Universiteit Leiden
Opleiding Informatica

A comparison of
hashing algorithms

Name: Nick van den Bosch
Date: August 19th
1st supervisor: Dr. Michael Lew
2nd supervisor: Dr. Erwin M. Bakker

BACHELOR'S THESIS

Leiden Institute of Advanced Computer Science (LIACS)
Leiden University
Niels Bohrweg 1
2333 CA Leiden
The Netherlands

Abstract

Hashing is a way to map digital data of arbitrary size to a hash of a fixed size. This method of mapping data is used in a variety of different processes such as verifying data with cryptographic hash functions or finding duplicate records in data by using a hash table. In this document we will look into the performance of several hashing algorithms to see which algorithm performs best at particular bitsizes. This was done by creating two big datasets and comparing the time it took to hash the data and checking the collision rates of the hashed data, first at the standard fixed bitsize of each algorithm and then on cuts of the ha shed data.

Contents

1 Introduction **4**

2 Related Work **5**
- 2.1 Speeding up Sha hash Functions on 2nd generation Intel Core processors . . . 5
- 2.2 Segmented Hash tables . . . 6
- 2.3 Symmetric loading with Performance Guarantees for Distributed Hash Tables . . . 6
- 2.4 A Hash-based Path Identification Scheme for DDoS Attacks Defence . . . 6
- 2.5 Hardware-based Multi-hash Scheme for High Speed IP Lookup 6

3 Methodology **7**
- 3.1 Dataset Description . . . 7
  - 3.1.1 URL Dataset . . . 7
  - 3.1.2 Random Number Dataset . . . 7
- 3.2 Algorithm Description . . . 8
  - 3.2.1 MD5 . . . 8
  - 3.2.2 Sha-1 . . . 9
  - 3.2.3 Sha-256 . . . 9
  - 3.2.4 Fnv-1a . . . 10
  - 3.2.5 Crc32 . . . 10
- 3.3 Comparing the Algorithms . . . 11

4 Results **11**

5 Conclusion and Discussion **14**

6 References **14**

1 Introduction

Hash algorithms are used in a variety of different fields in computing, such as finding duplicate records in an unsorted file using a hash table[11], verifying file integrity and password verification using cryptographic hash functions[10][14][16]. The way a hash function works is that it converts any form of digital data of arbitrary size and maps this to another piece of digital data of fixed size. The return value of this process is called a hash.

An important property of hash functions is that the hashed data should be as unique as possible, meaning that it should be very difficult to find two different inputs i1 and i2 so that hash(i1) = hash(i2). This property is called collisions resistance[15] and if two different inputs hash to the same output we speak of a collision. In addition to this, cryptographic hash functions also have two other important properties to ensure data security, pre-image resistance and second pre-image resistance [1]. Pre-image resistance is that given a hash value h it should be very difficult to find the input i such that h = hash(i), meaning that the hash is strictly a one-way function. Second pre-image resistance means that given an input i1 it should be very difficult to find an input i2 such that hash(i1) = hash(i2). Second pre-image resistance in the cryptographic hashing algorithm Sha-1 is shown in figure 1, where the inputs are near exactly the same, but the hashes are completely different.

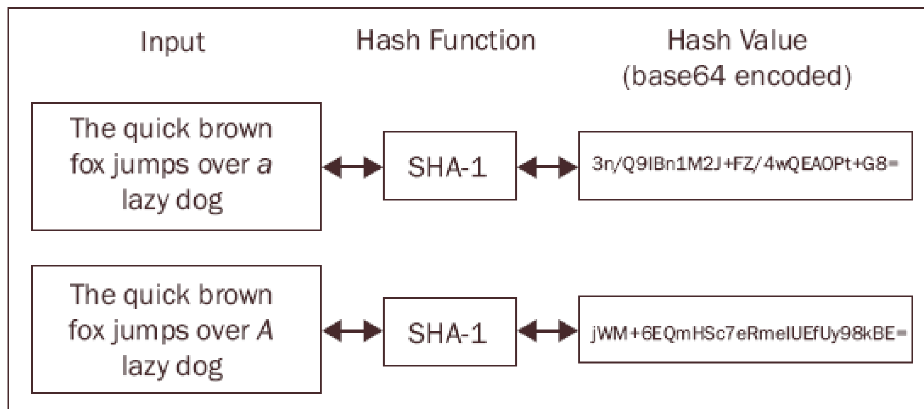

Figure 1: Example of pre-image resistance in hashing

An example of the use of hash functions is for checking for duplicate records in data. With a hash function, each record of the original data is mapped to an index of a hash table T. After all the records have been hashed, any duplicates will be found by going through T for every index and checking whether there are multiple hashed records on one index. If there are multiple hashed records on one index, you fetch the original records and compare them. When using an appropriate hash table size, finding duplicates this way is likely to be faster than any alternative approach.

When using hash functions you want the result to be optimal, thus having the least amount of collisions as possible with using the minimum number of bits for your hash. In this document we will be comparing the performance of several commonly used hashing algorithms on a number of different output sizes to see how each algorithm performs for its standard output size and on cuts of the standard size.

2 Related Work

In this section some of the previous work that was done in the field of hash functions will be discussed.

2.1 Speeding up Sha hash Functions on 2nd generation Intel Core processors

In this article [5] Gueron describes software techniques that optimise the speed of Sha-1, Sha-256 and Sha-512 on 2nd generation intel core processors. This is done by replacing specific instructions to enhance the performance of these algorithms for this processor family. Results of this research were speedup factors of 1.19 for SHA-1, and 1.22 for SHA-256 and SHA-512.

2.2 Segmented Hash tables

Kumar and Crowley propose a segmented hash table architecture[12], splitting the memory that is used to store the hashes into N logical segments. Each input then has N potential storage locations and the destination segment is chosen as to minimise collisions. This resulted in significant performance improvements. The average case performance is improved by 40%.

2.3 Symmetric loading with Performance Guarantees for Distributed Hash Tables

Hsiao and Chang discuss a new load balancing algorithm for reallocation of virtual servers in distributed hash tables[13]. Their algorithm has improved performance metrics, including load balance factor improvements and a better algorithmic conversion rate compared to similar researches.

2.4 A Hash-based Path Identification Scheme for DDoS Attacks Defence

With the more frequent use of distributed denial of service (DDoS) attacks, Jin et al. describe a hash based path identification for protecting against these kinds of attacks regardless of the IP addresses used[17]. By using several different filtering strategies, the victim that is being attacked is able to accept most of the legitimate packets and drop the majority of the malicious packets.

2.5 Hardware-based Multi-hash Scheme for High Speed IP Lookup

In this document Demetriades et al. discuss a new hash-based IP lookup scheme that achieves high storage efficiency and high performance[18]. They do this by using a multi-hashing approach and resolving hash collisions by dynamically migrating IP prefixes that are already in the lookup table as new prefixes are inserted. When compared to similar methods for this problem, their scheme reduces area and power requirements by 60% and 80% respectively, while achieving competitive lookup rates.

3 Methodology

The methodology section is divided in three topics. The first section will give an overview of the data used in the experiment. The second section will discuss the algorithms that were used. In the third section the process of hashing the data and checking for collisions will be explained.

3.1 Dataset Description

This section will discuss the characteristics of the two datasets that were used for this research. Both datasets were text files created for this project and thus were not used before. The type of data used for both of the datasets are C++ strings and this data was hashed to the same type of C++ strings.

3.1.1 URL Dataset

The main and biggest dataset consists of 1 billion entries. Each entry containing a concatenated string. To create this dataset we used an array filled with 1000 different words and concatenated them in every possible way to get 1 billion different URLs. An example of an entry of this dataset is "have.old.read".

3.1.2 Random Number Dataset

The secondary and smaller dataset consists of 10 million entries of random numbers created with a Linear Congruential Generator[2]. The range of the numbers is between 0 and $2^{32} - 1$. After creation, the dataset was checked for uniqueness to avoid unnecessary collisions for the research.

3.2 Algorithm Description

For this research we have used 5 different algorithms to hash the data:
Three cryptographic hash algorithms.

- MD5
- Sha-1
- Sha-256

A non-cryptographic hash algorithm.

- Fnv-1a

A cyclic redundancy check commonly used as hash function.

