Investigation into the relation between processing power and the speed of attacks against cryptographic hashing functions

To what extent do increasing processor speeds impact the efficiency of brute-force attacks against Merkle-Damgård-based cryptographic hashing functions?

Elian Rieza
https://elianrieza.dev
ARCHIVED IB FORMAL RESEARCH

Contents

1 Introduction

The advent of cryptographic hash functions (CHFs) was a crucial breakthrough in digital cryptography; designed to provide irretrievable fingerprints for any form of digital data. The Merkle-Damgård construction (Merkle, 1979) marked a significant milestone which served as a basis for developing subsequent hash functions families such as MD and SHA (Rivest, 1978). CHFs produce a secure and unique hash that is nearly impossible to reverse-engineer and could be provided as a "shared-secret". As faster hardware are released by manufacturers, processing time for computations would decrease, forming a time-memory trade-off where computations needed for CHFs decrease and generated in a faster amount of time, as faster hardware become commercially available.

Flagship CPUs of chip-manufacturing companies have been designed with high clock speeds, multiple cores and larger caches which can perform cryptographic operations at faster speeds than their predecessors. This includes supporting advanced instruction sets which theoretically accelerate cryptographic calculations by directly supporting operations in time-memory tradeoffs, whilst being aided by faster clock speeds and multi-core capabilities (Intel, 2022).

Growing up in a technology-centric society, I've developed an interest cryptographic security, whilst building my own computers and dabbling in Hashcat and John. Combining these two interests inspired me to conduct an **investigation into the relation between processing power and the speed of attacks against cryptographic hashing functions** and researching **to what extent do increasing processor speeds impact the efficiency of brute-force attacks against Merkle-Damgård-based cryptographic hashing functions?** Previous researched delved into research of CHF efficiency and collisions best-case efficiencies against Merkle-Damgård-based CHFs on different systems and their processing powers, there has been a lack of research into brute-force attacks and their respective efficiency in brute-force attacks against Merkle-Damgård-based CHFs. This EE discusses brute-force methods and construction of CHFs with examples and explanation of their methods, followed by a discussion of CPU elements in terms of CHFs.

2 Background Information

2.1 Cryptographic Hash Functions (CHFs)

CHFs are one-way algorithms that take an input message (M) and produce a fixed-size output ($H(M)$), called a hash. $H(M)$ is originally binary, being converted into hexadecimal in the final calculation, represented as $H(M)$. $H(M)$ is unique to the M, but is built where $H(M)$ is designed to be computationally unfeasible to reverse. CHFs are typically used to provide a digital fingerprint of a file's contents, often used to verify the authenticity of a file, creating a trust factor between the sender and receiver as hash manipulation is a near impossible task (Kundakalesi, 2015).

A CHF's calculation depends on construction method and calculation steps. While a generalization of CHFs calculation is impossible due the variance of methods in such calculation, its properties are widely agreed upon (Rhodes, 2019) and as follows.

Computational Efficiency

$H(M)$ needs to be calculated in as short period of time, prioritizing practicality and speed. Calculation time of $H(M)$ decreases based on the processing capability of a system but increase based on amount of calculation operations required to calculate $H(M)$.

Deterministic

CHFs must always result in the same $H(M)$ given the same M. As mentioned before, $H(M)$ must be unique and any small changes in M's format must be different as well.

Impossible to Reverse-Engineer

M must be impossible to reproduce given only $H(M)$. This means that $H(M)$ cannot be replicated by reproducing steps that produce $H(M)$ in the first place.

Pre-Image & Collision Resistance

Two properties out of the scope of this EE, CHFs must not reveal details of M based on $H(M)$ (pre-image resistance) and $H(M)$ must **always** be unique of M (collision resistance).

2.1.1 Merkle-Damgård Construction

Merkle (1979) first described the Merkle-Damgård Construction. A widely-used construction method for building CHFs, using iterative processes where M is divided then expanded into blocks and compresses it into smaller fixed size output using a compression function. Each block is concatenated with the previous compression function's output to create the current compression function's input. The current compression's output and the previous compression produces a new "state" as its output. After the compression function, the resulting $H(M)$ is converted into hexadecimal.

Merkle (1979) offers a standard, but not solid, method of how a CHF calculates $H(M)$. It's possible for CHFs to deviate from the method calculation-wise not method-wise. A Merkle-Damgård CHF can use a different compression function but no longer a Merkle-Damgård CHF if blocks aren't concatenated. CHFs could divide and expand input blocks multiple times to further increase the impossibility to reverse-engineer, called *rounds*.

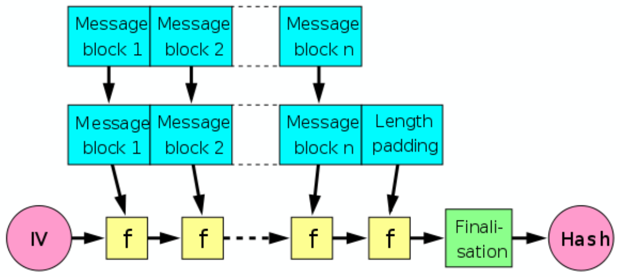


Figure 1: Illustration of the Merkle-Damgård Construction (Göthberg, 2007)

2.2 Brute-Force Attacks

A brute-force attack attempts to iterate every single possible plaintext until $H(M)$ is found, In a successful attack, the attacker finds the unknown message M through continuously hashing a changing string of text (Bosnjak et al., 2018). In such attacks, the attacker repeats attempts to find $H(M)$ using an M where in each iteration, the value of M is incremented by the next value alphanumerically. This is labeled as a *simple brute-force attack* (Knudsen & Robshaw, 2011). A simple brute-force attack can vary per attack. We can define two types of simple brute-force attacks: a "head-first" brute-force attack and a "tail-first" brute-force attack.

A "tail-first" brute-force attack works where M attempts to iterate the final character in a string.

$$M_n = AAAAA$$

the next "tail-first" iteration becomes

$$M_{n+1} = AAAAB$$

Different to a "head-first" brute-force attack where M attempts to iterate the root character instead.

$$M_n = AAAAA$$

the next "head-first" iteration becomes

$$M_{n+1} = BAAAA$$

Practically, if a range of numerical, lowercase and uppercase values are used, an actual password is more likely to have a higher value at the end. Wodinsky (2021) found that the most common password used in breached or leaked accounts is "123456", followed closely by "123456789", and "12345". The complexity of brute-force attacks, regardless of type, always is 2^L where L is digest bit length. Thus, MD4 has a brute-force efficiency of 2^{128} and so on.

2.3 CHF Algorithms

2.3.1 MD4 (Rivest, 1990)

Rivest's (1990) MD4 hashing algorithm offered a fast and efficient CHF to create hashes. A 128-bit digest size, represented in a 32-character hexadecimal output and based on a modified version of the Merkle-Damgård construction method where an iterative design exists to replicate AND, OR, XOR, and ROT bitwise operations to "mix" message bits as the final calculation. Boasting 1,450,000 bytes/second given its comparatively miniscule amount of rounds. MD4 consists of 48 of these bitwise operations, grouped in 3 rounds of 16 where each operation concatenates three 32-bit block messages with each block undergoing 5 ROTs. This suggests that from a 36-character length M, it'll take an average of 0.0248 ms (Rivest, 1990) to calculate the MD4 $H(M)$.

2.3.2 MD5 (Rivest, 1992)

Rivest (1992) released the widely-used MD5 and was designed as an extension of MD4 in a trade-off of speed for security. While similar to MD4, MD5 offered a fourth round of operations with each step now having a unique additive constant with a replacement in operator from OR to NOT in round 2 to make MD5 less symmetrical than MD4 and with optimized bitwise shifts. As a result, this promotes a faster avalanche effect to further individualize resulting hashes but had the intentional effect of reducing hashing time efficiency but results in hashes with similar properties to MD4. It takes an average of 627 ms to calculate $H(M)$ of MD5 of a 36-character length M: 25,300 times slower than MD4 (Latinov, 2018).

2.3.3 SHA-1 (NIST, 1995)

With a 160-bit digest size, SHA-1 (NIST, 1995) is represented in a 40-character hexadecimal output, generated from a computationally massive 80 operational rounds, providing a comparatively larger output compared to previous Merkle-Damgård-based CHFs. The initial block, 512 bits in length, is split into 80 32-bit values. For 80 rounds in a SHA-1 operation, it is divided into 4 groups of 20 where each group has a different constant. Each 32-bit value goes under 20 rounds per 4 groups and results in 4 parts of the digest which are concatenated to receive $H(M)$. Similar to MD5's unique additive constant, SHA-1 employs 4 different 32-bit constants for each round. This has the effect of creating a further, non-linear resulting $H(M)$ and enhance its avalanche effect. SHA-1 takes an average of 604 ms to calculate $H(M)$ of SHA-1 of a 36-character length M, only 27 ms slower than MD5 (Latinov, 2018).

2.3.4 SHA-256 (NIST, 2001)

In 2001, SHA-256 was introduced by the NIST. Part of the SHA-2 family, it provides a robust and secure method for generating hash outputs based on a Merkle-Damgård construction **but** using Davies-Meyer compression function. Its larger digest size from 224-bits to 512-bits (256-bits in SHA-256 and represented in a 64-character hexadecimal output), provides a larger output space to make it computationally infeasible to calculate M through brute-forcing. Increased operation rounds at 64 rounds in SHA-256 and improved message expansion which, combined with 64 operation rounds and more complex logical functions, causes a greater avalanche effect. Though, due to the computational intensity of SHA-2 algorithms, it's expected that SHA-256 will take longer to generate hashes compared to the previous 3 CHFs. This is proven by an average of 737.8 ms (Latinov, 2018) to calculate $H(M)$ of SHA-256 of a 36-character length M, almost two-tenths of second more than MD5.

2.4 CPU Architecture

The Central Processing Unit (CPU) chip performs arithmetic operations, logical operations in order to process data from input devices into an output or to storage (Kostas & Markos, 2015). Based on von Neumann architecture - where instructions are stored in RAM and are fetched, decoded by the CPU and being executed. A von Neumann CPU contains two components: the Control Unit, containing registers and the Arithmetic Logic Unit, performing arithmetic and logical operations.

The functions and architecture of the CPU have seldom changed in the past 30 years, with changes mainly occurring in form factor and performance. A 2002 Intel Celeron 1000A (Intel, 2001) would still be based off von Neumann architecture with same registers and structure as Intel's 2023 flagship i9-12900KS (Intel, 2022) despite having 589 times less transistors, being 4GHz slower and physically being 9 times larger. In recent years, CPU architecture has changed where more processing units, ”cores”, are in one CPU, with each core being able to support developments such as parallel processing and more advanced instruction sets.

Although, instruction sets supported for CPU architectures have evolved constantly. Since 2008, the majority of x86-64 CPUs have supported SSE 4.x, AVX, MMX and AES. AES stands out as its latest release, AES-NI, was specially designed to support encryption standards by the NIST and has supported extra round instructions per given operation, decreasing calculation time (Gueron, 2010).

2.4.1 Multicore Processing & CPU Architecture

The early 2000s introduced hardware implementations of multi-core CPUs where a ”core” is a processor and their respective caches. Multi-core CPUs are connected through a series of buses with a shared, increased bandwidth to accommodate those multiple cores (Schauer, 2008). Compared to single cores, multi-cores have lower frequencies - resulting in lower power consumption - explaining the mid-2000s plateau in clock speed (Swanson, 2015). Multi-core CPUs have somewhat ”enhanced” the Von-Neumann cycle, with n cores executing instructions n times simultaneously.

Furthermore, parallel processing aims to carry out processes simultaneously to get the most possible performance possible from a processor (Schauer, 2008). As opposed to the Von-Neumann cycle, parallel processing executes multiple tasks simultaneously by dividing processes and assigning them to different cores.

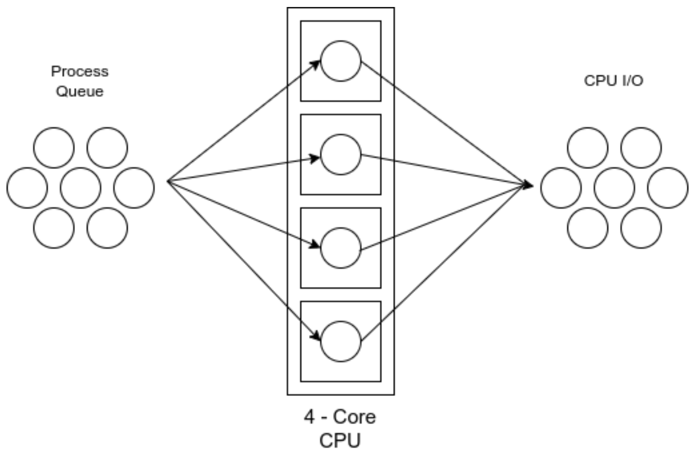


Figure 2: An illustration of parallel processing on four cores.

The main process is first divided into subtasks, or instructions that can be executed independently. Next, instructions are assigned to the available cores and as dependencies are present in subtasks being executed, synchronization is required to maintain a processing order. