Stronger Hash Functions Through Block Shuffling

Elizabeth Reid
Chris Wilkens
Christina Wright

May 14, 2008

Abstract

In this paper we present a new block chaining approach for iterative hash functions based on deterministic block shuffling. We prove that our new scheme does not weaken collision resistance and that it is provably secure against message extension attacks under very specific circumstances. Along the way, we prove that no hash function with a finite state size can be provably secure against message extension attacks if it only reads the input once. Finally, we implement our chaining approach and compare its performance to existing hash functions.

1 Introduction

Hash functions are currently a hot topic of research. Because they are widely used in many contexts [10], it is important to have hash functions that are secure. Yet, current implementations of hash functions are imperfect and subject to attack [13]. While some attacks are very specific to the particular hash algorithm [12], work has also been done to scrutinize commonly used hash constructs [7].

Particularly, the Merkle-Damgård structure [8][11] is frequently used since it provably preserves collision resistance of the underlying compression function. Unfortunately, it has been found to have other flaws [4], such as failure to preserve random oracle and psuedorandom properties [1]. It has also been found to be vulnerable to multicollision attacks. This is problematic because others have shown that useful collisions can be obtained from abstract collisions [9].

An attack on the Merkle-Damgård structure is the Herding attack of Kelsey and Kohno [2]. In this attack they append a string s to a prefix p in order to force the message to a desired hash h. To be secure against this type of attack, it must be difficult to compute s given p and h where $hash(p||s) = h$.

Another type of multicollision attack Merkle-Damgård structure is subject to is the message extension attack. That is, when messages A and B collide then the messages $A||C$ and $B||C$ collide for any C. It was shown that reordering the message blocks [5] does not solve this problem. More strongly, no reordering and repeating of the message blocks will improve the multicollision resistance of the Merkle-Damgård hash [3].

Daum and Lucks give a clear demonstration of how dangerous this attack can be by generating two PostScript files with the same MD5 hash[6]. The two documents look perfect, partly because the MD5 collision is embedded in a constant defined at the top of the file. The subsequent code, which is the same for both files, checks the value of the constant and prints one message or the other.

Solutions to the message extension attack have been found. Such as making the input messages prefix-free. One simple solution is double hashing: take $h(h(m)||m)$. This approach requires reading the message twice, but is provably secure against an extension attack. Thus, we will examine a potential solution to the Merkle-Damgård extension vulnerability which is constrained to read in the message only once.

In this paper, we will explore hash functions. Our primary contribution is to analyze a new block chaining approach based on deterministic block shuffling. In section 2, we describe the details of our algorithm. In section 3 we prove some basic properties of our construction, and in section 4 we give a few proofs of security. Finally, we discuss why our algorithm, h_{MIX}, is not immediately vulnerable to existing Merkle-Damgård attacks in section 5 and implement our scheme in section 6.

Along the way, we present the interesting result that any hash function with finite state cannot be provably secure against message extension attacks if it only reads the message once. We do this in section 4.

2 The Algorithm

Let h_1 and h_2 be two standard, Merkle-Damgård hash functions with initialization vectors k_1 and k_2 respectively and an output size of $|h|$ bits and a block size of β. Let $N = 2^n$ be a parameter of our mixer chosen such that $N * n = |h|$. Next, let $M = m_1||m_2||\ldots||m_b$ be a message divided into b blocks of length $\beta - n$. Finally, let k_3 be another key that we will use later. We wish to hash M as $h_{MIX}(M)$.

If $b \neq a \cdot N$ for some a (i.e. b is not a multiple of N,) then pad it with $1000\ldots$ followed by the message length to make h a multiple of N (this is the same as a standard Merkle-Damgård strengthening [8][11].) When we refer to M hereafter it is this padded message.

Our design will consist of three stages: a feeder buffer, a mixing array, and an output hash function. These stages are defined as follows:

1. *Feeder.* The feeder is FIFO queue that holds N *message-sized* (i.e. $\beta - n$-bit) blocks. Each step, one block is removed from the queue and sent to the mixer while the next block of the message is put in the queue.

2. *Mixer.* The mixer holds N *hash-sized* (i.e. β-bit) blocks. Each step, a block selected from a "random-looking" function (say, index e) and is removed from the mixer and sent to the hash. Then, e is appended to the head block of the feeder the resulting value is put into the mixer at location e. The *ordering* of the blocks inside the mixer is said to be the blocks listed in order of their location in the mixer.

