Parallel Hash Collision Search by Rho Method with Distinguished Points

Brian Weber
Computer Engineering Dept.
University of Maryland, Baltimore County
Baltimore, MD 21250, U.S.A.
Email: brianw5@umbc.edu

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

***Abstract*—In this paper, we realized a memory efficient general parallel Pollard's rho method for collision search on hash functions introduced by Van Oorschot and Wiener in 1996. This utilizes the principles of the birthday paradox to greatly increase the probability of a finding a collision, while using significantly less memory than the classic birthday attack, and allowing a larger portion of the subject hash function to be searched before running out of memory by saving only a few select digests called distinguished points. Using our implementation, we are able to find an average of 50 MD5 half collisions in the first hour of searching using a distributed memory high performance computing system called Penzias (one of CUNY HPC systems) on 32 processors. We then extend the technique with Cyrillic character replacement to search for meaningful MD5 half collisions. Next we analyze and measure how the performance of our implementation scales with different processor counts. Finally, we experiment with how the rarity of distinguished points affects the rate at which collisions are found at varying numbers of processors.**

Keywords-Hash function collisions; rho method; distinguished points; MD5; MPI; HPC.

I. Introduction

A cryptographic hash function takes an arbitrary size (much longer) message and produces a fixed size (much shorter) message digest (or hash). Inevitably there will be a lot of distinct messages being hashed to the same digest, this is called hash collisions. But due to the properties of preimage, second preimage, and collision resistances, hash collisions are not easy to find. The previous two crypto groups from the REU (Research Experiences for Undergraduates) Site at the College of Staten Island have done investigations on finding hash collisions by utilizing the CUNY High Performance Computing facilities. The first group was to find second preimage collisions using brute force parallel programming in C and MPI [3]. The second group was to use the classic birthday attack to find arbitrary and meaningful collisions for MD5 and its simplified versions [2].

In this paper, we explore a memory efficient parallel Pollard's rho method with distinguished points, introduced in [15], to search for hash collisions. A distinguished point (DP) is a message digest with a certain number of leading zeros. Searching for DPs is the first important step in searching for hash collisions. The DP search has been used in the mining of bitcoin and other cryptocurrencies. In order to get the reward for creating a block, a bitcoin mining machine or a node needs to find a nonce (a number used only once) such that the digest of the concatenation of this nonce, the digest of previous block, and the list of transactions that make up this block, should be less than a target [8]. The target is a DP. Because this is to find a second preimage of the target, the only way to find such a nonce is to hash (brute-force) enough nonces one by one until you hit one by luck. In bitcoin case, the number of leading zeros in a DP increases over the time to increase the difficulty of the search.

The DP search has also been used in the time memory trade-off (TMTO) attack [7] for breaking ciphers (or one-way functions), which consists of an offline pre-computation phase to prepare tables (a kind of rainbow table) and an online phase. In the offline phase we pre-compute t tables, each contains m pairs of tuples (start-point, endpoint, l). Each chain starts with the start-point, ends at a DP as the endpoint after l iterations, where $l > t$. During the online phase, instead of t^2 table lookups (time complexity), we only need t of these whenever a DP is encountered. This is really good for hardware and parallel implementation [14].

The rest of the paper is organized as follows. In Section II, we briefly introduce hash functions MD5 and its simplified versions, and birthday paradox. In Section III, we introduce Pollard's rho method for collision search and distinguished point. We describe our experiment implementation and search results in Section IV. We give an application of the collision search method to forge two meaningful documents that hash to the same digest in Section V, and we conclude the paper in Section VI.

II. Hash Functions and Birthday Paradox

As a cryptographic primitive, hash function is extremely useful in data and information integrity checking, digital signature schemes, secret sharing schemes [5], Bloom filters [4], and many other security protocols.

A. MD5 and Simplified MD5

MD5 is an iterative hash function based on Merkle-Damgard construction. After padding the message is chopped into 512-bit long blocks. Each block, together with the previous block's hash result, is hashed (compressed) iteratively. The result of last block's hash is the digest

of the entire message. Let H be the hash function MD5, m be a message, h be the digest, i.e. $h = H(m)$. h is a 128-bit long bit-string represented in four 32-bit words $A, B, C,$ and D. Then we have $h = A||B||C||D$, where "||" is the concatenation operator.

Simplified shorter versions of MD5 are introduced in [2] to make experiment easier. Otherwise, we would be waiting forever for just a single full collision. Let h_2 denote the 64-bit MD5 half-hash, it is derived from the original hash. We define h_2 as

$$h_2 = (A||B) \oplus (C||D), \tag{1}$$

where "$\oplus$" is bitwise exclusive-or (XOR) operator. Let h_4 be the 32-bit quarter-hash, which is derived from the four 32-bit words of the original hash. We define h_4 as

$$h_4 = A \oplus B \oplus C \oplus D. \tag{2}$$

MD5 Example: Given the message (a character string without quotes) "The quick brown fox jumps over the lazy dog." Its ASCII value in hexadecimal representation (without spaces) is 54686520 71756963 6b206272 6f776e20 666f7820 6a756d70 73206f76 65722074 6865206c 617a7920 646f672e (length=44).

MD5 full hash: e4d909c290d0fb1ca068ffaddf22cbd0

MD5 half hash: 44b1f66f4ff230cc

MD5 quarter hash: 0b43c6a3

B. Birthday Paradox

Given a set of n people, what is the probability that at least two people share a birthday? It is easier to calculate the probability that no two in the set have the same birthday. The probability for any two people not having the same birthday is $\frac{364}{365}$, because one takes a day, the other has the remaining 364 days to choose from. There are $\binom{n}{2} = \frac{n(n-1)}{2}$ pairs of people. The probability that no two have the same birthday can be obtained by multiplying $\frac{364}{365}$ itself $\binom{n}{2}$ times, i.e., $\overline{P_r}(n) = \left(\frac{364}{365}\right)^{n(n-1)/2}$. Thus the probability of any pair shares a birthday is

$$P_r(n) = 1 - \overline{P_r}(n) = 1 - (364/365)^{n(n-1)/2}. \tag{3}$$

From the above equation, we only need $n = 23$ people, the probability that two people share a birthday reaches 50%. In general given a set S with $|S|$ values, and n randomly chosen values from set S,

$$P_r(n) = 1 - ((|S| - 1)/|S|)^{n(n-1)/2}. \tag{4}$$

The birthday attack is to utilize the birthday paradox to search for hash collisions. In general it follows the following steps. (1) Build up a large list of preimages and their digests. (2) The probability that there will be a collision between any 2 items in the list grows exponentially with respect to the size of the list. (3) Because of this exponential growth, as a rule of thumb, given a hash function with an n bit digest, it should only be treated as strong as a hash function with $n/2$ bits. (4) This is because only roughly $2^{n/2}$ digests are needed to have a 50% chance of finding a collision.

But there are severe drawbacks of the naive birthday attack, for example (1) birthday attacks use a large amount of memory space (i.e., $2^{n/2}$) because you have to save each message and its digest; (2) this greatly limits the amount of messages and hashes you can search.

III. Pollard's Rho Method for Collision Search

A. Pollard's Rho Method

Pollard's rho method, invented by Pollard [9] in 1975, is a way of integer factorization. The improvements were made by Brent [1] in 1980. Due to the birthday paradox, the smaller prime factor p of a composite number $n = pq$ can be found in $O(\sqrt{p})$ operations with over 50% probability. Rho method uses a polynomial modulo n (for example, $u(x) = x^2 + 1 \mod n$) iteratively to generate a pseudo-random sequence. It only needs to save a few variables, and is very space efficient. Because it is in the finite space, the sequence will eventually repeat. The trajectory of the sequence resembles the shape of the Greek letter ρ (rho) ("the ρ must be drawn starting at the bottom"[9]).

We use Figure 1 to illustrate rho method. Given a function f with the same domain and range S ($f : S \to S$), we select a starting value $x_0 \in S$. We produce a pseudo-random sequence $x_i = f(x_{i-1})$, for $i = 1, 2, \ldots$. This sequence will form a cycle eventually because S is a finite set. The sequence starts with a leader and then a cycle. Given x_l as the last point on the leader prior to the cycle, x_{l+1} is on the cycle. Given x_c is the point on the cycle before x_{l+1}, there is a collision because $f(x_l) = f(x_c)$, but $x_l \neq x_c$.

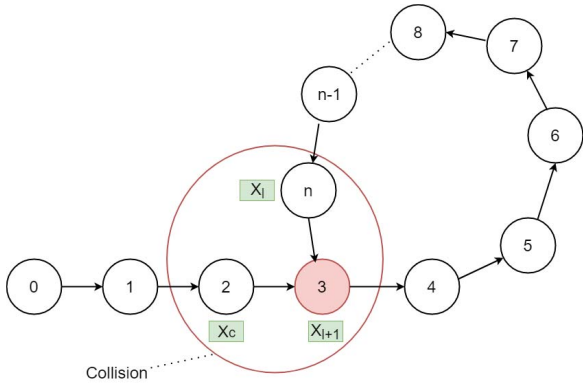

Figure 1: Rho method collision diagram

However, the rho method is serial in nature, and difficult to parallelize it. According to Brent [1] "Parallel implementation of the 'rho' method does not give linear speedup." This is true when each processor produces its own independent sequence of points. A single processor running for a longer time than multiple processors increases the likelihood of finding a collision.

B. Floyd's Algorithm in Collision Search

A smple way to detecting a collision with Pollard's rho method is to apply Floyd's two finger cycle-finding algorithm [12], also known as "tortoise and hare algorithm." We start with two pointers (fingers) a and b at the same random point x_0, i.e., $a = x_0$ and $b = x_0$. Then we advance pointer a at normal speed (one point at a time) as $a = f(a)$, and pointer b at double speed (two points at a time) as $b = f(f(b))$ until they meet as $a = b = x_m$. From the meeting point x_m, we move any one of the two pointers, say a, back to the initial point x_0. Then we advance both pointers, a from x_0 and b from x_m, at equal normal speed until they collide. If we use Figure 1 as an example, the preceding points of the two pointers $a = x_c$ and $b = x_l$ are not equal, but advance each pointer one more step, they collide at x_{l+1}, we find a collision, that is the entry point into the cycle.

When the path has n points and the tail is short, Floyd's two finger algorithm needs $O(3n)$ function evaluations to reach the meeting point, and another $O(2n)$ operations to find entry point (the collision). The total time complexity is $O(5n)$. This is not the most efficient method for collision searching. Moreover, it is not easy to parallelize Floyd's algorithm to speedup the collision search.

C. Using Distinguished Points to Find Hash Collisions

Pollard's rho method for collision search can be parallelized when using the distinguished point, which was introduced by Quisquater and Delescaille for finding DES collisions [10], [11]. The idea was noted earlier by Rivest (see [6] p.100), although he called it endpoint.

A *distinguished point (DP)* is chosen based on a distinguishing and easily testable property. We use a certain number of leading zeros in a message digest as the distinguishing property, i.e., a message digest with a certain number of leading zeros is a DP. The parallel collision search works as follows: (1) Each processor selects a starting point (SP) and produces a trail of points (or digests) until it reaches a DP, it also keeps track on the number of digests produced. (2) When a DP is found, add it to a common list and start a new search from a new SP (so you don't find the same DP from the same SP). (3) A *collision* is found when there is a repeated DP in the common list (illustrated in Figure 2).

Once we have a repeated DP on the list, we can locate the collision without track of all digests. First we hash the longer trail from its SP until its distance to the DP is the same as the distance from the DP to the shorter trail's SP. Then we alternate hashing on each trail until they both reach the same digest, the collision is found (see Figure 3).

Since there is a central list, speedup is very close to linear with increasing processor counts.

The rarity of DPs is a parameter that can be tweaked to give the behavior wanted by the user. We want to **balance distinguished point rarity** by making sure that there are not too many DPs, otherwise memory will run out too quickly.

