XSA-strengthening: Strengthening MD5 and Other Iterated Hash Functions Through Variable-length External Message Expansion

[CSD Technical Report 08-894, USC, Los Angeles, California, January 2008.]

William C. Cheng
Dept. of Computer Science
University of Southern California
Los Angeles, California
bill.cheng@usc.edu

Leana Golubchik
Computer Science, EE-Systems, IMSC
University of Southern California
Los Angeles, California
leana@cs.usc.edu

Abstract

In recent years, it has been demonstrated that collisions can be systematically constructed for some popular cryptographic hash algorithms, such as MD5 and SHA-1. Various ways of enhancing these hash functions via *message pre-processing* or *external message expansion* have been proposed to make them resistant to known collision attacks. Message pre-processing/expansion is a way of creating a new hash function from an original one. It has the advantage of being backward-compatible with the original hash function, and therefore, may extend the useful life of the original hash function.

In this paper, we examine a novel approach to message pre-processing/expansion, which we call *eXtremeShrinking ARC4 Strengthening* (or *XSA-strengthening*). *XSA-strengthening* is based on the idea of the self-shrinking generator, the ARC4 cipher, and *MD-strengthening* and can be applied to any *Merkle-Damgård iterated hash function*. *XSA-strengthening* is deterministic, has small space and computational overhead, and can be efficiently implemented. We believe that it can be a useful tool for strengthening Merkle-Damgård iterated hash functions.

Index Terms

Hash functions, MD5, SHA-1, Collisions, External Message expansion, Message Pre-processing

I. Introduction

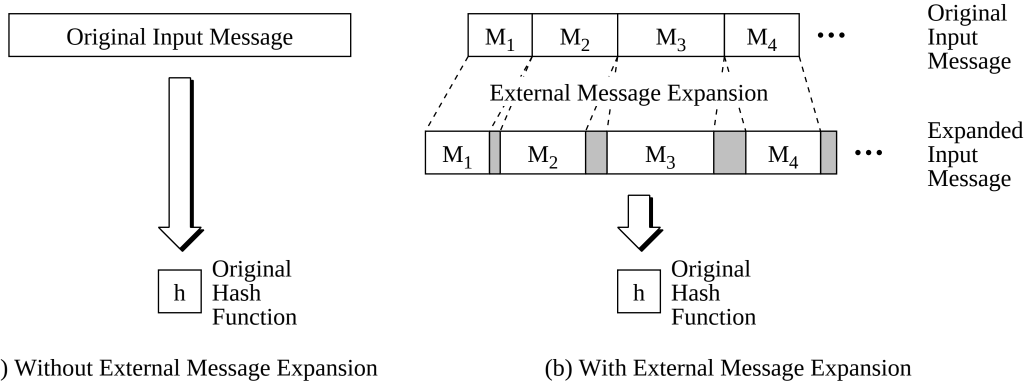
A cryptographic hash function is a vital component in many security products and services such as digital signatures, authentication services, virus checkers, etc. An important property of a cryptographic hash function is *collision resistance*. With collision resistance, it is computationally difficult to find two distinct input strings that hash to the same value. The security of some of these products and services depends on the collision resistance property of the underlying hash functions.

In recent years, it has been demonstrated (e.g., in [1] and [2]) that collisions can be systematically constructed for some popular cryptographic hash algorithms that are based on the *Merkle-Damgård construction*, such as MD5 [3] and SHA-1 [4]. Researcher have been seeking replacements for these “broken” hash functions to be used in future security products and services. One approach is to stay with the Merkle-Damgård construction design principle and design new hash functions that are resistant to current attacks. Another approach is to invent new hash functions that are not based on the Merkle-Damgård construction. A third approach is to continue with Merkle-Damgård construction but also use *external message expansion* (or *message pre-processing*) to create a new hash function from an existing one. With external message expansion, the original hash function is not altered, but the input to the original hash function is modified and/or expanded. Since the original hash function is preserved, external message expansion has the advantage that it is backward-compatible, and therefore, may extend the useful life of the original hash function. External message expansion is the main focus of this paper. (For the rest of this paper, we will use the abbreviation *MD* to stand for *Merkle-Damgård* , except in the context of the MD5 algorithm.)

Please note that the term *message expansion* often refers to expanding a message block *inside* a hash function. Since our goal is to leave the original hash function unaltered, we use the term *external message expansion* to mean performing message expansion *outside* the original hash function.

A. External Message Expansion

In Figure 1(a), we illustrate the approach where the original input message is fed directly to the original hash function to produce a hash value. In Figure 1(b), we depict the *external message expansion* approach, where blocks are inserted into various places of the original input message to create an expanded input message. The expanded input message is fed to the *unaltered* original hash function to produce a hash value. In general, not all inserted blocks are equal in size; furthermore, they do not have to be inserted at regular intervals.



(a) Without External Message Expansion

(b) With External Message Expansion

Fig. 1. Computing a hash value with or without *external message expansion*.

We can view *MD-strengthening* [5], [6] as an approach that uses external message expansion. With MD-strengthening, the length of the input message is encoded in a *Length block* which is appended to the input message. (Some padding bits are inserted between the original input message and the Length block if necessary.) The expanded message is then fed into the iterated hash process. With the use of MD-strengthening, it is difficult to create collisions for messages of different lengths. In a way, MD-strengthening forces an attacker to look for collisions from messages having the same *structure*. It was suggested in [7] that "iterated hash functions should be used only with MD-strengthening".

Our basic idea is to create a message-dependent *variable-length block*, which we call the *SA block*, to be appended to the original message. The expanded message is then fed into an unmodified MD iterated hash function. The SA block is thus inserted between the original message and the MD Length block. The length of the SA block is a function of the original message. We would like the content of the SA block to be hard to control in an attack. In addition, if we can make the *length* of the SA block difficult to control, then MD-strengthening can help to make it even harder to create collisions. Similarly to many previously proposed external message expansion techniques, this technique can be applied to any MD iterated hash functions. We call our technique *ShrinkingARC4-strengthening* (or *SA-strengthening*). To provide additional protection, we present a relatively straight-forward extension to *SA-strengthening* which we call *eXtremeShrinkingARC4-strengthening* (or *XSA-strengthening*). The basic idea of *XSA-strengthening* is to insert additional *variable-length blocks* into the message stream.

In [8], Kauer et al. proposed to expand the input message by using another hash function, in their case, a *keyed hash function*, to compute a *tag* of the input. The *tag* is added at the beginning and the end of the original input message to produced the expanded input message. The expanded input message is fed to the original unmodified hash function. The original input message can also be segmented and the same method can be used with each segment. A drawback of this approach is that its computational overhead can cause loss of performance.

In [9], Szydlo and Yin proposed three methods. One is called *message whitening* where strings of zeroes are inserted into the original input message at predetermined positions. The second method is called *message self interleaving* where the message is divided into blocks and duplicates of these blocks are inserted into the original message. The third method is called *message duplication* where the message is concatenated with itself to produce the expanded message. All these methods can be efficiently implemented. A drawback of these approaches is that, in practice, the expanded message length can be at least 25% more than the input message and this may slowdown performance.

In an unpublished paper, Fortner proposed to append the state variables of a mechanism that permutes the elements of an array in a seemingly random way [10]. Although Fortner's proposal did not use the *ARC4 cipher*, it is similar to our proposal of *ARC4 Hash* described in Section II-A. Many of the parameters in Fortner's proposal, such as the size of the array, how to initialize the array, and initial values of the registers, are left unspecified.

All of the above proposals (and others, such as [11], which consider improvements to hash functions but do not specifically focus on external message expansions) only considered inserting fixed-size blocks into the original input message. With *SA-strengthening*, we use the idea of a *self-shrinking generator* [12] to produce variable-length *ShrunkenARC4 blocks* (or *SA blocks*) to be inserted into the message stream. *To the best of our knowledge, there is no existing work on using message-dependent variable-length external message expansion to strengthen a hash function.* To be efficient in calculating the SA block, we use a modified ARC4 algorithm.

Traditionally, the self-shrinking generator and ARC4 [13] are both used in stream ciphers. We choose them because they have nice security properties and because they are efficient in both space and time.