3. *Hash.* This stage uses h_2 to produce the final output hash. It uses a standard Merkle-Damgård chaining structure and chains blocks in the order that they are received from the mixer. The order that blocks enter hash stage will be known as the *hash ordering.*

We will use the mixer to shuffle the blocks of M. The resulting chaining algorithm may be described as follows:

1. Pad M if necessary.

2. Fill the mixer with empty blocks (i.e blocks with the value $e||0$ where e is again the index of the mixer location of the block.)

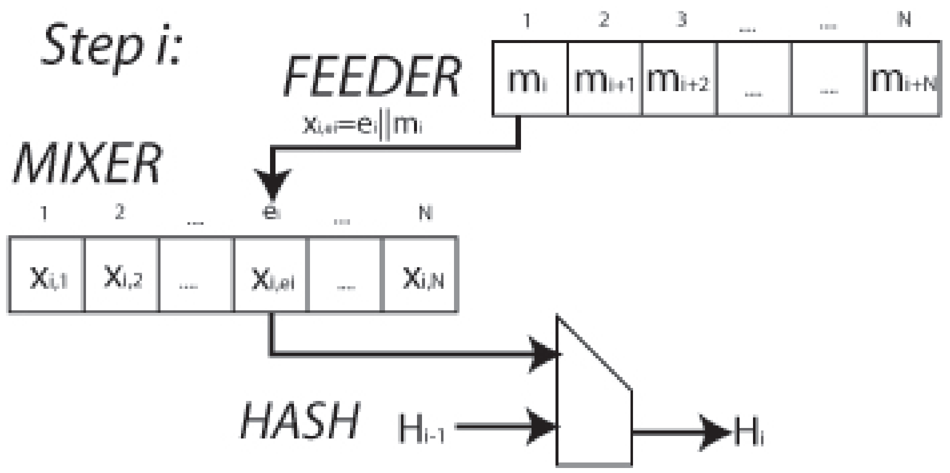

Figure 1: The algorithm at step i. A block is removed from the feeder and sent to the hash. The location is filled with the next block in the feeder.

3. Load the first N blocks of M (i.e. $m_1, \ldots m_N$) into the feeder.
4. Pick a random integer e on the range $[1, 2, \ldots N]$.
5. Send the block stored at location e in the mixer to the hash stage.
6. Take the next block m from the feeder and put $e||m$ it in location e in the mixer.
7. Repeat steps 4-6 until the feeder is empty.
8. Send all N blocks in the mixer to the hash, preserving their ordering from the mixer.

The loop in step 7 will run exactly once for each block in M and then terminate. Thus, it runs b times. This gives us a framework for defining steps in the system. Let e_i be the value of e chosen when block i is moved from the feeder to the mixer. Let $FEEDER_i = W_i = \{w_i^1, \ldots w_i^N\}$ be the blocks in the feeder when block i is at the head of the feeder. (Note that each w_i is $\beta - n$ bits long, so $FEEDER_i$ is $N \cdot (\beta - n)$ bits.)

Likewise, let $MIXER_i = X_i = \{x_i^1, \ldots x_i^N\}$ be the blocks of the mixer at the same time. In the mixer, each block is a value of e concatenated with a message block, which we will denote as μ. (Note that the mu represent a permutation of the message blocks of M as they leave the mixer. We introduce mu because there is no concise way to represent which block of the message is actually in the mixer at the time of ejection.) This gives $x_i^j = j||\mu_i^j$ For consistency, $MIXER_{b+1}$ is the state of the mixer once all blocks have left the feeder.

Finally, let γ_i be the block sent to the hash stage in step i. It follows that the *hash ordering* is $\Gamma = \gamma_1 \ldots \gamma_{b+N}$.

As an exercise, we observe that the algorithm above to compute $h_{MIX}(M)$ may be completely defined as follows: *Initialization:*

$$x_1^j = e_j||0$$

$$w_1^j = m_j$$

Recursion:

$$\gamma_i = \begin{cases} x_i^{e_i} & i \leq b \\ x_i^j & i = b + j \text{ and } 1 \leq j \leq N \end{cases}$$

$$x_{i+1}^{e_i} = \begin{cases} e_i||w_i^1 = e_i||\mu_i^j & j = e_i \\ x_i^j & \text{otherwise} \end{cases}$$

$$w_{i+1}^j = \begin{cases} 0 & i + j > b \\ m_{i+N} & j = N \\ w_i^{j+1} & \text{otherwise} \end{cases} = \begin{cases} m_{i+j} & i + j \leq b \\ 0 & \text{otherwise} \end{cases}$$

Output:

$$\begin{aligned} h_{MIX}(M) &= h_2(\Gamma) \\ &= h_2(\gamma_1|| \ldots \gamma_b||\gamma_{b+1}|| \ldots \gamma_{b+N}) \\ &= h_2(x_1^{e_1}||x_2^{e_2}|| \ldots x_i^{e_i}|| \ldots x_b^{e_b}||MIXER_{b+1}) \\ &= h_2(e_1||\mu_1^{e_1}||e_2||\mu_2^{e_2}|| \ldots e_i||\mu_i^{e_i}|| \ldots e_b||\mu_b^{e_b}||1||\mu_{b+1}^1|| \ldots N||\mu_{b+1}^N) \end{aligned}$$

Given the above algorithm, we must specify a way to deterministically select “random-looking” e values. We will generate them using another hash function, h_1.

