that's *almost* a second preimage, but the length block changes, and so the final hash output is different.

4.2 Long-Message Attacks with Expandable Messages

Using expandable messages, we can bypass this defense, and carry out a second-preimage attack despite the length block at the end. We start with a long message as our target for a second preimage. The attack works as follows:

ALGORITHM: LongMessageAttack(M_{target})

Find the second preimage for a message of $2^k + k + 1$ *blocks.*

Variables:

1. M_{target} = the message for which a second preimage is to be found.
2. M_{link} = a message block used to link the expandable message to some point in the target message's sequence of intermediate hash values.
3. A = a list of intermediate hash values
4. h_{exp}= intermediate chaining value from processing an expandable message.

Steps:

1. C = MakeExpandableMessage(k)
2. h_{exp} = the intermediate hash value after processing the expandable message in C.
3. Compute the intermediate hash values for M_{target}:
   - $h[-1]$ = the IV for the hash function
   - $m[i]$ = the ith message block of M_{target}.
   - $h[i] = F(h[i-1], m[i])$, the ith intermediate hash output block. Note that h will be organized in some searchable structure for the attack, such as a hash table, and that elements $h[0, 1, ..., k-1]$ are excluded from the hash table.
4. Find a message block that links the expandable message to one of the intermediate hash values for the target message after the kth block.

   – Try linking messages M_{link} until $F(h_{exp}, M_{link}) = h[j]$.
5. Use the expandable message to produce a message M^* that is j blocks long.
6. Return second preimage $M^*||M_{\text{link}}||m[j+1]||m[j+2]...m[2^k+k]$.

Work: The total work done is the work to find the expandable message plus the work to find the linking message.

1. For the generic expandable message-finding algorithm, this is $k \times 2^{n/2+1} + 2^{n-k+1}$ compression function calls.
2. For the fixed-point expandable message-finding algorithm, this is $3 \times 2^{n/2+1} + 2^{n-k+1}$

An Illustration To illustrate this, consider a second preimage attack on SHA1[SHA02]. The longest possible message for SHA1 is $2^{64} - 1$ bits, which translates into just under 2^{55} blocks. For simplicity, we will assume the target message is $2^{54} + 54 + 1$ message blocks (about 2^{60} bytes) long.

1. Receive the target message and compute and store all the intermediate hash values.
2. Produce a $(1, 54 + 2^{54})$-expandable message. This requires about 3×2^{81} compression function computations, because SHA1 permits easy finding of fixed points.
3. Starting from the end of the expandable message, we try about 2^{106} different message blocks, until we find one whose hash output is the same as one of the last $54+2^{54}$ intermediate hash values of the target message. This requires computing about 2^{106} compression functions on aveage.
4. Expand the expandable message to compensate for the message blocks of the target message skipped over, and thus produce a second preimage. This takes very little time.

Summary of the Attack The long-message attack can be summarized as follows: For a target message substantially less than $2^{n/2}$ blocks in length, the work is dominated by the long message attack. Thus, a second preimage attack on a 2^k-block message takes about 2^{n-k+1} compression function computations, assuming abundant memory.

4.3 Variations on the Attack

Some straightforward variations of this attack are also possible, drawing from the variations available to the expandable messages. For example, the algorithms for producing an expandable message work from any starting hash value, and are not affected by the message blocks that come after the expanded message. Thus, this attack can be used to “splice together” two very long messages, with an expandable part in the middle. Similarly, if it is important that the second preimage message start with the same first few hundred or thousand message blocks as the target message, or end with the same last few hundred or thousand blocks, this can easily be accommodated in the attack. Another variation is available by using Joux’s multicollision-finding trick, or the related ones described below: By setting up the expandable message to be a 2^u-multicollision, we can find 2^u distinct second preimages for a given long message, without adding substantial cost to the attack. Additionally, keyed constructions that leave the attacker with offline collision search abilities are vulnerable to the attack; for example, the "suffix mac" construction, $\mathrm{MAC}_K(X) = \mathrm{Hash}(X||K)$ is vulnerable to a second preimage attack, as well as the much more practical, previously-known collision attack.

Low-Memory and Parallel Versions of the Attack These methods for finding expandable messages assume unlimited memory. In the real world, memory is limited, and bandwidth between processing units and memory units is likewise limited. In the full paper, we will consider how the parallel meet-in-the-middle attack techniques of [vOW96,vOW99] can be applied to this attack.