The remainder of this paper is structured as follows. In Section II, we present *SA-strengthening*. We describe how we modify the ARC4 cipher to be used as a hash function and how to generate a variable-length SA block to expand the input message. Section III presents additional strengthening to our approach called *XSA-strengthening*. In Section IV, we give our security rationale. Section V concludes with a summary and a discussion.

II. SA-STRENGTHENING

Motivated by efficiency, we mainly operate in octets (or equivalently, bytes). If the bit length of the input message is not a multiple of 8 bits, it is padded with zero bits to make the bit length a multiple of 8 bits. We will use the `input[]` array to represent the input message where `input[0]` is the first byte of the input message.

A. ARC4 Hash

We first describe how we modify the ARC4 cipher and make it into a hash function. The internal state for the ARC4 cipher includes an array of 256 bytes denoted by `S[0..255]`. ARC4 also uses two index registers, `i` and `j`, where register `i` steps through the array indices and register `j` indexes seemingly random element of the array.

ARC4 can be divided into 3 algorithms. The *initialization algorithm* initializes the array to an incrementing pattern, and it is depicted as follows:

```
for i from 0 to 255
    S[i] := i
```

The *key scheduling* algorithm uses a key array to permute the state array into a seemingly random pattern. The key scheduling algorithm is depicted in pseudocode as follows, with the key array denoted by `key[0..(n-1)]` where `n` denotes the length of the key in bytes:

```
j := 0
for i from 0 to 255 do
    j := (j + S[i] + key[i mod n]) mod 256
    swap(S[i],S[j])
end for
```

The *output* algorithm selects seemingly random elements of the state array and outputs their values. It is depicted as follows:

```
u := v:= 0
while GeneratingOutput do:
    u := (u + 1) mod 256
    v := (v + S[u]) mod 256
    swap(S[u],S[v])
    output S[(S[u] + S[v]) mod 256]
end while
```

To convert ARC4 into a hash function, we can simply substitute the input message for the key array. Let `n` be the length of the input message in bytes. If the length of the message is less than 256 bytes, line (5) above may use the input message multiple times. Thus, line (5) can be replaced by line (5a) below:

```
( 5a)     j := (j + S[i] + input[i mod n]) mod 256
```

If the length of the message is greater than or equal to 256 bytes, we replace lines (4) and (5) above with lines (4a), (4b), and (5b) below:

```
( 3)  j := 0
( 4a) for x from 0 to n-1 do
( 4b)     i := x mod 256
( 5b)     j := (j + S[i] + input[x]) mod 256
( 6)      swap(S[i],S[j])
( 7)  end for
```

We require that for a non-null message whose length is less than 256 bytes, the input message is repeatedly fed to the original hash function until 256 bytes are sent. This is similar to *message duplication* described in [9]. But unlike [9], we do not restrict the input message to be repeated at most twice. Although it has been shown that feeding the input message multiple times may not increase the strength of the hash function substantially [14], it is done here to ensure that every element of the state array has been moved. We will refer to this as the *Message Self-Repeat* (or *MSR*) requirement.

Another modification we make is to use a slightly different initialization algorithm, in order to make the initial pattern in the state array more difficult to model in an attack. We arbitrarily choose to initialize the state array to the values of the AES S-box [15]. The AES S-box is depicted next:

63	7c	77	7b	f2	6b	6f	c5	30	01	67	2b	fe	d7	ab	76
ca	82	c9	7d	fa	59	47	f0	ad	d4	a2	af	9c	a4	72	c0
b7	fd	93	26	36	3f	f7	cc	34	a5	e5	f1	71	d8	31	15
04	c7	23	c3	18	96	05	9a	07	12	80	e2	eb	27	b2	75
09	83	2c	1a	1b	6e	5a	a0	52	3b	d6	b3	29	e3	2f	84
53	d1	00	ed	20	fc	b1	5b	6a	cb	be	39	4a	4c	58	cf
d0	ef	aa	fb	43	4d	33	85	45	f9	02	7f	50	3c	9f	a8
51	a3	40	8f	92	9d	38	f5	bc	b6	da	21	10	ff	f3	d2
cd	0c	13	ec	5f	97	44	17	c4	a7	7e	3d	64	5d	19	73
60	81	4f	dc	22	2a	90	88	46	ee	b8	14	de	5e	0b	db
e0	32	3a	0a	49	06	24	5c	c2	d3	ac	62	91	95	e4	79
e7	c8	37	6d	8d	d5	4e	a9	6c	56	f4	ea	65	7a	ae	08
ba	78	25	2e	1c	a6	b4	c6	e8	dd	74	1f	4b	bd	8b	8a
70	3e	b5	66	48	03	f6	0e	61	35	57	b9	86	c1	1d	9e
e1	f8	98	11	69	d9	8e	94	9b	1e	87	e9	ce	55	28	df
8c	a1	89	0d	bf	e6	42	68	41	99	2d	0f	b0	54	bb	16