Specifically, we will compute N values of e at a time as the hash of an N-block chunk of the message. We get the mathematical form:

$$E_{cN} = e_{cN+1}||e_{cN+2} \ldots ||e_{(c+1)N} = k_3 \oplus h_1(FEEDER_{cN+1}).$$

(The rationale for this choice is to prevent the adversary from being able to specify a single value of e by changing one block input. Originally, we computed each e value based on the instantaneous contents of the feeder and mixer. Unfortunately, in this scheme the adversary could independently tweak blocks to achieve the desired block order.)

3 Preliminary Results

In this section, we prove the following statements about our structure:

Lemma 1: No distinct messages M_1 and M_2 have the same Γ (i.e. $M_1 \neq M_2 \Rightarrow \Gamma_1 \neq \Gamma_2$.)

Lemma 2: Let M yield hash ordering Γ. If $M' = M||A$ where $A = a_1|| \dots A_{b_a}$, then $h_1(A_1|| \dots A_N) = k_3 \oplus (1||2|| \dots N)$.

Lemma 3: Exactly $\frac{1}{N^b} \leq \frac{1}{N^N}$ of the proper length Γ sequences are possible for a message of b blocks.

To prove these, we first demonstrate that if we know the entire hash ordering Γ, then we can recover M. First, for each $\gamma_i = e_i||\mu_i^{e_i}$, we know two things about the values at location e_i in the mixer:

1. The block at location e_i was $\mu_i^{e_i}$ prior to step i. In fact, it held this value since the last step t that $e_t = e_i$.
2. The block at location e_i will be m_i until location e_i is written again, i.e. the next time t that $e_t = e_i$, *unless* $i > b$, in which case we have already read the entire message and are just flushing the mixer. In the latter case, the value at location e_i is not well defined.

As a result, if we know two steps t_1 and t_2 that correspond to "consecutive" ejections from location e, then we know $m_{t_1} = \mu_{t_2}^e$. The μ value may be read from Γ, so we actually know m_{t_1}. This immediately demonstrates that for all i where e_i is not the last occurrence of a particular value of e, we know m_i.

Next, since we flush the mixer at the end by reading blocks in order, we know that the last occurrence for any given e is for some $i > b$. Consequently, we may conclude that for all $i \leq b$, we know m_i. Thus, we know M.

Proof of Lemma 1: Based on the previous discussion, given Γ we can deterministically recover M. It follows that no two messages M may generate the same Γ, and therefore any distinct input messages have distinct Γ sequences.

Proof of Lemma 2: Given a hash ordering Γ that corresponds to some message M, if Γ is a subsequence of Γ' for some M' (say, for steps t_1 trough t_2,) then we know $m'_{t_1} \dots m'_{t_2-N} = M$. It follows that if $\Gamma' = \Gamma||B$, then $M' = M||A$ for some blocks A.

Now, we know that the last N blocks of Γ were flushed from the mixer in order. It follows that they have e values $1, 2, \dots N$. Moreover, when we append A, those e values will be computed as $k_3 \oplus h_1(a_1|| \dots a_N)$. Since the e values must remain the same, it follows that $k_3 \oplus h_1(a_1|| \dots a_N) = (1||2|| \dots N)$.

Proof of Lemma 3: For a given block length b, there are $2^{b \cdot (\beta - n)}$ possible input messages and $2^{b \cdot \beta}$ possible Γ sequences. Since each message corresponds to exactly one Γ sequence, it follows that the fraction of Γ sequences that correspond to valid messages is $\frac{1}{2^{b \cdot n}} = \frac{1}{N^b}$.

Note that since we pad the message, $b \geq N$ and therefore $\frac{1}{N^b} \leq \frac{1}{N^N}$.

4 Provable Security

Now, we consider the provable security characteristics of this scheme. We have a few preliminary results for collision finding and message expansion attacks, but believe that more may be possible.