Briefly, even a rather naive technique for mounting the attack with less memory still beats the 2^n bound for $n = 160$. Consider a second-preimage attack on a 2^{48} block message, in which we have 2^{48} n-bit memory locations to use. One simple way to do the meet-in-the-middle attack used to find a collision between two tables of 2^{80} items with 2^{48} memory locations is to simply run the 2^{80} items from the first table (generated computationally each time) against 2^{48} items from the second table, and to iterate this process 2^{32} times. This requires total work of $2^{80} \times 2^{32} = 2^{112}$ compression function computations, and it must be done 48 times, for a total of less than 2^{118} compression function computations. The resulting $(48, 48 + 2^{48} - 1)$-expandable message is then used in the long-message attack on the 2^{48}-block message, requiring about 2^{112} work. The attack is thus dominated by the difficulty of finding an expandable message, but at less than 2^{118} work, it is still substantially faster than the 2^{160} compression function computations expected to find a second preimage.

The enormous memory requirements apply only to finding expandable messages; the long-message attack requires a great deal of computation, but in general, it requires no more memory than is needed to hold the target message. Thus, it may make sense to precompute a single (a, b)-expandable message, and amortize the cost of doing so over many second-preimage attacks on different messages.

The attack can reuse an expandable message as the base for finding second preimages on as many target messages as may be provided.

5 Expandable Messages and Multicollisions

In [J04], Joux demonstrates a beautiful way to produce a large number of messages that collide for an iterated hash function, with only a little more work than is needed to find a single pair of messages that collide. Here, we demonstrate ways to use expandable messages to find multicollisions, and ways to combine the Joux technique with expandable messages to add flexibility to the structure of the multicollisions.

5.1 A Simple Multicollision Technique

Consider two (a, b)-expandable messages concatenated together. We can produce $(b - a + 1)$ different messages that all have the same value with this pair of expandable messages, by choosing to expand the first message to $a + i$ blocks, and the second to $b - i$ blocks, for $0 \le i \le b - a$.

In the case where the compression function allows fixed points to be easily found, a $(2, Z)$ expandable message (where Z is the maximum number of blocks that the hash function will process) can be produced for twice the work of a collision search. Concatenating two such expandable messages gives a $Z - 2$-multicollision for four times the work of collision search. For concreteness, with SHA1 (which allows easy finding of fixed points), we have $Z \approx 2^{54}$, and thus can find $2^{54} - 2$-collisions for only 4×2^{80} work, rather than the approximately 54×2^{80} work of the Joux attack[5].

5.2 A More Powerful Method for Finding Multicollisions

Consider concatenating k (a, b)-expandable messages. We can vary the lengths of the different expandable messages so that the total message length stays the same, and produce, for long messages, enormously large numbers of multicollisions.

For example, a $(54, 54+2^{54}-1)$-expandable message costs about $54 \times 2^{n/2+1}$ work to construct, and a set of 16 of them costs 16 times as much work. However, these then permit a $\binom{2^{54}}{16}$-collision to be constructed. This is cheaper than the Joux attack for the same size of multicollision.

With hash functions with easily findable fixed points, the multicollisions are still easier to find: each $(2, 2^{54})$-expandable message costs only $2^{n/2+1}$ work. With $64 \times 2^{n/2}$ work, the Joux attack finds a 2^{64}-collision; this technique finds a $\binom{2^{54}}{64}$-collision, albeit an impractically long one.

5.3 Combining With Joux

Finally, it is possible to combine Joux multicollisions with expandable-message multicollisions. This allows multicollisions to be constructed that look quite different from the Joux multicollisions, and are somewhat more flexible in structure. This may allow Joux attacks to go forward even on cascaded constructions that attempt to foil his attack.

6 Conclusions and Open Questions

In this paper, we have described a generic way to carry out long-message second preimage attacks, despite the Damgard-Merkle strengthening done on all modern hash functions.

These attacks are theoretical because 1) they require more work than is necessary to find collisions on the underlying hash functions, and 2)

[5] We note, however, that Joux's multicollisions are of reasonable length, whereas ours are not.

the messages for which second preimages may be found are generally impractically long. However, they demonstrate some new lessons about hash function design:

1. An n-bit iterated hash function provides fundamentally different security properties than a random oracle with an n-bit output. This was demonstrated in one way by Joux in [J04], and by another here.
2. An n-bit iterated hash function begins to show some surprising properties as soon as an attacker can do the work necessary to find collisions in the underlying compression function.
3. An n-bit iterated hash function cannot support second-preimage resistance at the n-bit security level, as previously expected, for long messages.
4. Easily found fixed points in compression functions (such as those based on the Davies-Meyer construction) allow an even more powerful second-preimage attack, as well as allowing multicollisions to be found even more cheaply than by the Joux attack.