Let `sbox[0..255]` be the array that holds the values of the AES S-box, i.e., `sbox[0]=0x63`, `sbox[1]=0x7c`, . . ., `sbox[255]=0x16`. The updated initialization algorithm is then as follows:

```
( 1)  for i from 0 to 255
( 2a)     S[i] := sbox[i]
```

We now assume the use of the updated initialization and key scheduling algorithm (with (2a), (4a), (4b), (5a), and (5b)). Then, after we have exhausted the input, we can simply output the state array sequentially as the hash value. (We do not need the ARC4 output algorithm here.) The hash value is always 256 bytes in length. We will refer to this algorithm as the *ARC4 hash algorithm* and to the hash it produces as the *ARC4 hash* of the input message.

The ARC4 hash algorithm described here is fast and takes constant (and small) amount of space to execute. But *it is not a good cryptographic hash algorithm*. For instance, if two input messages differ only in the last byte, their ARC4 hashes will differ by at most three bytes. In addition, in the case of very short messages, the corresponding ARC4 hashes can be similar. Yet it may still be useful in strengthening the original hash function, if it is used to expand the input message. (Recall that with MD-strengthening, the length block is a very simple function of the input message. Yet it has utility in guarding against some collision attacks.)

Given the above algorithm, the size of an ARC4 hash is fixed. Below, we show how to make it into a variable-length hash.

B. Shrunken ARC4 Hash

Let `a[0..(n-1)]` be an array of bytes where `n` is an even number. Let the *parity* of a byte be the bit-wise XOR of all the bits in the byte. (To calculate the parity of a byte efficiently, a lookup table can be used.) Let `shrink(a)` be defined by the following pseudocode:

```
i := 0
while i < n do
    if parity(a[i]) = 1
        output a[i+1]
    i := i + 2
end while
```

In the above procedure, we will refer to `a[0]`, `a[2]`, ..., `a[n-2]` as *parity bytes*. If the parity of a parity byte is odd, the following byte is included in the output. Otherwise, the following byte is not included in the output.

If `a` is an ARC4 hash, we will refer to `shrink(a)` as a *shrunken ARC4 hash*. The length of a shrunken ARC4 hash may range from 0 to 128 bytes. If two ARC4 hashes differ in only two bytes, then there is a reasonably high probability that their shrunken hashes are identical. If we use a *parity accumulator*, we may lessen this problem. The *parity accumulator* is a byte corresponding to the byte-XOR of all the previous bytes whose parity has been examined. Its initial value is set to zero. Instead of evaluating the parity of the current byte, we evaluate the byte-XOR value of the current byte and the parity accumulator. Let the variable `pacc` be the parity accumulator; we modify `shrink(a)` to be the following:

```
i := 0
while i < n do
    pacc := pacc XOR a[i]
    if parity(pacc) = 1
        output a[i+1]
    i := i + 2
end while
```

We use $sah(m)$ to denote the *shrunken ARC4 hash* of an input message m. Please note that a *shrunken ARC4 hash* is a function of the input message *only*. It does not depend on the original hash function. It is clear that if two input messages have identical ARC4 hash values, their shrunken ARC4 hash values will also be identical since our algorithm is *deterministic*.

C. SA-strengthening

Let m be an input message. Let $msr(m)$ be equal to m if m is the null message or if the length of m is greater than or equal to 256 bytes. Otherwise, $msr(m)$ is obtained by repeating m as many times as necessary to fill 256 bytes.