4.1 Message Extension Attacks

Now, we wish to consider the security of our scheme against message extension attacks. We cannot prove a general statement of security (in fact, we disprove it,) but we can give some limited proofs.

We will use the fact that if messages M_1 and M_2 that collide in h_{MIX}, they must have hash orderings Γ_1 and Γ_2 that collide in h_2. Assume that an adversary is successful in completing a message extension attack, namely, appending blocks A to both M_1 and M_2 while maintaining the collision. One of the following must have occurred:

1. The adversary turned Γ_1 and Γ_2 into $\Gamma'_1 = \Gamma_1||A$ and $\Gamma'_2 = \Gamma_2||A$.
2. The adversary turned Γ_1 and Γ_2 into Γ'_1 and Γ'_2 of a different form that still form a collision in h_2.

Lemma: Extending the message by the first method is as hard as inverting h_1.

Proof: We consider an easier problem: pure appending. To purely append A to M, the resulting Γ must be of the form $\Gamma||B$. We may reduce inverting h_1 (i.e finding x such that $h_1(x) = y$ for some specified y) to pure appending as follows:

1. Take $k_3 = (1||2 \dots N) \oplus y$.
2. Pick an arbitrary message M and purely append blocks A to M.
3. By lemma 2, $h_1(a_1|| \dots a_N) = k_3 \oplus (1||2|| \dots N) = y$.
4. take $x = a_1|| \dots a_N$.

Thus, by reduction, pure appending is as hard as inverting h_1. Note also that x will always have the same length. It follows that this is as hard as the potentially harder problem of finding an x of length $N \cdot (\beta - n)$ such that $h_1(x) = y$.

Now, the second case is much harder. First, we conjecture that if the adversary has no control over M_1 and M_2, then extension is still hard.

Conjecture: If the adversary has no control over M_1 and M_2, then the second case is hard.

Rationale: First, we assume that Γ_1 and Γ_2 share no intermediary values in h_2. Observe that no reordering will happen before the last N blocks of Γ. Consequently, h_2 will have at most $N-1$ new intermediary values. If the two messages had no common intermediary (or final) values initially, then it is unlikely that they will have any after shuffling. Consequently, the probability of causing a permutation of the message blocks that still produces a collision is low.

Another reason why this is plausible is that many attacks on the Merkle-Damgård structure require that the adversary select the target hash value (e.g. the herding attack [2].) In this case, we don't allow the adversary to do this, therefore it seems likely that it is not easy.

In the general case, we may prove the following much weaker statement about extension resistance:

Lemma: The second case is as hard as finding Γ'_1 and Γ'_2 where there is no A such that $\Gamma'_1 = \Gamma_1||A$, $\Gamma'_2 = \Gamma_2||A$, and $h_2(\Gamma'_1) = h_2(\Gamma'_2)$.

Proof: This is almost just a restatement of the second case. We know that Γ'_1 and Γ'_2 must be a collision in h_2, and we can recover M_i from Γ_i, so message extension must be as hard as finding such a collision.

Unfortunately, this reduction is as strong as we can expect to prove. Consider a collision in which the last N blocks of Γ_1 and Γ_2 are identical: one may conclude that the final states of the mixers are identical. One may also conclude that the value of the output hash is is not only identical at the end, but identical before the mixer is flushed. Consequently, any blocks may be appended to M_1 and M_2 while maintaining the collision. Therefore, in this case message extension is easy.

Theorem: Any hash function h that reads the input only once and has a finite n-bit internal state cannot be provably secure against message extension attacks. Specifically, for any such function we can construct a pair of messages for which message extension is easy.

Proof: First, we observe that any deterministic hash function h may be modeled as a 3-tuple of functions $(I(IV, m_1), U(S, m_i), F(S))$ defined as follows:

1. $I : \{0,1\}^{|IV|}, \{0,1\}^{|m_i|} \rightarrow \{0,1\}^n$ takes the initialization vector and the first message block and outputs some n-bit state.

2. $U : \{0,1\}^n, \{0,1\}^{|m_i|} \rightarrow \{0,1\}^n$ takes the previous state S and a new message block and outputs a new state.

3. $F : \{0,1\}^n \rightarrow \{0,1\}^m$ takes the final state after reading the entire input message and produces an output.