We believe that the important lesson here is that the standard construction of iterated hashes from Merkle and Damgard does not provide all the protection we might expect against attackers that can do more than $2^{n/2}$ compression function computations. In some sense, the hash function is "brittle," and begins to lose its claimed security properties very quickly once the attacker can violate its collision resistance by brute force.

We believe these results, when combined with those of Joux, require a rethinking of what promises are being made by an iterative hash function with an n-bit intermediate state. We see three sensible directions for this rethinking to take:

1. A widespread consensus that an n-bit iterated hash function should never be expected to resist attacks requiring more than $2^{n/2}$ operations. This would invalidate widespread uses of hash functions in cryptographic random-number generation, as in [KSF99,DHL02,B98], key derivation functions [AKMZ04,NIST03,X963], and many other applications, and seems the least palatable outcome.
2. A clear theoretical treatment of the limits that exist for n-bit hash functions, and precisely what attacks more demanding than collision search they may be expected to resist. (For example, neither our results, nor those of Joux, appear to be applicable when the attacker cannot do offline collision search; perhaps this observation can be formalized.)
3. New constructions for hash functions in the vein of [L04], which maintain much more than n bits of intermediate state in order to make collision attacks on intermediate states harder (require 2^n work).

The important open questions are defined by the above list. Specifically, absent a solid theoretical treatment of the security properties of n-bit iterative hashes, along the lines of [PGV93] and [BRS02], but expanded to deal thoroughly with the full hash construction, it is difficult to justify using them in applications requiring more than $n/2$ bits of security with any confidence. We hope this work spurs such a treatment.

7 Acknowledgements

The authors wish to thank Bill Burr, Morris Dworkin, Niels Ferguson, Phil Hawkes, Julie Kelsey, Ulrich Kuhn, Stefan Lucks, Bart Preneel, Vincent Rijmen, and David Wagner for useful comments and discussions on the results in this paper.

References

[AB96] Anderson and Biham, "Tiger—A Fast New Hash Function," in proceedings of FSE96, Springer-Verlag, 1996.

[AKMZ04] Adams, Kramer, Mister, and Zuccherato, "On the Security of Key Derivation Functions," *Proceedings of the 7th Information Security Conference (ISC '04)*, Palo Alto, CA, USA, Springer-Verlag, 2004 (to appear).

[B98] Baldwin, "Preliminary Analysis of the BSAFE 3.x Pseudorandom Number Generators," *RSA Laboratories Bulletin No. 8*, RSA Laboratories, 1998.

[BRS02] Black, Rogaway, and Shrimpton, "Black-Box Analysis of the Block-Cipher-Based Hash-Function Constructions from PGV," *Advances in Cryptology–Crypto 02 Proceedings*, Springer-Verlag, 2002.

[BS93] Biham and Shamir, *Differential Cryptanalysis of the Data Encryption Standard*, Springer-Verlag, 1993.

[D89] Damgard, "A design principle for hash functions," *Advances in Cryptology–Crypto 89 Proceedings*, Springer-Verlag, 1989.

[DHL02] Desai, Hevia, and Yin, "A Practice-Oriented Treatment of Pseudorandom Number Generators," *Advances in Cryptology–Eurocrypt 02 Proceedings*, Springer-Verlag, 2002.

[J04] Joux, "Multicollisions in Iterated Hash Functions. Applications to Cascaded Constructions," *Advances in Cryptology–Crypto 2004 Proceedings*, Springer-Verlag, 2004.

[KSF99] Kelsey, Schneier, and Ferguson, "Yarrow-160: Notes on the Design and Analysis of the Yarrow Cryptographic Pseudorandom Number Generator," SAC 1999.

[L04] Lucks, "Design Principles for Iterated Hash Functions," IACR preprint archive, http://eprint.iacr.org/2004/253.pdf, 2004.

[M89] Merkle, "One way hash functions and DES," *Advances in Cryptology–Crypto 89 Proceedings*, Springer-Verlag, 1989.

