Finding Partial Hash Collisions by Brute Force Parallel Programming

Vincent Chiriaco  
Department of Computer Science  
University of North Alabama  
Florence, AL 35632, U.S.A.  
Email: vchiriaco@una.edu

Aubrey Franzen  
Department of Computer Science  
Northern Kentucky University  
Highland Heights, KY 41099, U.S.A.  
Email: franzena3@mymail.nku.edu

Rebecca Thayil  
Department of Physics  
Bryn Mawr College  
Bryn Mawr, PA 19010, U.S.A.  
Email: rebecca.thayil@gmail.com

Xiaowen Zhang  
Department of Computer Science  
College of Staten Island, CUNY  
Staten Island, NY 11314, U.S.A.  
Email: xiaowen.zhang@csi.cuny.edu

Corresponding Author

Abstract—A hash function hashes a longer message of arbitrary length into a much shorter bit string of fixed length, called a hash. Inevitably, there will be a lot of different messages being hashed to the same or similar hash. We call this a hash collision or a partial hash collision. By utilizing multiple processors from the CUNY High Performance Computing Center’s clusters, we can locate partial collisions for the hash functions MD5 and SHA1 by brute force parallel programming in C with MPI library. The brute force method of finding a second preimage collision entails systematically computing all of the permutations, hashes, and Hamming distances of the target preimage. We explore varying size target strings and the number of processors allocation to examine the effect these variables have on finding partial collisions. The results show that for the same message space the search time for the partial collisions is roughly halved for each doubling of the number of processors; the longer the message is the better partial collisions are produced.1

Keywords—Partial hash collision; brute force; MD5; SHA1; high performance computing; parallel programming, MPI.

I. INTRODUCTION

A cryptographic hash function is a cryptographer’s Swiss Army knife, which has various applications in information security and other areas. For digital signatures, a hash function is required to reduce a long document to a short, fixed-length hash. We then use the private key to sign (encrypt) the hash. Hash functions are used in network security protocols, random number generators, information integrity checking, secret sharing schemes [2], stronger hash functions creating [1], etc. This research explores brute force methods of finding partial collisions as second preimages of target strings with the cryptographic hash functions MD5 (Message Digest) and SHA1 (Secure Hash Algorithm). Both hash functions pre-process a message of arbitrary length into multiple 512-bit blocks, and then iteratively compress each block by using the Merkle-Damgard construction. MD5 generates a 128-bit hash while SHA1 generates a 160-bit hash.

Other than brute force, there is no known better algorithms for finding second preimage collisions for hash functions [4]. A collision happens when two different inputs share the same hash; whereas, a partial or almost-collision occurs when their hashes are similar. This similarity is measured by Hamming distance. When the Hamming distance approaches zero, the partial collision becomes a collision. This research focuses on searching for the best five partial collisions for MD5 and SHA1 with given target strings.

A. Research Goals and Experiment Environment

We design a brute force collision algorithm that uses multiple processors and parallel programming to test for hash speed and the second preimage resistance property by searching for best partial collisions in which the hash of the second message has the smallest Hamming distance with the hash of the target message. The hash speed is tested by execution time of running the same parallel program with a different number of processors. The second preimage resistance property is tested by choosing different sizes of the target strings. The longer the target string, the greater the chances are for finding better partial collisions. We implement two versions of the collision algorithm to test time effectiveness for using multiple processors. Version 1 takes in a “target” character string and hashes it to get its hash. We then permute the target string, hash each unique permutation, and compare its hash against the target’s hash for the best partial collisions. Version 2 is similar to version 1, except it takes a hex string (a bit string) as the “target.” Version 2 hashes every increment of the starting hex string until the entire message space is traversed. Both versions work for MD5 and SHA1.

Our parallel programs are written in C with MPI (Message Passing Interface) library. We used the source code of

1A two-page extended abstract of this paper was published in Proceedings of the 37th IEEE Sarnoff Symposium 2016, Newark, NJ, September 19-21, 2016.
MD5 by Ronald Rivest\(^2\) and SHA-1 by David Ireland\(^3\). The programs are first compiled and run locally on a PC using Open-MPI library. Then they are uploaded and executed on the CUNY High Performance Computing Center’s cluster called Penzias. Named after Arno Penzias, a Nobel Laureate in Physics and a CUNY alumnus, Penzias is a 1,152 core cluster with 4 gigabytes of memory per core. Penzias allows applications requiring up to 128 cores.