This is possible because h has a finite amount of internal state, therefore it can save at most n bits of information between block reads. In this framework, the final hash can be written as

$$h(M) = F(U(U(\ldots U(I(IV, m_1), m_2) \ldots, m_{b-1}), m_b)).$$

Let the final state before F be S_b, i.e.

$$S_b = U(U(\ldots U(I(IV, m_1), m_2) \ldots, m_{b-1}), m_b).$$

For any such function h, there exists a pair of messages of length $n+1$ that collide in S_b by the pigeonhole principle. Call these messages M_1 and M_2. Clearly, since F is deterministic, M_1 and M_2 collide in h. Now we append A to both messages. Since U is deterministic and both messages will see the same blocks of A, the two hashes will be the same for the remainder of the computation. Thus, for any message blocks A, $h(M_1||A) = h(M_2||A)$.

Consequently, any hash function h that reads the message only once and has a finite state cannot be provably secure against message extension attacks.

4.2 Collision Resistance

While message extension attacks are dangerous, the biggest question surrounding hash functions today is their collision resistance. Ideally, we would like to demonstrate that our scheme has improved collision resistance over its constituent components. We are not currently able to do this. However, we are at least able to prove that our scheme has not become weaker.

Lemma X: Finding collisions in h_{MIX} is at least as hard as finding collisions in h_2.

Proof: As shown earlier, if two different messages M_1 and M_2 collide, they must have different hash orders. Consequently, in order for the pair to be a collision in h_{MIX}, Γ_1 and Γ_2 must be a collision in h_2. Consequently, if we can find a collision in h_{MIX} we can immediately read off a collision in h_2.

5 Hypothesized Resistance to other Attacks

There are certainly many approaches from which we may attack h_{MIX}, but many of the obvious ones don't immediately work. For example, the multi-collision and herding attacks of [5][3][2] do not directly apply to structure as it is defined. In this section we will discuss a few approaches to attacking h_{MIX}, suggesting why it isn't immediately vulnerable to existing attacks.

5.1 Attacking h_1

One possible attack is to start by finding collisions in h_1. However, it isn't clear that collisions in h_1 have any relevance to collisions in the final output. Since it is impossible to have two messages with the same Γ, they are unnecessary.

5.2 Attacking h_2

Given that finding collisions is (thus far) only provably as finding collisions in h_2, it seems much more plausible that one could attack h_2 to find some Γ and then just hope that the resulting Γ actually corresponds to a message. The obvious flaw with this approach is that it is highly unlikely that a random Γ actually corresponds to a message. As we showed in lemma 3, the probability that a randomly selected Γ actually corresponds to a message is $\frac{1}{N^b}$ where b is the number of blocks in the input message (alternatively, N fewer than the number of blocks in Γ.)

For example, one might apply one of the approaches in [5][3][2] to find multicollisions in h_2. This gives us 2^k collisions in h_2. An adversary might hope that at least a few of these possible Γs would correspond to valid messages M. However, each multicollision is at least k blocks long. Consequently, the block length b of a message associated with one of Joux's multicollisions is at least $k-N$ blocks long. Finally, any valid Γ is at least $2N$ blocks long, so $k-N \geq N$ Therefore in expectation the adversary will have $\frac{2^k}{2^{(k-N)\cdot n}} \leq \frac{1}{2^{(n-2)\cdot k}}$ collisions, which then makes the probability of getting a single collision in h_{MIX} from a multicollision in h_2 is very small. Thus, on the surface, even using multicollision attacks against h_2 doesn't give an easy advantage.

5.3 Attacking h_{MIX} as a Merkle-Damgård Hash Function

Our hash function h_{MIX} is not immediately based on a Merkle-Damgård hash structure. However, if we look closely, it still may be related to one and attacked as such. Let the compression function be

$$f(h_2(\gamma_1 \ldots \gamma_{cN})||MIXER_{cN+1}, m_{cN+1}|| \ldots m_{(c+1)N})$$

$$= h_2(\gamma_1 \ldots \gamma_{(c+1)N})||MIXER_{(c+1)N+1}.$$

Any messages which collide in a Merkle-Damgård construction using the above compression function will also collide in h_{MIX}. Consequently, all the work on Merkle-Damgård hash functions may be applied to this new function to find collisions h_{MIX}. Even worse, the collisions found by such an attack are precisely the ones that render h_{MIX} most vulnerable to extension attacks.

Additionally, attacks that require the adversary to select the target value before hand (such as the herding attack of [2]) can also be applied to this new hash function because the adversary can easily compute the h_{MIX} value from the output of f.

The increased difficulty of attacking h_{MIX} would come from the large number of output bits, essentially making the process of finding collisions much harder.

6 Implementation

To test the practicality of our system, we implemented it in C with a design based on the theoretical outline above. However, some details were tweaked in order to decrease the run time of the function.

