Interim Report

COMP520-18Y (HAM)

Automated Searching for Differential Characteristics in SHA-2

Mitchell Grout

Supervisor(s):

Ryan Ko

Aleksey Ladur

Cameron Brown

Summary

Cryptographic hash functions are a cornerstone of modern cyber security. However, many such functions, including the SHA-2 family of hash functions, are assumed to have three key properties, which many protocols rely on. To help prove the security of the SHA-2 family of functions, we have cryptanalyzed a custom reduced variant of SHA-256 named MAW32, which shares the same internal structure. The purpose of this cryptanalysis is to determine if there are any weaknesses inherent in the SHA-2 family. This project aims to find novel ways of cryptanalyzing SHA-2 like functions, which in turn can be used to help determine how close we are to deprecating the SHA-2 family. The primary result of this project is that by using classic optimization algorithms, specifically genetic algorithms, we are able to find input differences which lead to collisions in our reduced SHA-2 variant with high probability. These techniques are likely extensible to the SHA-2 family of functions, and any other family which shares similar internal structures.

01/06/2018

Contents

Introduction

The SHA-2 family of hash functions are used widely in security to ensure the integrity of files, generate digital signatures, and securely store passwords. This family of functions was originally designed by the National Security Agency in 2001, standardized under FIPS 180. As with all cryptographically secure hash functions, the SHA-2 family of functions must satisfy three key properties:

- Preimage resistance: Given a hash value, it is computationally difficult to find a message which admits that hash value.
- Second preimage resistance: Given a hash value and its corresponding message, it is computationally difficult to find a second distinct message which admits that same hash value
- Collision resistance: It is computationally difficult to find two messages which admit the same hash value

These three conditions are assumed to be true, and are relied upon by many protocols; for example, password-based authentication assumes that the hash function used is one-way, hence passwords are unrecoverable in a short period of time from only their hashes. As such, it is necessary to verify these properties by attempting to violate them.

In this project we have considered a reduced variant of SHA-2, MAW32, and attempted to cryptanalyze it to greater understand the operation of SHA-2. Specifically, we have worked on finding values Δ_0 for which the hash of a message m and $m \oplus \Delta_0$ are likely to be the same, with a high probability. Through the lens of differences, this means we are searching for cases where $H(m) \oplus H(m \oplus \Delta_0) = 0$ with high probability.

Background

There are many techniques used in the field of cryptanalysis. The most prolific technique for the analysis of cryptographic hash functions is differential cryptanalysis, first used in the cryptanalysis of DES[1]. This framework allows cryptanalysts to determine how differences in input values affect differences in output values for the components used in hash functions. This allows the cryptanalyst to more easily determine what pairs of inputs will likely cause a collision to occur in the hash function. In this context, a difference

typically refers to the XOR difference of two integers. To propagate the set of differences $\Delta_1, \dots, \Delta_n$ through the function $f(x_1, \dots, x_n)$, we randomly sample values for $x_1, \dots, x_n$, compute the value of $f(x_1, \dots, x_n) \oplus f(x_1 \oplus \Delta_1, \dots, x_n \oplus \Delta_n)$, and use these results to calculate the frequency of a particular output difference occurring. We refer to this process as the propagation of differences through a function. We may then extend this notion by chaining more than one function together and propagating differences; as the result of one propagation could be zero or more values, the result of propagating through a series of functions is a search tree. We will call a path through this search tree a differential characteristic. If the end of a differential characteristic leads to an output difference of zero, we will also call it a zero trail.

There are three main stages in a differential cryptanalytic attack; finding a start-point, also called an input difference Δ_0, propagating the difference through the hash function and determining how many zero trails exist, and finally finding sufficient conditions for the trails to hold. Both the second and third parts are typically done in conjunction with each other. To propagate a single difference through an entire hash function, we must propagate it through each component present in the hash function. This can be viewed as the composition of a number of functions, so propagating an input difference through a hash function gives us a search tree of differential characteristics. As such, the propagation function may be implemented as a recursive function. Consequently, the problem of propagating a difference through a hash function is in EXPSPACE. To make the problem more tractable, it is necessary to determine which differential characteristics in the tree are not possible; this can be done by searching for sufficient conditions for a characteristic to hold, i.e. what values certain registers must be for us to take the path described by the characteristic every time, and then attempt to find any contradictions present.

The focus of this project is on the first part of a differential cryptanalytic attack, specifically the discovery of good input differences Δ_0. Past research has been ambiguous as to how these values are found; the typical pattern is that is that papers either state a known-good value from other research, or a value is given without explanation. For example, in the cryptanalysis of MD4[2], a value for Δ_0 without full explanation. Similarly, in more recent papers such on the cryptanalysis of SHA-256[3], the value of Δ_0 is simply cited from another paper. To help deal with this lack of public knowledge in the field of

differential cryptanalysis, we have decided to focus on finding good values of Δ_0 specifically.

Design

As this project requires searching through a very large search space in a very short period of time, it is important that we have fine control over memory allocations, data type sizes, alignments, and all other overheads. As such, we opted to use both C and C++ in implementing the relevant tools. The reason that C++ is included is strictly for the STL; we require many different reusable abstract data types, including vectors, queues, sets, and maps, which would be tedious and error-prone to manually implement in C. As such, we accepted any overhead that these may introduce, for the benefit of being able to write our code in confidence.

Due to the sheer size of SHA-2, specifically block size and digest size, it was decided that it was not feasible to test ideas directly on SHA-2, as testing would take too long to complete. As a compromise, a reduced SHA-2 variant was created, called MAW32. The specific details for this algorithm are given in appendix A. This hash function mimics the internal structure of SHA-256, but with a reduced number of registers, reduced register sizes, reduced block sizes, and reduced digest sizes. However, the structure of the compression function is almost identical. The assumption in this project is that techniques that are applicable to MAW32 will be extensible to SHA-256 and all other variants, as a result of the shared internal structure.

Another issue was with regards to propagation of differences; there are currently three common ways to model differences:

1. XOR differences
2. Modular differences
3. Generalized conditions on pairs of bits

The use of generalized conditions on pairs of bits was first described in the cryptanalysis of SHA-1[4]. Although this method is very flexible, it was decided that using XOR differences was the best choice at this stage, with the potential to change to using generalized conditions at a later point in time. The choice for this was twofold: firstly, XOR differences were simple to compute and understand, making implementation straightforward. Secondly, storing XOR differences had a very small memory footprint and improved performance via locality of data. To store a generalized condition on a pair of bits

requires 4 bits, while storing the XOR difference of a single pair of bits requires 1 bit. Even in the case of MAW32, storing generalized conditions affects the locality of data, as storing all conditions requires 16 bytes, which along with all other stored data typically causes cache misses during operation.

During the project, many small utility programs were created to assist in testing. Many of these tools arose as a more natural way to generate the data required by the main propagation program, while others served to validate results and test ideas. There are currently three utility programs which are a part of this project:

1. **maw_diffs:** A utility program which propagates a set of input differences through a given component, and provides detailed information about the output differences, including frequency of occurrences.
2. **trail_gen:** Another utility program which performs a similar functionality to **maw_diffs.** This program is responsible for the creation of memo files that are used by the main program to propagate differences. This helps to take the burden off of the main program, allowing for differential characteristics to be found faster.
3. **hasher:** A program housing a large amount of functionality, this serves as an interface for performing a variety of tasks with hash functions, including computing hashes up to a certain number of rounds, iterating through all possible values in a range, randomly sampling a given range, and applying input differentials to attempt to find collisions.