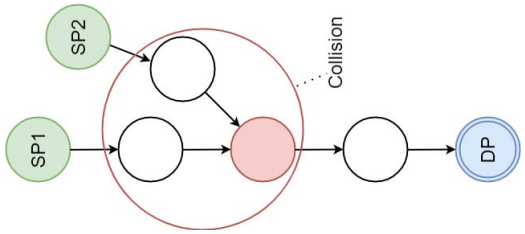

Figure 2: DPs lead to collision diagram (SP: starting point, DP: distinguished point)

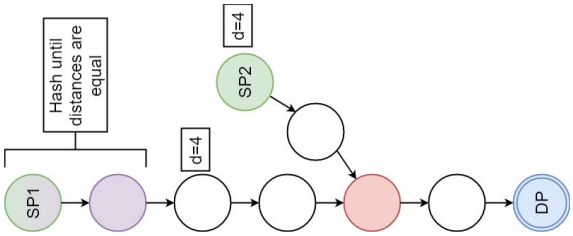

Figure 3: Locating the collision after the repeated DP is found

When there are too many DPs, we can increase the number of leading zeros of DP. Extremely common DPs may also cause communication channels to become over saturated. DPs that are too rare will not be found often enough, and cause other undesirable effects. The goal is to maximize memory use without running out, keep communication at an acceptable level, and keep DPs common enough that they can be found in a reasonable amount of time.

We can get optimal DP rarity. Let n be the number of DPs, Δ the number of leading zeros, and d the number of bits in digest. The odds of **no** repeat given n DPs is $Q(n) = \left(\frac{2^{d-\Delta}-1}{2^{d-\Delta}}\right)^{\frac{n(n-1)}{2}}$. Therefore the odds of repeat given n DPs is $P(n) = 1 - Q(n)$. The average or expected number of DPs e before a repeat is found is

$$e = \sum_{i=0}^{\infty} \left(i \times P(i) \times \prod_{j=0}^{i-1} Q(j) \right). \tag{5}$$

When DPs are too uncommon, the two things become more common. (1) Robinhood: when the trail of one starting point hits another starting point, causing a pseudo collision (see Figure 4). (2) Unnamed problem: if a distinguished point if found more than twice, the same collision may be found more than once, even though there is a new unique collision to be found. This is because only one starting point is stored per DP (see Figure 5).

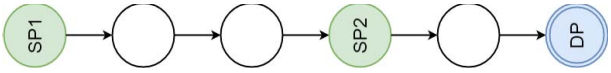

Figure 4: Robinhood case

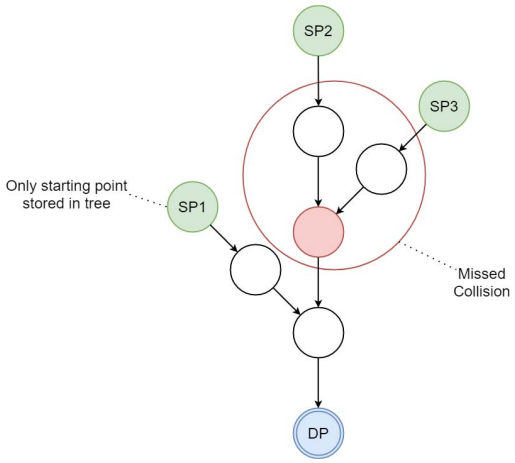

Figure 5: Unnamed problem

IV. Experiment Implementation and Results

A. Implementation Overview

In our implementation, we chose to use a master worker model. A master node coordinates each worker, and assigns jobs to worker nodes as soon as they complete their tasks. In this case, workers' primary task is searching for distinguished points, and secondarily searching for the collision point after repeating DPs have been found. Workers are assigned a fixed portion of the full search space to look. After a repeated distinguished point is found, a worker is paused, and is assigned to find the collision corresponding to the repeated DPs. Once the collision is found, it is reported to the master, and the worker resumes where it left off.

The master node is responsible for maintaining a record of the status of each worker node, as well as maintaining the list of DPs, and looking for repeats. The list is structured as a binary search tree for fast sorting and easy repeat detection.

B. Collision Search Results

Our implementation appeared to be relatively effected at finding collisions. While using quarter MD5, communication channels were so saturated by DP messages that using lower processor counts resulted in higher collision rates, due to the lower amount of communication. Figures 6 through 10 show some comparisons of different parameters that were tested in our searches. All relationships match what is theoretically expected, with some minor exceptions.

Figure 6 shows the number of DPs found vs the number of processors at different DP rarities. The number of DPs found increases almost perfectly linearly with increasing processor amounts, highlighting even with large numbers of DPs. This highlights the efficiency and scalability of our implementation. The communication overhead makes no significant impact on performance.