For an MD iterated hash function h and an input message m, the *ShrinkingARC4-strengthening* (or *SA-strengthening*) of h is defined as a hash function g with $g(m) = h(msr(m)||sah(m))$, where $||$ denotes concatenation. It should be clear that this approach is suitable for processing *streaming data* [9].

D. Test Vectors

We now give some examples. The *shrunken ARC4 hash* for a null input message is 60 bytes long; it has the following hexstring representation:

```
6bc501d7d4ccf1d8c7121a6ea0b384d1ed4ceffbf9a39db6ffd2ec97175ddc2a
88db065c9579c8ea7a78a6c6dd1fbd8a66030eb911d99455df0d5416
```

Let SAMD5 denote the *SA-strengthened* MD5. Then the SAMD5 hash of a null input message is the MD5 hash of the above shrunken ARC4 hash. Therefore, the output is:

```
765ffaac6fa64bd6f49f9d715f1168e7
```

Let SASHA1 denote the *SA-strengthened* SHA1. Then the SASHA1 hash of a null input message is the SHA1 hash of the above shrunken ARC4 hash. Therefore, the output is:

```
3cf2e441e0e25e014355e845827acfaf99b344d5
```

If the *parity accumulator* is not used, then the *shrunken ARC4 hash* for a null input message is 63 bytes long, and it has the following hexstring representation:

```
6b2bd776d4afcca5f115c7c312e21a3bb3e384fc4ccfef4df97fa38f9df5b621
ff0ceca75d73dceedb3206d3956dea08782ea63e6635b9c1111e55a10de654
```

Suppose now that a 16-byte input message has the following hexstring representation:

```
000102030405060708090a0b0c0d0e0f
```

Since the length of this input message is 16 bytes (which is less than 256 bytes), it must be fed repeatedly (i.e., 16 times) to the underlying hash function. Its *shrunken ARC4 hash* is 66 bytes long with the following hexstring representation:

```
a3ac9f867e6365c98ec76bbc5d82d86fa295e7390629343cc5a1e8684b0c0027
cdce357dbeaee025fc5b4f18791a8ddf53aa7c1913613a241236f53774a81f16
891d
```

Its SAMD5 hash value is:

```
3a4e8f6d7c5adb8dbb7a6bad07d8dd4c
```

Its SASHA1 hash value is:

```
4216c58a620b84ad952e8a5bf37cb63937920840
```

E. Notes on Shrunken ARC4 Hash Statistics

Given a collection of random input messages, a desirable property is that the distribution of the length of a shrunken ARC4 hash has a mean of 64 bytes. We have generated a large number of random input messages of various lengths, and our experiments indicate that this is indeed the case.

F. Parameterization

For *SA-strengthening*, some variations are possible. For example, (1) not using the *parity accumulator*, (2) using a different algorithm to initialize the state array (instead of using the AES S-box), (3) no padding (this includes not using message self-repeat for short messages), (4) always padding the message with bytes of zeroes so that the length of the padded message is a multiple of 256 bytes (not using *message self-repeat* for short messages), (5) always padding first with one non-zero byte (e.g., `0x80`), and then adding as few bytes of zeroes as needed so that the length of the padded message is a multiple of 256 bytes, and (6) instead of outputting the state array sequentially as the ARC4 hash value, use the ARC4 output algorithm to "clock out" some fixed number of bytes. However, we do not recommend any of the above options and suggest that *SA-strengthening* should be used without any parameters. The reasons are as follows.