Progress

Once these decisions were made, we began work on implementing a characteristic-searching function for MAW32. For scalability and simplicity, we opted to perform propagations on each individual component in MAW32. In some papers, authors opted to 'split' the input when propagating through things such as addition, but we felt that for MAW32 the overhead of propagation was not enough to justify this choice. Additionally, due to the small input size for each component, we opted to simply brute-force the solution. In the most extreme case of MAJ, this required 2^{24} computations to perform a propagation, which took a very short period. However, it was obvious that computing then discarding these results was not useful, so we decided on the space-time trade-off of memoization, since the calculations involved with a propagation are

computationally pure. Even with this however, we noticed a large delay in propagation, consistent with many memo misses. Due to this we made an even larger space-time trade-off involving precomputation of all propagations. In total, this requires no more than 200MB of space for MAW32. Unfortunately, this is not extensible; the amount of space required to store all possible SHA-256 propagations is far too large, so a choice must be made at a later date as to how this problem will be solved in a more extensible fashion.

Although we can propagate input differences through MAW32, we unfortunately do not have any good means of producing said differences. At the time, testing was performed by randomly choosing a 64-bit input, which is the block size for MAW32, running it through the propagation functions, and observing the result. However, we quickly noticed that in most cases, a randomly selected input difference did not manage to propagate through to the end. In fact, in most cases, it never worked. According to Mendel et al[3], a good input difference "should span over few steps and only some message words should contain differences."; as such, we modified our input difference generator to set only the last 32-bit portion of our 64-bit input difference leaving the first 32-bit portion zeroed out. Although this seemed to improve performance, it was still far too slow. Again, in Mendel et al[3], it is stated that good input differences should have "large parts of the expanded message having no difference", meaning we may apply a heuristic to our input difference generator. Our heuristic involves taking a candidate input difference, propagating only the message expansion, and determining whether or not the input difference expand to having a difference of zero. An added benefit of this is that we can pre-emptively bailout of propagations where probabilities fall too low. With this in place, we noticed a reasonable speedup in propagations, but a slowdown in choosing input differences.

This lead us to our main focus for the project; how to choose good input differences. In most papers that we read, the explanation for how input differences were chosen was very vague. In the original cryptanalysis of MD4 and RIPEMD[2], the input difference was simply given with no proper explanation given; many papers acknowledge this, such as Schläffer and Oswald[5], stating that the value was likely "found by hand with a great deal of intuition.", and other ad-hoc methods. In other papers such Mendel's recent paper on SHA-256[6], known-good values were used and improved. As such, there was uncertainty in our project in how to properly compute these values. We found that we were able to find values by random sampling, but this was not efficient enough. Even for only 8/16 rounds, random sampling would take approximately one minute per viable input difference found. Although this was improved by parallelization, and will be improved further by

distribution of work, there was a need for an alternative. As such, we considered the classic optimizations algorithm, specifically genetic algorithms.

As with all genetic algorithms, the design depends on the problem. In our case, we opted to model genes as the bit-array representation of an input difference, without the fitness function being the proportion of zero differentials found during propagation. The gene pool was allowed to vary from run-to-run, but was typically set at around 128 genes. Currently three difference techniques are used for evolving the gene pool. The first two are the typical evolution and single-point-crossover techniques. The third technique is titled immigration, which involves introducing new genes randomly into the pool to help prevent the pool from converging to a single solution. This typically gives good results, as evidenced in appendix B, but has a glaring issue; the fitness function chosen does not properly select for the input difference most likely to produce a valid collision. As an example, the input difference $\Delta_0 = 0x00000000025023f0$ has a fitness of ~ 0.0493, while the input difference $\Delta_0 = 0x0000000008008880$ has a fitness of ~ 0.0631, yet the first input difference is capable of producing collisions in MAW32 at a much faster rate than the second. An example of this can be seen in appendix C. Note that the second input difference was unable to produce any collisions within the 1 second timeframe provided, and was similarly unable to provide collisions within 30 seconds of sampling.

Planning

So far, we have followed our goals as set out in our proposal quite closely. Unfortunately, we have spent more time working on MAW32 than we had initially planned however. This was due to unforeseen issues with initially implementing everything in pure C, as well as some misinterpretations of processes outlined in other papers. Additionally, the order in which some of our planned goals were achieved has been changed, where the simpler goals have been implemented first. Currently there are four major goals remaining in our project, which we anticipate being able to finish by the end of June. Each of these goals constitutes an important step in being able to fully generalize the current results of our research.

The first and most important goal is to deal with the sheer size of the search tree generated while propagating an input difference. As the problem of propagation is in EXPSPACE, the search tree will be very large, and will grow even larger when we attempt to propagate through later rounds. To deal with this, we must determine which branches

can and cannot be taken at any step in the propagation. By doing this we can eliminate branches which are contradictory, hence reducing the overall size of the tree. Consequently, this will hopefully make our fitness function more representative of what we are trying to optimize, which is the number of valid zero trails.

The second goal is to consider implementing alternate optimization algorithms, specifically simulated annealing, and determining how it fares against the currently implemented genetic algorithm. If this approach also works, it may even be preferable to using a genetic algorithm, as we no longer need to build an entire gene pool, which currently takes a very long time.

The final goal is to extend our existing work up to SHA-2. We predict this to be quite straightforward, since the similar internal structure of MAW32 should allow us to simply change datatypes and definitions without much thought. However, some work will need to be done to deal with the amount of memory used, ensuring propagations run in a tractable amount of time, and many other currently unforeseen issues with extension.

Conclusion

Our project has so far produced promising results, indicating that the use of classic optimization algorithms may in fact be an avenue for creating and improving input differences for MAW32. Current research indicates that this is a much better approach to purely random selection. We hypothesise that this is extensible to SHA-2 as a consequence of MAW32 sharing almost identical internal structure to SHA-256. If this is the case, then it is likely that we will be able to improve currently known good values for Δ_0 in SHA-256 to help improve the current cryptanalysis of SHA-2 even further.