Figure 7 shows the numer of leading zeros vs the number of DPs found at different processor counts. As expected, the number of DPs changes exponentially with a linear change in the number of leading zeros. This is because the rarity of distinguished points changes exponentially with a linear change in the number of leading zeros, which can be seen through a simple counting argument. In this figure we see our first discrepancy however. With 32 processors searching for DPs with 8 leading zeros, communication costs become very high, drastically reducing performance.

Figure 8 clearly shows that the number of DPs found increases linearly with time as expected. Figure 9 shows the birthday paradox in effect. As the number of DPs increases linearly, the number of collisions begins to increase exponentially. Figures 8 and 9 used three trials of 32 processors on half MD5 with 20 leading zeros for half an hour. Each line refers to a different trial at a different starting point.

Finally, in an effort to find the optimal amount of leading zeros for finding collisions, an experiment comparing the number of collisions found vs the number of leading zeros was done at varying number of processors. Figure 10 shows the results of this experiment, which suggest that the optimal amount of leading zeros is somewhere around half the length of the size of the digest for half MD5.

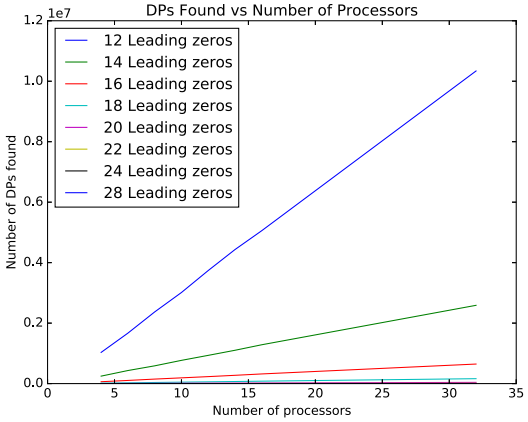

Figure 6: # of DPs found vs # of processors

V. Applications

A. Motivations and Digital Signatures

All experiments discussed up to this point serve no purpose except as a benchmark of how MD5 stands up to brute force attacks from modern hardware. In order to gain more useful results, further algorithms must be introduced, such as a method for forging digital signatures. This section introduces the method that we chose to implement, and elaborates on the results that we got.

A digital signature is a cryptographic tool used to bind messages to their sender, so that the receiver can verify the authenticity of the messages. The process of sending a signed message is as follows. First, the sender hashes the message they would like to send to obtain its digest. Next,