1) *No parity accumulator.* The use of the parity accumulator extends the state of the SA engine by 8 bits. This increases the complexity of the SA engine and makes the SA engine more difficult to model.
2) *Different initial state.* The AES S-box is arbitrarily chosen as the initial state of the state array in the SA engine. At the time of this writing, there does not appear to be an advantage to using a different initial state.
3) *No padding.* The disadvantage of this approach is that *ARC4 hashes* of short messages with the same prefix can be very similar.
4) *Pad with zeroes to multiple of* 256 *bytes.* The main advantage of this approach is simplicity. However, a disadvantage is that for short messages, padding with zeroes may be less effective in mixing the state array as compared to padding with the message itself.
5) *Pad with* `0x80` *followed by zeroes to multiple of* 256 *bytes.* The advantage of this approach is that the padded message is unique. But, uniqueness is not required (as in the case of MD-strengthening). The disadvantage is that a total of 256 bytes will be added if the length of the original input message is a multiple of 256 bytes. This overhead may not be desirable.
6) *Use the ARC4 output algorithm to generate the ARC4 hash value.* The advantage of using the ARC4 output algorithm is that (a) we can reduce the size of the ARC4 hash to speed up the hash computation, and (b) we can fine-tune the number of bytes in the ARC4 hash. But this can be viewed as a disadvantage since we do not know how many bytes to output. Simply outputting the state array as the ARC4 hash has the advantages of simplicity and that it gives additional structure in the expanded message, since the state array is always a permutation of the values, 0 through 255.

III. XSA-STRENGTHENING

SA-strengthening is similar to *MD-strengthening* in that (a) each method produces a block to be appended to the input message, and (b) the expanded message is fed to the iterated hash function. As proposed in [8] and [9], additional blocks can be inserted at various places of the input message to provide additional strengthening. Similarly, we propose a way of extending SA-strengthening by inserting *variable-length blocks* at fixed positions of the input message, to provide additional protection. We call this *eXtremeShrinkingARC4-strengthening* (or *XSA-strengthening*).

The basic idea is quite simple. For every 256 bytes of the original input message, insert 8 or fewer bytes (and 4 bytes on the average) based on the state of the modified ARC4 cipher. We refer to these added bytes as *noise bytes*. If the length of the input message is not a multiple of 256 bytes, no *noise bytes* are appended at the end. After the whole input message is read, a *shrunken ARC4 hash* is appended at the end. (In general, we can insert up to K noise bytes for every 2^K bytes of the original input message. The following discussion is limited to the case where $K = 8$.)

The *noise bytes* can be obtained by selecting 16 bytes from the state array and applying the shrinking algorithm described in Section II-B with n being 16. The 16 bytes can come from fixed locations of the state array. But we chose to basically use a modified ARC4 output algorithm to "clock out" 16 bytes as follows (with r being 16).

```
(10a) for i from 0 to (r-1) do
(11)      u := (u + 1) mod 256
(12)      v := (v + S[u]) mod 256
(13)      swap(S[u],S[v])
(14)      output S[(S[u] + S[v]) mod 256]
(15a) end for
```

Please note that variables u and v are now globally persistent variables (and no longer local to the output algorithm). Therefore, we need to modify the initialization algorithm as below:

```
( 0)  u := v := pacc := 0
( 1)  for i from 0 to 255
( 2a)     S[i] := sbox[i]
```

Please also note that the *parity accumulator* is a globally persistent variable as well. It carries the XOR'ed values of all examined parity bytes.

A. Taking Care of Short Messages

Since no *noise bytes* will be inserted for an input message whose length is less that 256 bytes, short input messages may be more vulnerable to attacks. Below we describe a way of inserting *noise bytes* into short messages.

The basic idea here is also quite simple, and it only applies to the first 128 bytes of the input message. After reading a total of 2^k bytes from the input, where $k \geq 1$ and $k \leq 7$, insert at most k bytes into the message. (In the general setting described above, $1 \leq k \leq K - 1$.) The modified ARC4 output algorithm described above can be used with r being $2k$.

Please note that if the length of the input message is less than 128 bytes, the input message must be repeated and expanded to conform to the *message self-repeat* requirement described in Section II-A.

Since we do not want to rely on knowing the length of an input message (in order to satisfy the streaming data requirement), we will carry out the operation just described during the processing of the first 128 bytes of input data for all messages.

B. Test Vectors

For a null message, no *noise bytes* can be added. In this case, *XSA-strengthened* MD iterated hash function outputs the same hash value as a corresponding *SA-strengthened* MD iterated hash function.