- Crc32

3.2.1 MD5

MD5 is a widely used cryptographic hash function producing a 128 bit hash value [3]. It was designed by Ronald Rivest in 1991 to replace an earlier hash function, MD4. It was first published in April 1992.
The fundamental method of MD5 is as follows: First, the input message is broken up into chunks of 512 bit blocks (sixteen 32 bit words). For this, the message is padded, meaning bits are added so that it is divisible by 512. The algorithm operates on a 128-bit state, which is divided in four 32-bit words, denoted A,B,C, and D. These are initialised to certain fixed constants.
The processing of a block consists of four stages, termed rounds. Each round is based on 16 similar operations based on a non-linear function F, modular addition and left rotation. A different function F is used in each round and the operations used are combinations of the logical operations XOR, AND, OR, and NOT.

The differences between MD5 and its predecessor MD4 are that a fourth round of operations is added, additional constants have been added, each step now adds in the result of the previous step which promotes a faster avalanche effect, shift amounts have been optimised, and the order in which input words are accessed in rounds 2 and 3 is changed.
In 2004 it was shown that MD5 is not collision resistant. As such, MD5 is not suitable for applications like SSL certificates or digital signatures that rely on this property for digital security.

3.2.2 Sha-1

Sha-1 is a cryptographic hash function designed by the United States National Security Agency [4]. It was first published in 1995. Sha-1's message digestion is based on principles similar to those in the design of MD4 and MD5.
First, constants and buffers are defined with initial values. The input message is padded and split in 512-bit blocks. Each block is then divided in 16 words. Before processing any blocks, 5 variables H0, H1, H2, H3, and H4 are initialised. These are then copied into different variables A, B, C, D and E. After this, same as with MD5, functions F composed of combinations of logical operations are ran on the variables A through E and parts of the input message. Then the result of H0 + A, H1 + B, H2 + C, H3 + D, and H4 + E are stored into the variables H1 through H5. After this process, the message digest is represented by the 5 words H0 H1 H2 H3 H4.
Sha-1 is very similar to its predecessor Sha-0. SHA-1 differs from SHA-0 only by a single bitwise rotation in the message schedule of its compression function. This was done, according to the NSA, to correct a flaw in the original algorithm which reduced its cryptographic security.

3.2.3 Sha-256

SHA-2 is a set of cryptographic hash functions designed by the United States National Security Agency [6]. Sha-2 functions include significant changes from their predecessor Sha-1. Some elements of Sha-1 are still used in Sha-256, for example the padding to alter the input message to be a multiple of 512. For Sha-256, instead of the 5 variables in Sha-1, H0 through H4 and A through E, eight variables are used.

The blocks of 512 are divided in sixteen 32 bit words and expanded to 64 words, one for each round of the compression function. As with Sha-1 and MD5, logical operations and rotations are performed on these input words and variables and the message digest is represented by the words H0 through H7.

3.2.4 Fnv-1a

Fowler-Noll-Vo (FNV) is a non-cryptographic hash function created by Glenn Fowler, Landon Curt Noll, and Kiem-Phong Vo. The basis of the FNV hash algorithm was taken from an idea sent as reviewer comments to the IEEE POSIX P1003.2 committee by Glenn Fowler and Phong Vo in 1991. Currently, there are two different versions of the FNV hash, FNV-1 and FNV-1a. These are commonly used in hash performance researches [8] [9].
The basis of the FNV-1 function is: First a variable 'hash' is initialised. Then for each byte of data to be processed, two operations are performed. First the hash is multiplied with a specific prime number, then this result is XOR'd with the byte of data that needs to be processed.
The FNV-1a function differs from the FNV-1 hash by the order in which the multiplication and XOR are performed. For this function the XOR is performed first, after which the multiplication with the prime is done.

3.2.5 Crc32

Crc32 is a cyclic redundancy check which is an error-detecting code commonly used to detect accidental changes to raw data [7]. Because the check value that this function returns has a fixed length, the function is occasionally used as a hash function. The CRC was invented by W. Wesley Peterson in 1961. The 32-bit CRC function of Ethernet and many other standards is the work of several researchers and was published in 1975.
The fundamental method in a crc is that the input represented in binary is repeatedly XOR'd with a polynomial also represented in binary. This means the original message is repeatedly divided by the polynomial until the remainder value is left. This remainder value is the return value of the crc, which in our case is the hash value.

3.3 Comparing the Algorithms

For this experiment we first created the data as discussed in an earlier section. Implementations for each of the used algorithms were obtained and some were slightly altered to function for hashing strings instead of other datatypes.

Both of the datasets were hashed at the standard fixed size of each algorithm which resulted in 10 hashed files. Multiple cuts were made of these files, where each of these cut files would contain substrings of the original hash but at a smaller fixed size.
Every file in this collection of hashed data was then sorted and checked for consecutive pairs. These consecutive pairs, representing collisions, were then counted and displayed in the tables in the results section.

4 Results

In table 1 we can see the amount of collisions of each algorithm at the different hash sizes for the URL dataset. The hash sizes that we tested on are shown in the first column. Every other column shows the amount of collisions per algorithm at the hash size to its left. Some fields show dashes, because the standard output size of this algorithm is smaller than the size of that field and so we haven't tested this algorithm for that size.

As we can see in the table, if we reduce the size of the hash, the amount of collisions increases. This was to be expected, because when you have less characters in your output, there will be less possible unique combinations and thus there will be more collisions on very big datafiles. However, the table shows different results for the Fnv-1a algorithm, which are not really comparable to the other algorithms. As stated in at section 3, the Fnv-1a implementation used in this experiment did not have a fixed outputsize and for this reason was expected to give different results. Due to this, the results for Fnv-1a are really far from the other algorithms and cannot be compared to the results of the others.

Hash Size	Sha256	Sha1	MD5	Fnv1a	Crc32
64	0	-	-	-	-
40	0	0	-	-	-
32	0	0	0	-	-
16	0	0	0	-	-
12	1780	1752	1812	-	-
11	28433	28630	28314	-	-
10	456237	456077	454572	108079568	-
9	7253277	7259196	7256042	556370433	-
8	108107094	108081539	108080386	208826336	108086845
7	725305601	725290703	725315707	31505141	725292526

Table 1: Collisions of URL dataset

When we look at the table we can see that for the sizes of 16-64 characters there are no collisions for any algorithm, this is probably because the used dataset does not have enough entries to cause collisions for these sizes. When looking at the table for the sizes 12-9 we can see that difference in the amount of collisions between the algorithms is relatively small (when excluding Fnv-1a for earlier given reasons). We see that MD5 performs best for sizes 11 and 10, Sha-1 performs best for size 12 and Sha-256 performs best when the outputsize is 9. When looking at the even smaller sizes and including Crc32, the table shows that the amount of collisions of Crc32 is comparable to the other three algorithms. We also see that the amount of collisions is getting larger, but the differences in collisions between the algorithms are relatively small compared to the total amount of collisions.

Tables 2 and 4 show the time it took to hash the data at the standard size of each algorithm for both of the datasets. We can see that the algorithms that have a larger outputsize take more time to hash the data.

	Sha256	Sha1	MD5	Fnv1a	Crc32
Time	1h7m5s	57m16s	38m54s	14m42s	19m35s

Table 2: Elapsed time for hashing URL dataset

Table 3 is the same as table 1, but with the results of the random number dataset. The results are partially alike, seeing that the collision rates go up when the hash size is lowered. However, Sha-1 performs best for sizes 8,9 and 10 for this dataset and MD5 performs best for size 11, showing different performance than with the other dataset. What stands out in this table is that Crc-32 has a lot less collisions than the other algorithms on size 7.

Hash Size	Sha256	Sha1	MD5	Fnv1a	Crc32
64	0	-	-	-	-
40	0	0	-	-	-
32	0	0	0	-	-
16	0	0	0	-	-
12	0	0	0	-	-
11	6	2	1	-	-
10	42	41	45	11510	-
9	708	696	712	108420	-
8	11699	11576	11650	1030856	12967
7	184578	184101	184032	5639774	176504

Table 3: Collisions of random number dataset

	Sha256	Sha1	MD5	Fnv1a	Crc32
Time	2m45s	1m43s	29s	7s	9s

Table 4: Elapsed time for hashing random number dataset

5 Conclusion and Discussion

For this research we tried to find out what hashing algorithms have the least amount of collisions for different hash sizes. The results showed that the difference in collisions between the algorithms was relatively small compared to the total amount of collisions.

Comparing the two datasets per hash size, we see a difference in which algorithms have the least amount of collisions. For the URL dataset, MD5 is the algorithm that has the least amount of collisions for most hash sizes and for the random number dataset this is Sha-1. The biggest difference between the two datasets is that Crc-32 is by far the best at size 7 in the random number dataset, which is not the case for the other dataset.
Besides this, for this experiment there is no algorithm that has a substantially smaller amount of collisions than the others at any size.

6 References

[1] Rogaway, P., & Shrimpton, T. (2004, January). Cryptographic hash-function basics: Definitions, implications, and separations for preimage resistance, second-preimage resistance, and collision resistance. In Fast Software Encryption (pp. 371-388). Springer Berlin Heidelberg.

[2] Fontaine, C. (2011). Linear Congruential Generator. Encyclopedia of Cryptography and Security, 721-721

[3] Jrvinen, K., Tommiska, M., & Skytt, J. (2005, January). Hardware implementation analysis of the MD5 hash algorithm. In System Sciences, 2005. HICSS'05. Proceedings of the 38th Annual Hawaii International Conference on (pp. 298a-298a). IEEE.

[4] Xiao-Hui, C., & Jian-Zhi, D. (2010, April). Design of SHA-1 Algorithm based on FPGA. In Networks Security Wireless Communications and Trusted Computing (NSWCTC), 2010 Second International Conference on (Vol. 1, pp. 532-534). IEEE.

[5] Gueron, S. (2012, April). Speeding Up SHA-1, SHA-256 and SHA-512 on the 2nd Generation Intel Core Processors. In 2012 Ninth International Conference on Information Technology-New Generations (pp. 824-826). IEEE.

[6] Gilbert, H., & Handschuh, H. (2004, January). Security analysis of SHA-256 and sisters. In Selected areas in cryptography (pp. 175-193). Springer Berlin Heidelberg.

[7] Koopman, P. (2002). 32-bit cyclic redundancy codes for internet applications. In Dependable Systems and Networks, 2002. DSN 2002. Proceedings. International Conference on (pp. 459-468). IEEE.

[8] Breitinger, F., & Baier, H. (2013). Similarity preserving hashing: Eligible properties and a new algorithm mrsh-v2. In Digital forensics and cyber crime (pp. 167-182). Springer Berlin Heidelberg.

[9] Estbanez, C., Saez, Y., Recio, G., & Isasi, P. (2014). Performance of the most common noncryptographic hash functions. Software: Practice and Experience, 44(6), 681-698.

[10] Alkandari, A. A., Al-Shaikhli, I. F., & Alahmad, M. A. (2013, September). Cryptographic Hash Function: A High Level View. In 2013 International Conference on Informatics and Creative Multimedia (pp. 128-134). IEEE.

[11] Eltabakh, M. Y., Ouzzani, M., & Aref, W. G. (2007, July). Duplicate Elimination in Space-partitioning Tree Indexes. In Scientific and Statistical Database Management, 2007. SSBDM'07. 19th International Conference on (pp. 18-18). IEEE.

[12] Kumar, S., & Crowley, P. (2005, October). Segmented hash: an efficient hash table implementation for high performance networking subsystems. In Proceedings of the 2005 ACM symposium on Architecture for networking and communications systems (pp. 91-103). ACM.

[13] Hsiao, H. C., & Chang, C. W. (2013). A symmetric load balancing algorithm with performance guarantees for distributed hash tables. Computers, IEEE Transactions on, 62(4), 662-675.

[14] Liu, F. (2011, August). On the security of digest access authentication. In Computational Science and Engineering (CSE), 2011 IEEE 14th International Conference on (pp. 427-434). IEEE.

[15] Mozafari, B., & Savoji, M. H. (2006, December). A new collision resistant hash function based on optimum dimensionality reduction using Walsh-Hadamard transform. In Information Technology, 2006. ICIT'06. 9th International Conference on (pp. 149-154). IEEE.

[16] Aslam, B., Wu, L., & Zou, C. C. (2010, July). PwdIP-Hash: a lightweight solution to phishing and pharming attacks. In Network Computing and Applications (NCA), 2010 9th IEEE International Symposium on (pp. 198-203). IEEE.

[17] Jin, G., Zhang, F., Li, Y., Zhang, H., & Qian, J. (2009, October). A Hash-Based Path Identification Scheme for DDoS Attacks Defense. In Computer and Information Technology, 2009. CIT'09. Ninth IEEE International Conference on (Vol. 2, pp. 219-224). IEEE.

[18] Demetriades, S., Hanna, M., Cho, S., & Melhem, R. (2008, August). An efficient hardware-based multi-hash scheme for high speed IP lookup. In High Performance Interconnects, 2008. HOTI'08. 16th IEEE Symposium on (pp. 103-110). IEEE.