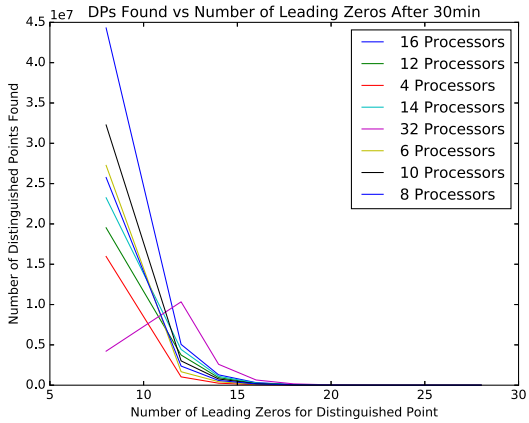

Figure 7: # of DPs found vs # of leading zeros

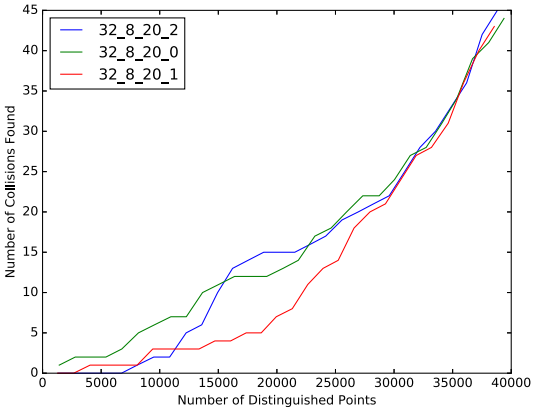

Figure 9: # of collisions found vs # of DPs found

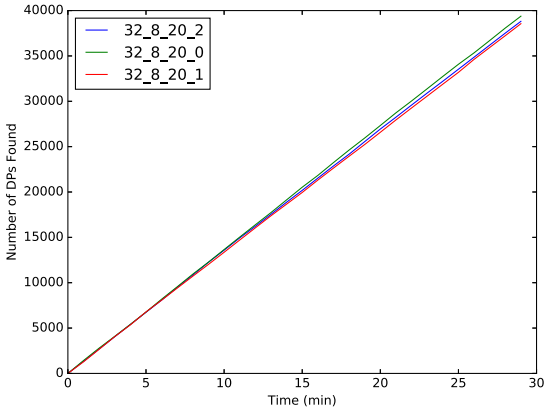

Figure 8: # of DPs found vs time

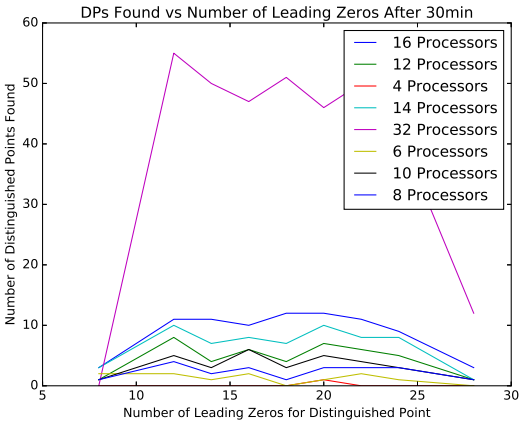

Figure 10: # of collisions found vs # of leading zeros

the sender encrypts the resulting digest with their private key. The output of the encryption is the final signature, which is sent along with the message to the recipient. To verify the message, the recipient hashes the message to obtain the its digest, and decrypts the signature using the sender's public key. If the two digests match, then the signature is valid.

B. *Forged Documents*

Digital signatures can be defeated if an adversary is able to generate two messages m and m' that have different meanings, but the same digest. The messages are written such that the sender would be willing to sign one, but not the other. The adversary can repeatedly manipulate the messages m and m' such that their digests change, but the semantic meaning of the messages do not, until he finds a message pair with the same digest. He can then present the sender with the benign message to sign, and receive back a signature that is valid for both the benign and malicious message.

Example of Forged Documents:

Document attacker wants signed (m): "Good afternoon,

I agree to pay you **$1000000** dollars for your super expensive, fancy car. I will send you the money on Tuesday, and will pick it up on Wednesday.

Best, John Smith"

Document victim will sign (m'): "Good afternoon,

I agree to pay you **$1000** dollars for your super expensive, fancy car. I will send you the money on Tuesday, and will pick it up on Wednesday.

Best, John Smith"

The attacker alters both messages and hashes each version until the digests are the same. The victim signs the altered version of m' and the signature can be copied to the altered version of m.

C. Generating Two Documents with the Same Digest

In this section, we introduce an algorithm that we used for obtaining two messages with the same digest.

Let R be the set of all possible digests ($h \in R$). We split R into two roughly equal sized subsets S_1, S_2. Let $g_m(h) : R \rightarrow M$ be an injective function, which takes a digest h and a message $m \in M$ and generates another version of m with the same semantic meaning. We define a function $f : R \rightarrow R$ as follows

$$f(h) = \begin{cases} H(g_m(h)) & \text{if } (h \in S_1) \\ H(g_{m'}(h)) & \text{if } (h \in S_2). \end{cases} \quad (6)$$

The above function generates deterministic trails of digests, exactly like the cycling experiments shown in the earlier sections. Because of this, the DP algorithm can be used without modification using this function. Once a collision is found, as long as the digests are from different messages, the message pair has been found. However, because the digests must be from different messages, the probability of finding a valid collision is reduced by 1/2 from the cycling experiments.

D. Implementation of Altering Documents

In this subsection, we introduce the function $g(h)$ that we used to generate altered messages. In UTF-8 encoding, there are several characters in different alphabets with the same appearance, but different hex encodings. For example, the lowercase a in the Latin alphabet looks exactly the same as the Cryllic small letter a, but have the hex value 0x61 and 0xD0B0, respectively.

We found every such pair for the Latin alphabet and created a table of each corresponding hex value, see Figures 11 and 12. To modify the messages, we create a list of all letters in the message that has a valid replacement and assign it an index, up to the length of the chosen hash function's digest - 1, or until the end of the message is reached. The more replaceable characters there are, the more likely the collision search is to be successful.