Given a 16-bytes input message with the following hexstring representation:

```
000102030405060708090a0b0c0d0e0f
```

We tabulate the *noise bytes* and *shrunken noise bytes* below for all values of k. The *shrunken noise bytes* are fed to the original hash function after the 2^k-th bytes of the input message are fed to the original hash function.

k	noise bytes (hexstring)	shrunken noise bytes (hexstring)
1	573b	3b
2	11122183	1283
3	b3578a6c75ce	6c
4	8de9aca7c712d0b8	12
5	e7c73d5a9b46af389270	5a70
6	d633d8a08f34b9f8ddc01e6c	34
7	a41275ee14771d020019af138c97	1297
8	8145a1ab977b17f8c0302ca1dd4d65bb	457bf830

In this case, the *shrunken ARC4 hash* is 59 bytes long with the following hexstring representation:

```
cac3dac89148fca558a16f4ab914b8e43489b2a345282082d83918864ce10299
dda4bd29f2aae89f55c0212bf04d7210f643f8bc61df9b4f966009
```

Thus, the original hash function is fed the following data (i.e., the expanded input message):

```
00013b02031283040506076c08090a0b0c0d0e0f12
000102030405060708090a0b0c0d0e0f5a70
000102030405060708090a0b0c0d0e0f
000102030405060708090a0b0c0d0e0f34
000102030405060708090a0b0c0d0e0f
000102030405060708090a0b0c0d0e0f
000102030405060708090a0b0c0d0e0f
000102030405060708090a0b0c0d0e0f1297
000102030405060708090a0b0c0d0e0f
000102030405060708090a0b0c0d0e0f
000102030405060708090a0b0c0d0e0f
000102030405060708090a0b0c0d0e0f
000102030405060708090a0b0c0d0e0f
000102030405060708090a0b0c0d0e0f
000102030405060708090a0b0c0d0e0f
000102030405060708090a0b0c0d0e0f457bf830
cac3dac89148fca558a16f4ab914b8e43489b2a345282082d83918864ce10299
```

```
dda4bd29f2aae89f55c0212bf04d7210f643f8bc61df9b4f966009
```

Let XSAMD5 denote the *XSA-strengthened* MD5. For the above 16-byte input message, its XSAMD5 hash value is as follows:

```
24b6f1abc59ab1de11999896ff4d2ea2
```

Let XSASHA1 denote the *XSA-strengthened* SHA1. For the above 16-byte input message, its XSASHA1 hash value is as follows:

```
72e87eeff307b8991e7d2b0282d361268dd83b78
```

C. Parameterization

XSA-strengthening only has one parameter, K, which was introduced at the beginning of Section III. By default, K is 8 and it corresponds to the internal table size of the ARC4 engine used. For a long message, excluding for the first 256-byte block and the last block, the average overhead is 4 bytes of inserted data for every 256 bytes of input data (1.6% overhead). The first 256-byte block has an expected overhead of 36 bytes, and the last block has an expected overhead of 64 bytes. For a very short message (less than 256 bytes in length), there is a fixed overhead of 100 bytes on the average. If it is desirable to have smaller overhead for long messages, a larger K can be used. $K < 8$ is not recommended because it will result in output being performed before the internal states have gone through a full permutation cycle.

IV. Security Rationale

With *external message expansion*, an expanded input message is created from the original input message by introducing redundant information into the message stream. The redundant information creates dependencies among bits in the expanded input message. In a way, the redundant information creates *structure* in the expanded input message. This is similar to the way MD-strengthening creates structure in the expanded input message. With *SA-strengthening* and *XSA-strengthening*, more and more structure is introduced into the expanded input message.

In [9], Szydlo and Yin showed that external message expansion (message pre-processing) can mitigate all known collision attacks for the approaches they proposed. We think that the same argument can be made for SA-strengthening and XSA-strengthening, although we have not done the analysis to date (SA-strengthening and XSA-strengthening appear to be more difficult to analyze due to the complexity of the ARC4 cipher and the self-shrinking property of the output.)

Since message-dependent variable-length external message expansion is a new technique, further carefully examination must be performed before it can be used widely.