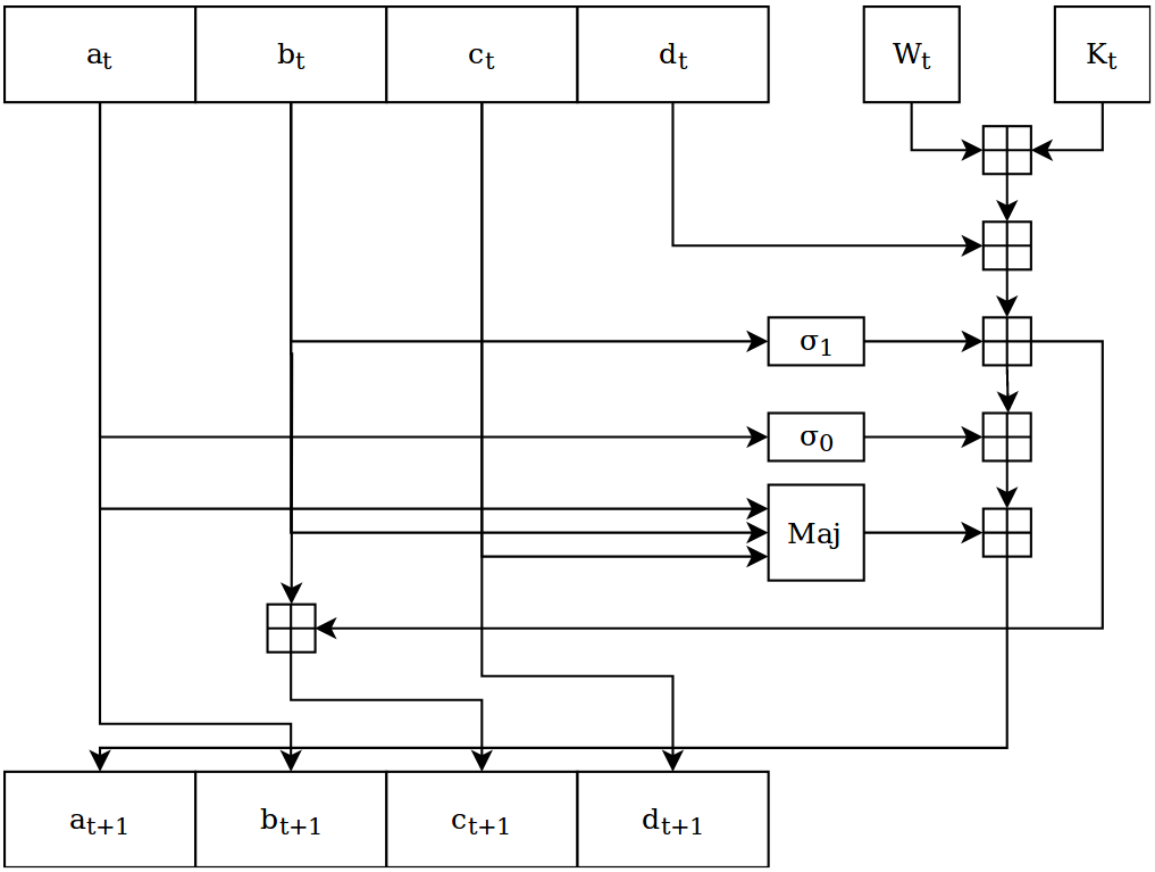
References

1. Biham, E. and A. Shamir, *Differential Cryptanalysis of the Data Encryption Standard*. 1993: Springer-Verlag New York.
2. Wang, X., et al. *Cryptanalysis of the Hash Functions MD4 and RIPEMD*. 2005. Berlin, Heidelberg: Springer Berlin Heidelberg.
3. Mendel, F., T. Nad, and M. Schläffer. *Finding SHA-2 Characteristics: Searching through a Minefield of Contradictions*. 2011. Berlin, Heidelberg: Springer Berlin Heidelberg.
4. De Cannière, C. and C. Rechberger. *Finding SHA-1 Characteristics: General Results and Applications*. 2006. Berlin, Heidelberg: Springer Berlin Heidelberg.
5. Schläffer, M. and E. Oswald. *Searching for Differential Paths in MD4*. 2006. Berlin, Heidelberg: Springer Berlin Heidelberg.
6. Mendel, F., T. Nad, and M. Schläffer. *Improving Local Collisions: New Attacks on Reduced SHA-256*. 2013. Berlin, Heidelberg: Springer Berlin Heidelberg.

Appendix

A. A description of MAW32:

MAW32 uses 64-bit blocks and produces a 32-bit digest. It uses the usual Merkle-Dåmgard construction using a compression function almost identical to SHA-2. The compression function is called 16 times when hashing an input. The 32-bit IV is given by the fractional expansion of π, available from the sequence A062964. The values of the key schedule K_t are given by the fractional expansion of e, available from the sequence A170873. The internals of the compression function are given below:



With components given by:

$$\sigma_0(x) = x^{\gg 2} \oplus x^{\gg 3} \oplus x^{\gg 5}$$
$$\sigma_1(x) = x^{\gg 1} \oplus x^{\gg 4} \oplus x \gg$$
$$Maj(x, y, z) = (x \wedge y) \oplus (x \wedge z) \oplus (y \wedge z)$$
$$x \boxplus y = (x + y) \bmod 2^8$$

The message expansion is given by $W_t = \begin{cases} M_t & : 0 < t \leq 8 \\ \sigma_0(W_{t-3}) + W_{t-4} + \sigma_1(W_{t-8}) & : 8 < t \leq 16 \end{cases}$

The registers a_t, b_t, c_t, d_t are 8-bits wide, and the default values are $(a_0, b_0, c_0, d_0) = IV$