6.1 Feeder

While the behavior of the Feeder replicates that of a queue, we chose to implement it as an array, which allows for much faster input and output of data. The values in the Feeder are loaded into it and moved out of it in chunks with no data processing required in between, so *fread()* can be used to load the data directly into the Feeder array, which is very efficient. Furthermore, the hash function used to process the values in the Feeder (to determine the order of movement of the blocks) also requires that the data be stored in an array. To temporarily move the values into an array and then back into a queue is unnecessary and inefficient.

Holding all of the values in the Feeder together as an array also allows them to be cast to a bitmap to speed up the transfer of 27 bits at a time between the Feeder and the Mixer, which would be impossible to do with traditional data types. Thus, by forcing all of the data together into an array, it allows for significantly faster transfer out as well as in.

6.2 Mixer

The Mixer is also represented as an array, although of a longer length than the Feeder array. This is because the Mixer holds not only the data from the file being hashed, but also tags each block of data with the e value associated with its location. Because these e values never change within the array, they can be inserted at the beginning of the hash and never removed or replaced afterward.

The Mixer array is an array of ints rather than an array of chars so that it is easier to do bitwise computations to transfer the proper bits from the

Feeder into the Mixer while still retaining the e value tags at the beginning of each block. The Feeder array is temporarily turned into a bitmap so that the 27 remaining bits required to fill the 32 bit block in the int Mixer array can be extracted quickly. The first 5 bits from the Mixer array are joined with the 27 bits from the Feeder and inserted into the 32 bit portion allocated for each index in the Mixer array.

6.3 Hash

We used the SHA-1 hash implemented in the OpenSSL library for both of the required uses of a commercial hash function in our code. The first use is to generate the "random" order in which blocks are moved out of the Mixer and replaced with new values from the Feeder. This is done with a single call to the *SHA1( )* function with all of the data stored in the Feeder array given to it at once. The output of the function is a 160-bit "random" value, which can be converted to 32 "random" numbers between 0 and 31 by breaking it into a bitmap of 5-bit numbers that indicate the index values to be moved.

The second use of the SHA-1 hash is to hash the output of the Mixer to determine the overall result of the function. However, because the Mixer releases blocks of information rather than all of the values to be hashed at once, we used OpenSSL's functions that allowed us to hash the values in pieces so that we would not have to store the entire file in memory before hashing it. This involved three functions in OpenSSL's library: *SHA1-Init( )*, *SHA1-Update( )* and *SHA1-Final( )*. *SHA1-Init( )* and *SHA1-Final( )* were used to initialize and finalize the hashing, respectively, and *SHA1-Update( )* was used to feed the hash the new values whenever they became available.

6.4 Speed Analysis

Our hash function runs slower than the SHA-1 hash function, which we expected. This is not a surprise because our hash function uses the SHA-1 hash function to go through all of the data, but we also have to move the values from the Feeder table into the Mixer table, and then re-hash them again with another use of SHA-1. Furthermore, the *SHA1( )* code found in OpenSSL has been refined and streamlined through the efforts of multiple computer scientists who undoubtedly know more C than we do, especially tricks to streamline specific code. Thus, our code currently runs roughly twice as slow as the OpenSSL *SHA1()* function, but could probably be made