The rest of the paper is organized as follows. In Section II, we briefly introduce hash function properties and the hash functions MD5 and SHA1. In Section III, we define a data structure called TUPLE. We describe our collision algorithm in two versions, other utility algorithms, and some important remarks on designing and implementation. We show experimental results of found partial collisions and time vs. number of processors performances on character string and hexadecimal bit-string targets for both MD5 and SHA1 in Section IV. We conclude the paper in Section V.

II. CRYPTOGRAPHIC HASH FUNCTIONS

A hash function [5], [11] compresses a message of arbitrary length into a short, fixed-length output string. This fixed-length string is called a message digest, hash, hash value, or fingerprint. Widely used hash functions are MD5 and SHA1; these iterative hash functions are based on the Merkle-Damgard construction.

A. Hash Function Properties and Merkle-Damgard Construction

Let \( H \) be a hash function. \( h = H(x) \) means that \( h \) is the hash of an input \( x \). Hash function \( H \) needs to satisfy three properties: preimage, second preimage, and collision resistances. Preimage resistance states that given hash \( h = H(x) \) for input message \( x \), it is infeasible to find the preimage \( x \). This property also makes \( H \) a one-way function. Second preimage resistance says that given hash \( h = H(x) \) and input message \( x \), it is infeasible to find a second input message \( x' \neq x \) such that \( H(x') = h \). Collision resistance states that it is infeasible to find two distinct inputs messages \( x \) and \( x' \), such that \( H(x) = H(x') \). Collision resistance is the strongest property. A hash function that is collision resistant is also second preimage resistant; second preimage resistance implies preimage resistance [4].

Most iterative hash functions are based on the Merkle-Damgard construction, and the soundness of the construction was proved independently by Ralph Merkle [6] and Ivan Damgard [3] in 1989. The construction iteratively calls an underlying compression function \( C \), which compresses a longer fixed-length input of \( s \)-bit to a shorter fixed-length output of \( t \)-bit, i.e., \( C: \{0, 1\}^{s+t} \rightarrow \{0, 1\}^t \), where \( s > t > 0 \). After padding, any message \( M \) of arbitrary length can be broken into multiple \( s \)-bit blocks, \( s \) is block size, i.e., \( M = m_1\|m_2\|\ldots\|m_n \) where \( \| \) represents concatenation. These message blocks are compressed in succession by the compression function \( C \), see Figure 1. We have \( h_i = C(h_{i-1}, m_i) \), i.e., \( C \) takes a message block \( m_i \) and an intermediate hash \( h_{i-1} \) from the previous block as input, and then compresses the \((s + t)\) bit into a \( t \) bit output as a new intermediate hash \( h_i \), where \( 1 \leq i \leq n \). The first block needs to use \( h_0 = IV \) as the initial value, and the last intermediate hash \( h_n = h \) is the final hash of the entire message \( M \). We can write the whole process as \( H(IV, M) = h \), or \( H(M) = h \) for simplicity.

B. MD5 and SHA1

The MD5 hash function was designed by Ronald Rivest of MIT in 1992. It is an iterative hash function based on Merkle-Damgard construction. An arbitrary length input message \( M \) is broken into multiple fixed-length blocks; the block size \( s \) equals 512 bit. The last block is called the padding block. Padding starts with 1 followed by as many 0’s as needed. The last 64 bit are kept for representing message length. The hash size \( t \) is 128 bit.