[M90] Merkle, "A fast software one-way hash function," *Journal of Cryptology*, 3(1):43–58, 1990

[MOI91] Miyaguchi, Ohta, Iwata, "Confirmation that some hash functions are not collision free," *Advances in Cryptology–Crypto 90 Proceedings*, Springer-Verlag, 1990.

[MvOV96] Menezes, van Oorschot, Vanstone, *Handbook of Applied Cryptography*, CRC Press, 1996.

[NIST03] *NIST Special Publication 800-56, Recommendations on Key Establishment Schemes*, Draft 2.0, Jan 2003, available from csrc.nist.gov/CryptoToolkit/kms/keyschemes-jan02.pdf.

[PGV93] Preneel, Govaerts, and Vandewalle, "Hash Functions Based on Block Ciphers: A Synthetic Approach," *Advances in Cryptology–Crypto 93 Proceedings*, Springer-Verlag, 1993.

[R92] Rivest, "The MD5 message-digest algorithm," RFC1321, April 1992.

[SHA02] National Institute of Standards and Technology, *Secure Hash Standard,*' FIPS180-2, August 2002.

[vOW99] van Oorschot and Wiener, "Parallel collision search with cryptanalytic applications," *J. of Cryptology*, 12:1–28, 1999.

[vOW96] van Oorschot and Wiener, "Improving Implementable Meet-in-the-Middle Attacks by Orders of Magnitude," *Advances in Cryptology–Crypto 96 Proceedings*, Springer-Verlag, 1996.

[X963] "ANSI X9.63—Public Key Cryptography for the Financial Services Industry: Key Agreement and Transport Using Elliptic Curve Cryptography," American Bankers Association, 1999. Working Draft.

A Finding Fixed Points Efficiently in Many Compression Functions

Finding fixed points in many hash compression functions is simple.

Most widely used hash functions have compression functions designed around very large block-cipher-like constructions, following the general Davies-Meyer model. For the SHA and MD4/MD5 families, as well as Tiger, if $E(K, X)$ is a very wide block cipher, with K the key and X the value being encrypted, then the compression function is:

$$C(H, M) = E(M, H) + H$$

for some group operation "+". For these compression functions, it is possible to compute the inverse of this block-cipher-like construction, which we can denote as $E^{-1}(K, X)$. This makes it possible to find fixed points in a simple way, as discussed in [MOI91]:

1. Select a message M.
2. Compute $H = E^{-1}(M, 0)$.
3. The result gives a fixed point: $C(H, M) = H$.

A property of this method for finding fixed points is that the attacker is able to choose the message, but he has no control whatsoever over the hash value that is a fixed point for a given message. Also note that for these hash functions, each message block has exactly one fixed point.

Snefru is derived from a block-cipher-like operation that operates on a much larger block than the hash output, and which effectively has a fixed "key." Let $E(X)$ be this fixed "encryption" of a block. Further, let n be the hash block size, m be the message block size, $\text{lsb}_n(X)$ be the least significant n bits of X, and $\text{msb}_n(X)$ be the most significant n bits of X. Note that $E(X)$ operates on $n + m$-bit blocks.

The compression function is derived from $E(X)$:

$$C(H, M) = \text{lsb}_n(E(H||M)) + H$$

where the hash input and output are each n bits wide, and where $\text{lsb}_x(Y)$ represents the least significant x bits of the value Y. We can find fixed points for Snefru-like compression functions as follows, letting $E^{-1}(X)$ be the inverse of $E(X)$ once again:

1. Choose any X whose least significant n bits are 0.
2. Compute $Y = E^{-1}(X)$.
3. Let $H = \text{msb}_n(Y)$ and $M = \text{lsb}_m(Y)$.
4. The result gives a fixed point: $C(H, M) = H$.

This method gives the attacker no control over the message block. Unlike the Davies-Meyer construction, there is no guarantee that a given message block has even one fixed point; we would expect for some message blocks to have many, and for others to have none.

Note that the Snefru construction could easily be altered to make fixed points very hard to find, when the size of the message and hash blocks are equal, by the compression function as:

$$C(H, M) = \text{lsb}_n(E(H||M)) + H + M$$

or

$$C(H, M) = \text{lsb}_n(E(H||M)) + H + M + \text{msb}_m(E(H||M))$$

Also note that many other compression function constructions, such as the Miyaguchi-Preneel construction used by Whirlpool and N-Hash, do not permit a generic method for finding fixed points.