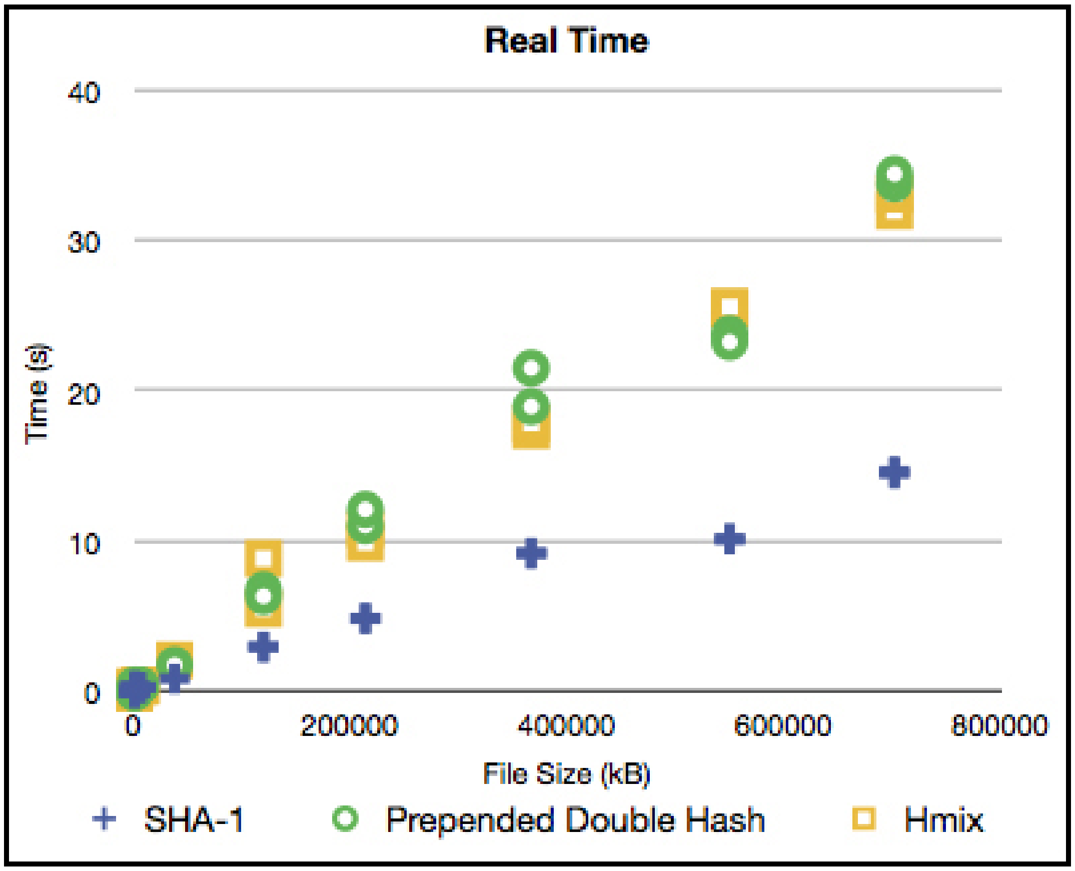

Figure 2: Real time statistics of h_{MIX} - The double hashing scheme and h_{MIX} both require roughly twice the time of SHA-1. The extra hashing in h_{MIX} is balanced by the file access time incurred by the double hashing scheme.

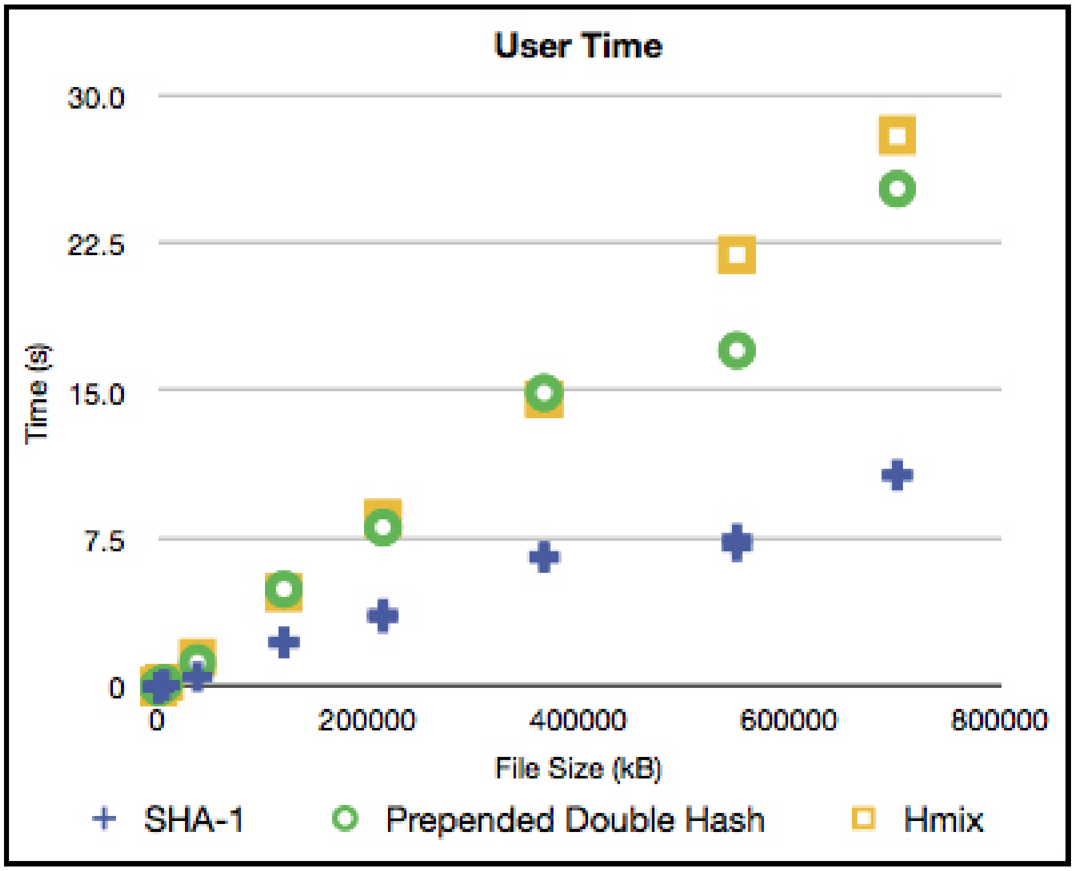

Figure 3: User time statistics of h_{MIX} - The double hashing scheme and h_{MIX} both require roughly twice the time of SHA-1. The extra hashing in h_{MIX} makes it slightly slower than the double hashing scheme.