MD5 operates on a 128-bit state, divided into four 32-bit words \( A, B, C, \) and \( D \).\(^4\) The compression function \( C \) uses each 512-bit message block in turn to modify the state. The processing consists of four rounds; each round goes through 16 similar operations based on four non-linear functions, which consists of bitwise XOR, AND, OR, NOT, SHIFT, and modular addition. The final 128-bit state left in \( A, B, C, \) and \( D \) is the hash of the entire message. It is \( h = H(M) \), and \( h \) is represented as follows,

\[
h = A||B||C||D. \tag{1}
\]

SHA1, designed by NIST (National Institute of Standards and Technology) in 1995, is constructed also based on Merkle-Damgard construction. Much like MD5, its block size is 512-bit, same padding method as MD5, but hash size is 160-bit. SHA1 operates on a 160-bit state, divided into five 32-bit words \( A, B, C, D, \) and \( E \). SHA1 also uses similar non-linear functions but with more rounds and longer hashes for increased security. The final 160-bit state in \( A, B, C, D, \) and \( E \) is the hash, \( h = H(M) \); it is represented as follows,

\[
h = A||B||C||D||E. \tag{2}
\]

\(^2\)http://people.csail.mit.edu/rivest/Md5.c
\(^4\)https://en.wikipedia.org/wiki/MD5
III. ALGORITHMS AND DATA STRUCTURES

A. TUPLE Structure

We define a TUPLE structure of three fields that holds three pieces of information: a string, its hash, and Hamming distance to the target hash. This structure is needed for our partial collision search purpose. The three fields are:

1) Message, an unsigned char array whose purpose is to hold the message to be hashed. We do not intend to hash messages that are longer than 30 characters, so we initialize the size of the array as 30.

2) Hash, an unsigned char array whose purpose is to hold the message’s hash. Since an MD5 hash is exactly 128 bits long, we initialized this array to have a size of 16. 16 unsigned chars are equivalent to 128 bits. For SHA1, the size of this unsigned array is 20; it is equivalent to 160 bits.

3) Distance, an unsigned char whose purpose is to hold the Hamming distance of this particular tuple message’s hash and a target message’s hash.

B. Two Versions of Collision Algorithm

The two versions of the collision algorithm invoke functions from the source code of MD5 and SHA1 to hash all character string permutations and all hex bit-string increments. They then search for the top five partial colliding strings with the target. The pseudo-code algorithms for the character string and hex bit-string partial collision search are shown in Algorithm 1 and Algorithm 2.

Algorithm 1: Partial collision search for a target character string

Data: A target character string $s_0$
Result: Array of top five character strings with best Hamming distance

Declare & initialize ... // same as in Algorithm 1
for each processor do
  Declare dynamic TUPLE array: topA
  Initialize distances in topA to 128 (160) for MD5 (SHA1)
  Call getOffset function for each processor
  for each increment $s_1, s_2, \ldots$ do
    $h_i = H(s_i)$
    if $dist(h_0, h_i) < \text{any distance in topA}$ then
      update topA
  end
  Root processor calls MPI_Gather
  ... // same as in Algorithm 1
end

Algorithm 2: Partial collision search for a target hex bit-string

Data: A target hex string $s_0$
Result: Array of top five hex strings with best Hamming distance

Declare & initialize ... // same as in Algorithm 1
for each processor do
  Declare dynamic TUPLE array: topA
  Initialize distances in topA to 128 (160) for MD5 (SHA1)
  Call getOffset function for each processor
  for each increment $s_1, s_2, \ldots$ do
    $h_i = H(s_i)$
    if $dist(h_0, h_i) < \text{any distance in topA}$ then
      update topA
  end
  Root processor calls MPI_Gather
  ... // same as in Algorithm 1
end

Algorithm 3: Hamming distance

D. Remarks on Algorithm Design and Implementation

During the implementation and debugging phases, we were faced with some challenges and trade-offs.

1) MPI_Gather only accepts MPI types as parameters, and creating an MPI customized data type TUPLE is highly cumbersome. In addition, the communication cost for sending such a complicated TUPLE type would be higher. So we decide to go with MPI data types for the MPI_Gather function.

2) In order to use the MPI_UNSIGNED_CHAR type for MPI_Gather, we concatenate the five strings, their hashes, and Hamming distances in topA to a long binary string.

---

\(^5\)https://en.wikipedia.org/wiki/Hamming_distance
\(^6\)http://www.geeksforgeeks.org/lexicographic-permutations-of-string/
After MPI_Gather, the root processor parses and separates the five strings along with their corresponding hashes and Hamming distances from all worker processors. However, this method of gathering endures higher communication costs. Additionally the root processor does extra work to parse the gathered strings.