B. Example results from the genetic algorithm

```
mjg44@shae ~/COMP520 λ ./maw_trail
[11:59:27] Initializing...
[11:59:27] Threads: 4
[11:59:27] Rounds: 8/16
[11:59:27] Threshold probability: 2^-3.000000
[11:59:27] Pool size: 128
[11:59:27] Loaded key memos from /Scratch/key-file-3.000000.bin
[11:59:27] Loaded add memos from /Scratch/add-file-3.000000.bin
[11:59:29] Loaded maj memos from /Scratch/maj-file-3.000000.bin
[11:59:29] Done!

[11:59:34] (Fingerprint: 00000000e4289448, Fitness: 0.003461)
[11:59:44] (Fingerprint: 0000000068a0e800, Fitness: 0.026295)
[12:00:10] (Fingerprint: 000000002c64eca4, Fitness: 0.027830)
…
[13:18:58] Population 28 bred.
[13:18:58] Current best:
[13:18:58] (Fingerprint: 0000000038208800, Fitness: 0.020656)
[13:20:28] Population 29 bred.
[13:20:28] Current best:
[13:20:28] (Fingerprint: 0000000038209800, Fitness: 0.018489)
[13:22:03] Population 30 bred.
[13:22:03] Current best:
[13:22:03] (Fingerprint: 0000000038209820, Fitness: 0.031727)
```

C. 8-round collisions in MAW32 using $\Delta_0 = 0\text{x}00000000025023\text{f}0$

```
mjg44@shae ~/COMP520 λ timeout 1 ./hasher diff maw32 8 0x00000000025023f0
91695960982b9f27 - 916959609a7bbcd7 => 63346e9d
8ec2dd1c5930f793 - 8ec2dd1c5b60d463 => 13756b08
a7bf717a3d332311 - a7bf717a3f6300e1 => 5f7c83b2
c0444b654388d42e - c0444b6541d8f7de => d8071af7
e50141a43c9cd717 - e50141a43eccf4e7 => 6a54b630
b554a5951471f3d8 - b554a5951621d028 => b876aca7
de2cb31f9f395c94 - de2cb31f9d697f64 => 44140c02
6c2441b00543336b - 6c2441b00713109b => fc601ad3
ee341e7b3117339d - ee341e7b3347106d => bf297102
656cc1db84347b93 - 656cc1db86645863 => 0145f965
b682e07430bd3318 - b682e07432ed10e8 => d9b57334
81fca88f17c1f468 - 81fca88f1591d798 => 75f5261c
96a7ec0885cff365 - 96a7ec08879fd095 => bffd463e
b75952fa51e64765 - b75952fa53b66495 => e3bf962e
c353f4daed563b57 - c353f4daef0618a7 => 95566baa
6c44c46dc1947f96 - 6c44c46dc3c45c66 => 4bbcf61a
4f0e0ba47c3cdb94 - 4f0e0ba47e6cf864 => 8cee0e25
56998f32d2164890 - 56998f32d0466b60 => a3b7ecab
bc96467ec41e031d - bc96467ec64e20ed => cf4056cc
3c67a38e02820420 - 3c67a38e00d227d0 => 3f8f32fc
6463f003ed22aba0 - 6463f003ef728850 => 7b7eb0b6
f41bcc4247184816 - f41bcc4245486be6 => 91582bb2
d9058e6c1ce8cb69 - d9058e6c1eb8e899 => 92ee9163
f71d5daf503dd791 - f71d5daf526df461 => 45ce302e
a3acd8e8e0ff6357 - a3acd8e8e2af40a7 => ce036fb0
c851335c80adaba7 - c851335c82fd8857 => f184c8f5
43b148c76fb65099 - 43b148c76de67369 => 5fcbb8cc
8dcbb9ebc1ce1fed - 8dcbb9ebc39e3c1d => f08240bd
```