V. Conclusions

In this paper we presented a novel method for external message expansion in order to create drop-in replacements of MD5 and other iterated hash functions. The original hash functions are unaltered. We called our method *eXtremeShrinkingARC4-strengthening* or *XSA-strengthening*. In *XSA-strengthening*, we modified the *ARC4 stream cipher* to be used as a hash function and a byte stream generator. We then used the idea of a *self-shrinking generator* to generate message-dependent variable-length *noise bytes* to be inserted into the original input message and a message-dependent variable-length block to be appended to the original input message to form an expanded message. We use a *parity accumulator* to create additional dependencies among the inserted data. For short messages, we use *message self-repeat* for additional protection.

XSA-strengthening has many attractive properties:

- it can be applied to any MD iterated hash function
- the underlying hash function is kept unaltered
- it is deterministic and simple to understand
- it has a fixed memory footprint and supports streaming data
- it has small computational overhead
- for a large input message, it expands the input message by a small amount

One disadvantage of XSA-strengthening is that for short input messages (less than 256 bytes in length), it expands the input message by at least 100 bytes, on the average.

XSA-strengthening is not intended to be a replacement of a well-designed cryptographic hash function. However, we think that it can be a practical tool for strengthening and increasing the useful lifetime of an MD iterated hash function. Since XSA-strengthening only expands the input message externally, potentially it may be useful for non-MD iterated hash functions as well.

References

[1] X. Wang and H. Yu, "How to break MD5 and other hash functions," in *Advances in Cryptology, EUROCRYPT'05*, R. Cramer, Ed. Springer-Verlag, 2005, pp. 19–35.

[2] X. Wang, Y. Yin, and H. Yu, "Finding collisions in the full SHA-1," in *Advances in Cryptology, CRYPTO'05*, V. Shoup, Ed. Springer-Verlag, 2005, pp. 18–36.

[3] R. Rivest, *RFC 1321: The MD5 Message-Digest Algorithm*, http://www.ietf.org/rfc/rfc1321.txt.

[4] D. Eastlake and P. Jones, *RFC 3174: US Secure Hash Algorithm 1 (SHA1)*, http://www.ietf.org/rfc/rfc3174.txt.

[5] I. Damgård, "A design principle for hash functions," in *Advances in Cryptology, CRYPTO'89*. Springer-Verlag, 1989, pp. 416–427.

[6] R. C. Merkle, "One way hash functions and des," in *Advances in Cryptology, CRYPTO'89*. Springer-Verlag, 1989, pp. 428–446.

[7] X. Lai and J. Massey, "Hahs functions based on block ciphers," in *Advances in Cryptology, EUROCRYPT'92*. Springer-Verlag, 1992.

[8] N. Kauer, T. Suarez, and Y. Zheng, "Enhancing the MD-strengthening and designing scalable families of one-way hash algorithms," in *2005 Cryptographic Hash Workshop*, Gaithersburg, Maryland, October-November 2005.

[9] M. Szydlo and Y. Yin, "Collision-resistant usage of MD5 and SHA-1 via message preprocessing," in *2005 Cryptographic Hash Workshop*, Gaithersburg, Maryland, October-November 2005.

[10] J. Fortner, *A Method for Pre-Processing Message Digest Output*, (unpublished, see http://csrc.nist.gov/pki/HashWorkshop/2006/program_2006.htm#unaccepted).

[11] J. Coron, Y. Dodis, C. Malinaud, and P. Puniya, "Merkle-damgård revisited : How to construct a hash function," in *Advances in Cryptology, CRYPTO'05*. Springer-Verlag, 2005.

[12] W. Meier and O. Staffelbach, "The self-shrinking generator," in *Advances in Cryptology, EUROCRYPT'94*, A. de Santis, Ed. Springer-Verlag, 1994, pp. 205–214.

[13] K. Kaukonen and R. Thayer, *A Stream Cipher Encryption Algorithm* Arcfour, IETF Internet Draft, http://tools.ietf.org/html/draft-kaukonen-cipher-arcfour-03, 1999.

[14] J. Hoch and A. Shamir, "Breaking the ICE - finding multicollisions in iterated concatenated and expanded (ICE) hash functions," in *Fast Software Encryption 2006*, Graz, Austria, March 2006.

[15] NIST, *Announcing the Advanced Encryption Standard (AES)*, FIPS 197, 2001.