3) The root processor only gathers message strings in topA arrays of worker processors after a certain number of permutations/increments in each round. The root processor then re-hashes these gathered strings, gets their Hamming distances, sorts them, and finally puts the best five tuples in its topA array. It then repeats round by round until all permutations/increments are handled. The drawback is the much re-hashing at the root processor and multiple MPI_Gather calls.

4) Instead of cycling through rounds, we design a load balancing mechanism and let all processors hash and record their top five tuples until all the permutations are handled. Only when the hashing is complete, the root gathers the top five messages from the workers. This way we only call the MPI_Gather once.

IV. EXPERIMENTAL RESULTS

To measure the performances, we have two variables to change: the number of processors and the size of target string. The number of processors is a power of two, up to 64, i.e., 2, 4, 8, 16, 32, 64. For the character string, the target is 7 to 13 unique characters; this guarantees non-repeating permutations and load balance. For the hex bit-string, the target is set from 3 to 10 bytes of hex ff. The execution time is taken from the root (processor-0); it runs the longest because the root does the extra works of gathering, re-hashing, and outputting the results. Due to space limit, we can only show some of results.

A. MD5 Results

1) Hex Bit-String as Target: When the target is a hex bit-string, we set the target as the four byte hex string: 0xffffffff. The output file contains the best five tuples` Hamming distance from target hash, hex string, and its hash. The file also contains the root processor elapsed time. In the example of Table I, the number of processors is 4. We run the same example with 1, 2, 4, and 8 processors; the time performance is shown in Figure 2.

2) Character String as Target: The smallest Hamming distances between found strings of 7, 8, 9 characters and the target string “abcdefg,” “abcedfgh” and “abdefghii” are around 40, 30, and 9, and the message spaces are 7!, 8! and 9!, respectively. When the 10 bytes character string “abdefghij” is the target, and running the program with different number of processors, the performance of time versus number of processors is shown in Figure 3.

B. SHA1 Results

1) Hex Bit-String as Target: The example result file shown in Table II is a variation of standard top five tuple. Instead it collects the top 4 hex strings, 1 from each processor (total 4 processors used). The information contained in the file includes processor ID, the 4 byte hex bit-string message, its corresponding hash, and Hamming distance. This file also gives the offset of the message subspace, which determines the number of hex strings to be hashed for each processor. The work load is balanced for each processor. Below the offset is the target hex string and its corresponding hash. The smallest Hamming distances for 3, 4, and 5 bytes, all ff target bit-strings hashes is around 40. The trend of execution time is a gradual decrease by about half for each increase in the number of processors, see Figure 4.

2) Character String as Target: Here we summarize the results from hashing permutations of target strings with lengths from 2 to 16, i.e., these targets: ab, abc, abcd, abcde, . . . , abcdefghijklmnop. Figure 5 shows the target length versus smallest Hamming distances of permutation’s hashes against target’s hash. We can see that the longer the target/permutation length is the smaller the Hamming distance, i.e., the better partial collisions exist.

The Tables III and IV are the results of using 16 processors on Penzias to SHA1 hash permutations of targets “abcdefgijklm” (length = 13), “abcdefgijklmn” (length = 14) and “abcdefgijklmno” (length = 15). The smallest Hamming distances are 42, 40, and 36, respectively. The time elapsed in seconds for finding these partial collisions

---

**Table I: MD5 hex string partial collisions, target 0xffffffff**

<table>
<thead>
<tr>
<th>Ham. Dist.</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>30</td>
<td>hex string: 0x7549bd34 Hash: a4de0a400de14b958d05447f1ed7057</td>
</tr>
<tr>
<td>31</td>
<td>hex string: 758f01d4b9eb232989a5c063fdcb7840 Hash: a56e2653aaed73590703c63f9dee4609</td>
</tr>
<tr>
<td>31</td>
<td>hex string: 0x62768a61 Hash: bf4f04599e5b5850fa5e8f5b1cf6997</td>
</tr>
<tr>
<td>31</td>
<td>Hex string: 012cd26a9d07f0d50279421f8d6d49 Processor-0 elapsed time: 534,952,093</td>
</tr>
</tbody>
</table>

---

**Figure 2: MD5 hex string performance: time vs. no. of processors**
Table II: SHA1 hex string partial collisions, target 0xffffffff

<table>
<thead>
<tr>
<th>Proc.</th>
<th>Hash string</th>
<th>Hamming distance</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>hex string: 0xc4d2449c Hash: cabc2124a8d723588b082d8b38cf4271cb2712702</td>
<td>42</td>
</tr>
<tr>
<td>0</td>
<td>hex string: 0x03d917ce, Hash: dd37452c65d83438b382603f3ad3879fa2a6382</td>
<td>43</td>
</tr>
<tr>
<td>1</td>
<td>hex string: 0x4affc445, Hash: 983664742ce3567d7c2931e5739d719627104a</td>
<td>43</td>
</tr>
<tr>
<td>2</td>
<td>hex string: 0x9abcf1fd7, Hash: d9b87627a793085c350360226f3b7089c7f268</td>
<td>42</td>
</tr>
</tbody>
</table>

target hex string: 0xffffffff

Hash: d9b87627a793085c350360226f3b7089c7f268
offset of each message space: 0x40000000
processor-0 elapsed time: 522.446491

are 497, 6583 and 104835, respectively.

C. Remarks on Results

Figure 3 shows the performance of time vs. number of processors when MD5 is the hash function and a 10 byte long character string is the target. The overall trend of the curve is a gradual decrease in time by about half for every doubling of number of processors. There is a slight rise when the number of processors reaches 8, so the overhead associated for MPI obviously did not prove efficient for this case. The time did decrease at 16 processors; it was about half of the time collected from 4 processors.

All other results in both MD5 and SHA1 cases, as displayed in Figure 2 for MD5 hex string and Figure 4 for SHA1 hex string, show a consistent decrease in time elapsed for each increase in the number of processors. Roughly, the time is halved for each increase in the number of processors.

V. CONCLUSIONS AND FUTURE RESEARCH

Two collision algorithms are designed and implemented to test the second preimage property of cryptographic hash functions using the brute force method to find partial hash collisions. Each of the parallel programs has functionality to assess the processing speed of each processor. The hash functions used to conduct this research are MD5 and SHA1. Partial collisions are measured by Hamming distances; the smaller the better. Our partial collision search algorithms deal with both character strings and hexadecimal bit-strings.

One important lesson we learned is that load balance is critical in parallel programming. We devise efficient ways to achieve it. The target character strings consists of the left portion of English alphabet in ascending order (this guarantees no duplicated letters in the target). For balancing workload, each processor takes permutations where their counter modulus total number of processors is equal to its processor ID. This guarantees that each worker processor handles an equal subset of permuted strings. The load balancing is relatively easier to achieve for the target hexadecimal bit-string. The total message space is $2^s$; $s$ is the string length in bits. Since the total number of processors $n_p$ is always a power of 2, each processor handles $2^n/n_p$ hexadecimal strings (increments).

Some of the findings include that the smallest Hamming distances between found regular character strings of 7, 8, 9 and the target character string “abcdefg,” “abcedfgh,” and
There is the Marc Stevens [10] single-block collision attack on MD5, and the Wang et al. [12] method for finding full-collisions in SHA1. This method is where input messages must satisfy a given set of bit conditions for each round of hashing. This method leads to finding full collisions at a faster speed compared to a brute force approach.

We can also employ different hash functions. There are several versions of Message Digest and Secure Hashing Algorithm. There are also separate families of cryptographic hashing functions such as the RIPEMD. This particular function takes attributes from versions of both MD and SHA families.

ACKNOWLEDGMENTS

This work is supported, in part, by the National Science Foundation and Department of Defense’s Research Experience for Undergraduates Site for CUNY-College of Staten Island. We want to thank CUNY HPC Center for providing MPI, parallel programming training, and their facilities.

REFERENCES


Table III: SHA1 character string partial collisions with 16 processors and target lengths 13 and 14

<table>
<thead>
<tr>
<th>Ham. Dist.</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>string: abcdefghijklm Hash: 4b98d26572724af655b8a25f4d203c15e7c9c string: lejdakmhbcg</td>
</tr>
<tr>
<td>12</td>
<td>string: 59a9b4f14d654e047eb20b53d507480df7c96 string: bkjaghlidmicd</td>
</tr>
<tr>
<td>42</td>
<td>string: 639d9266252d3ffcevf53e66f2d6460a635af94 string: ambelkjgifgcd</td>
</tr>
<tr>
<td>43</td>
<td>string: 59d412d69f3f2813b0e05ecaf07034db48c9dc7c8c string: gafbjdikemiejcl</td>
</tr>
<tr>
<td>43</td>
<td>Hash: 408b9e365f6f204bc80436deb5671d203e10d99eca</td>
</tr>
<tr>
<td>processor-0 elapsed time: 497.124382</td>
<td></td>
</tr>
</tbody>
</table>

Table IV: SHA1 character string partial collisions with 16 processors and target length 15

<table>
<thead>
<tr>
<th>Ham. Dist.</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>string: abcdefghijklmm Hash: 857e5f403abe72d5b8a2708821ee33cd0bbece string: njmabiceokfdhgl</td>
</tr>
<tr>
<td>36</td>
<td>string: 8572c14b441eae70f3438ae7989a1ac56d90b394 string: neajkdbcimglfh</td>
</tr>
<tr>
<td>37</td>
<td>string: 35e7c4a440169e7ad66a8b6356001fe23e90aad6 string: lejdakmfibhcg</td>
</tr>
<tr>
<td>41</td>
<td>Hash: 1742d75c97956c7928a0748121a33cc3c8e</td>
</tr>
<tr>
<td>processor-0 elapsed time: 6583.194296</td>
<td></td>
</tr>
</tbody>
</table>

“abcdefgfh” are around 40 for 7, and 30 for 8 and 9 character strings for MD5. For hexadecimal bit-strings of 4 bytes, the message space is $2^{32}$, the smallest Hamming distances from the target bit-string 0xffffffff is around 30 for MD5, 40 for SHA1. The trend of execution time is a gradual decrease by about half for each increase in the number of processors (power of two).

In the future we will explore more efficient methods in searching for partial and full hash collisions. For instance, the birthday attack is a much more efficient brute force method. The trade-off is that the birthday attack needs more memory, which increases the space complexity. There are also smart methods of exploiting bit conditions. For example, there is the Marc Stevens [10] single-block collision attack

Table: SHA1 character string partial collisions with 16 processors and target length 15

<table>
<thead>
<tr>
<th>Ham. Dist.</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>string: abcdefghijklm Hash: 293856a2e2a86faa52f4d203c15e7c9c string: lejdakmhbcg</td>
</tr>
<tr>
<td>12</td>
<td>string: 59a9b4f14d654e047eb20b53d507480df7c96 string: bkjaghlidmicd</td>
</tr>
<tr>
<td>42</td>
<td>string: 639d9266252d3ffcevf53e66f2d6460a635af94 string: ambelkjgifgcd</td>
</tr>
<tr>
<td>43</td>
<td>string: 59d412d69f3f2813b0e05ecaf07034db48c9dc7c8c string: gafbjdikemiejcl</td>
</tr>
<tr>
<td>43</td>
<td>Hash: 408b9e365f6f204bc80436deb5671d203e10d99eca</td>
</tr>
<tr>
<td>processor-0 elapsed time: 497.124382</td>
<td></td>
</tr>
</tbody>
</table>

Table: SHA1 character string partial collisions with 16 processors and target length 15

<table>
<thead>
<tr>
<th>Ham. Dist.</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>string: abcdefghijklm Hash: 857e5f403abe72d5b8a2708821ee33cd0bbece string: njmabiceokfdhgl</td>
</tr>
<tr>
<td>36</td>
<td>string: 8572c14b441eae70f3438ae7989a1ac56d90b394 string: neajkdbcimglfh</td>
</tr>
<tr>
<td>37</td>
<td>string: 35e7c4a440169e7ad66a8b6356001fe23e90aad6 string: lejdakmfibhcg</td>
</tr>
<tr>
<td>41</td>
<td>Hash: 1742d75c97956c7928a0748121a33cc3c8e</td>
</tr>
<tr>
<td>processor-0 elapsed time: 6583.194296</td>
<td></td>
</tr>
</tbody>
</table>