to run slightly faster through more streamlining efforts. However, it is not reasonable to assume that our running speeds will approach anything near that of SHA-1, because we call it.

Theoretically, we hoped that our hash function would have a run time roughly on par with a function that hashed a file using SHA-1, prepended that value to the file, and re-hashed it using SHA-1 again for the final output. Our logic was that if you ignored the time required to move the blocks between the Feeder and the Mixer arrays, our function hashes all of the values in a file a little over two times (the extra is added by the e value tags added in the Mixer array). We created a basic script that exhibited this behavior and ran it against the SHA-1 script and our hash function on the same files. This "double hash" function ran roughly as fast as our own, with some variation between specific executions (see figures 2 and 3.)

7 Conclusion

In this paper, we presented and analyzed a new approach to block chaining in a hash function based on block shuffling. Intuitively, shuffling the input blocks makes it harder for the adversary to exploit useful, known block patterns. For example, simply concatenating two messages may cause arbitrary permutations of their blocks, ideally destroying any nice properties (e.g. collisions) of those messages.

We demonstrated that our scheme is at least as collision resistant as its constituent parts and that it is more secure against message extension attacks. Possibly unfortunately, we also show that no finite-state hash function can be provably secure against message extension attacks if it only reads the input once.

Finally, we implemented our scheme and demonstrated that it is practical. As one would expect based on the number of operations, it roughly doubles the time to hash a message as compared to its constituent hash functions. This is significant; however, it is not an unreasonable penalty.

We hoped to demonstrate that our scheme is significantly more secure than existing schemes. We failed to do this; however, we did succeed in proving that it is at least as strong and arguing that it is stronger, as it is not immediately vulnerable to existing attacks against Merkle-Damgård style hash functions.

References

[1] Mihir Bellare and Thomas Ristenpart. Multi-property-preserving hash domain extension: The emd transform, August 4, 2006.

[2] Magnus Daum and Stefan Lucks. Hash collisions (the poisoned message attack).

[3] Jonathan J. Hoch and Adi Shamir. Breaking the ice - finding multicollisions in iterated concatenated and expanded (ice) hash functions. In Matthew J. B. Robshaw, editor, *FSE*, volume 4047 of *Lecture Notes in Computer Science*, pages 179–194. Springer, 2006.

[4] Cecile Malinaud3 Jean-Sebastien Coron, Yevgeniy Dodis and Prashant Puniya. A new design criteria for hash-functions.

[5] Antoine Joux. Multicollisions in iterated hash functions. application to cascaded constructions. In *CRYPTO*, pages 306–316, 2004.

[6] John Kelsey and Tadayoshi Kohno. Herding hash functions and the nostradamus attack.

[7] Werner Schindler Max Gebhardt, Georg Illies. A note on the practical value of single hash collisions for special file formats, 2005.

[8] Ralph C. Merkle. One way hash functions and des. In *CRYPTO '89: Proceedings on Advances in cryptology*, pages 428–446, New York, NY, USA, 1989. Springer-Verlag New York, Inc.

[9] O. Mikle. Practical attacks on digital signatures using md5 message digest. Cryptology ePrint Archive, Report 2004/356, http://eprint.iacr.org/2004/356.

[10] Bart Preneel. Analysis and design of cryptographic hash functions. PhD Thesis. Katholieke Universiteit Leuven, January, 1993.

[11] Ivan Bjerre Damgård. A design principle for hash functions. In *CRYPTO '89: Proceedings on Advances in cryptology*, pages 416–427, New York, NY, USA, 1989. Springer-Verlag New York, Inc.

[12] Xiaoyun Wang and Hongbo Yu. How to break md5 and other hash functions. In *EUROCRYPT*, pages 19–35, 2005.

[13] Andrew C Yao Xiaoyun Wang and Frances Yao. Cryptanalysis on sha-1 hash function. Keynote Speech at CRYPTOGRAPHIC HASH WORKSHOP, 2005.