After the indexes have been assigned, we hash. Then, our $g(h)$ function changes the messages based on the digest in the following way: if the bit in position n of the digest is a 0, then the replaceable letter in index n remains a Latin letter, otherwise, it is changed to its alternate value. For example, if the digest of the message "Hello" is 0x10, then the letter e will be replaced with its Cryllic counterpart.

With this technique, we were able to find a collision for the example messages given in Section V-B. A running result is shown in Figure 13, and the two messages hash to the same digest 0xF9F93282FC813DFD for half MD5.

VI. Conclusions and Future Work

In conclusion we were able to implement a memory efficient collision search algorithm by using the distinguished point method. This method is vastly more memory efficient than the classical birthday attack, while benefiting from the same exponential speedup as a result of the birthday paradox. Our implementation is not noticeably affected by communication overhead, and scales linearly with increasing processor counts. We were also able to extend our implementation of the distinguished point method to forge signatures that used half MD5 using methods introduced by Van Oorschot and Wiener, and using our novel version of the $g(h)$ function.

In the future, we would like to extend our experiments to SHA1. We have successfully found collisions for half SHA1, but have not conducted large scale experiments in the style we have for MD5. In addition, we would like to port our implementation to CUDA to make use of GPUs. GPUs have been shown to be able to calculate message digests an order of magnitude faster than general purpose CPUs [13].

Capital	Replacement	Replacement Title	Unicode	Alt+Decimal	Lowercase	Replacement	Replacement Title	Unicode	Alt+Decimal
A	A	Greek Letter Capital Alpha	U+0391	913	a	a	Cyrillic Small Letter A	U+0430	1072
B	B	Greek Letter Capital Beta	U+0392	914	b				
C					c				
D					d	d	Cyrillic Small Letter Komi De	U+0501	1281
E	E	Greek Letter Capital Epsilon	U+0395	917	e	e	Cyrillic Small Letter Ie	U+0435	1077
F					f				
G	G	Cyrillic Capital Letter Komi Sje	U+050C	1292	g				
H	H	Greek Letter Capital Eta	U+0397	919	h	h	Cyrillic Small Letter Shha	U+04BB	1211
I	I	Greek Letter Capital Iota	U+0399	921	i	i	Cyrillic Small Letter Byelorussian-Ukrainian I	U+0456	1110
J	J	Cherokee Letter Gu	U+13AB	5035	j	j	Cyrillic Small Letter Je	U+0458	1112
K	K	Greek Letter Capital Kappa	U+039A	922	k				
L	L	Cherokee Letter Tle	U+13DE	5086	l				
M	M	Greek Letter Capital Mu	U+039C	924	m				
N	N	Greek Letter Capital Nu	U+039D	925	n				
O	O	Greek Letter Capital Omicron	U+039F	927	o	o	Cyrillic Small Letter O	U+043E	1086
P	P	Greek Letter Capital Rho	U+03A1	929	p				
Q	Q	Cyrillic Capital Letter Qa	U+051A	1306	q	q	Cyrillic Small Letter Qa	U+051B	1307
R	R	Cherokee Letter E	U+13A1	5025	r				
S					s	s	Cyrillic Small Letter Dze	U+0455	1109

Figure 11: Replacement Table: Concept

```
#define REPLACE_TABLE_UPPERCASE \
        {0xCE91, 0xCE92, 0x0000, 0x0000, 0xCE95, 0x0000,\
         0x0000, 0xCE97, 0xCE99, 0x0000, 0xCE9A, 0x0000,\
         0xCE9C, 0xCE9D, 0xCE9F, 0xCEA1, 0xD49A, 0x0000,\
         0x0000, 0xCEA4, 0x0000, 0x0000, 0xD49C, 0xCEA7,\
         0xCEA5, 0xCE96}

#define REPLACE_TABLE_LOWERCASE \
        {0xD0B0, 0x0000, 0x0000, 0xD481, 0xD0B5, 0x0000,\
         0x0000, 0xD2BB, 0xD196, 0xD198, 0x0000, 0x0000,\
         0x0000, 0x0000, 0xD0BE, 0x0000, 0xD49B, 0x0000,\
         0xD195, 0x0000, 0x0000, 0x0000, 0xD49D, 0xD185,\
         0xD183, 0x0000}
```

Figure 12: Replacement Table: Code

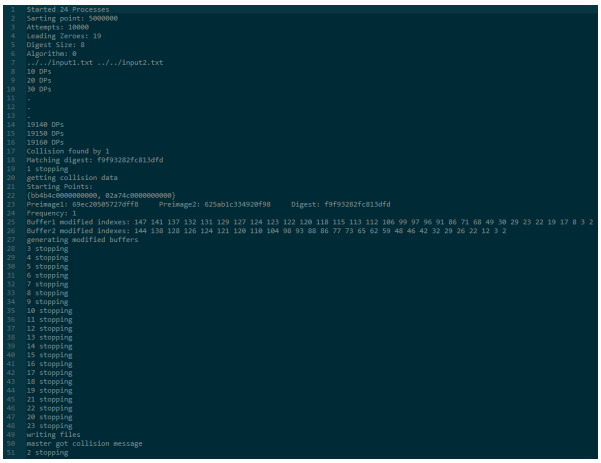

Figure 13: A sample run result

ACKNOWLEDGMENTS

This work is supported, in part, by NSF/DoD REU Site[1] (Award No.: 1359266) at CUNY-College of Staten Island. We sincerely thank Samantha Morris for her help on writing/testing code and collecting/analyzing the experiment results. We also thank the CUNY HPC Center[2] for providing MPI training and their facilities.

REFERENCES

[1] R. Brent. An Improved Monte Carlo Factorization Algorithm. *BIT Numerical Mathematics*, 20(2), 1980, pp. 176-184.

[2] M. Cantu, J. Kim, and X. Zhang. Finding Hash Collisions using MPI on HPC Clusters. *Proc. of 2017 IEEE Long Island Systems, Applications and Technology Conference (LISAT 2017)*, Farmingdale, NY, May 5, 2017, pp. 1-6.

[3] V. Chiriaco, A. Franzen, R. Thayil, and X. Zhang. Finding partial hash collisions by brute force parallel programming. *Proc. of IEEE LISAT 2017*, Farmingdale, NY, May 5, 2017, pp. 1-6

[4] C. Chum and X. Zhang. A new Bloom filter structure for searchable encryption schemes. *Proc. of 7th ACM Conference on Data and Application Security and Privacy (CODASPY 2017)*, Scottsdale, Arizona, March 22-24, 2017, pp. 143-145.

[5] C. Chum and X. Zhang. Hash function based secret sharing scheme designs. *Security and Communication Networks*. 6(5), 2013, pp. 584–592.

[6] D. Denning. *Cryptography and Data Security*. Addison-Wesley, 1982.

[7] M. Hellman. A Cryptanalytic Time-Memory Trade-Off. *IEEE Transactions on Information Theory*, 26(4), 1980, pp. 401–406.

[8] A. Narayanan, J. Bonneau, E. Felten, A. Miller, and S. Goldfeder. *Bitcoin and Cryptocurrency Technology*. Princeton University Press, 2016.

[9] J. Pollard. A Monte Carlo method for factorization. *BIT Numerical Mathematics*, 15 (3), 1975, pp. 331-334.

[10] J. Quisquater and J. Delescaille. How easy is collision search? Application to DES. *Proc. of Eurocrypt 1989*, pp. 429-434.

[11] J. Quisquater and J. Delescaille. How easy is collision search. New results and applications to DES. *Proc. of Crypto 1989*, pp. 408-413.

[12] A. Shamir. Random graphs in cryptography. *The Onassis Foundation Science Lecture Series*, June 28, 2010.

[13] M. Sprengers. *GPU-based Password Cracking*. Master of Science Thesis, Radboud University Nijmegen. February 2011.

[14] F. Standaert, G. Rouvroy, J. Quisquater, and J. Legat. A Time-Memory Tradeoff Using Distinguished Points: New Analysis & FPGA Results. Proc. of Workshop on Cryptographic Hardware and Embedded Systems (CHES 2002), LNCS 2523, Springer-Verlag, 2003, pp. 593–609.

[15] P.C. van Oorschot and M.J. Wiener. Parallel collision search with cryptanalytic applications. *Journal of Cryptology*, 12(1), 1999, pp. 1-28.

[1] http://www.cs.csi.cuny.edu/REU

[2] https://cunyhpc.csi.cuny.edu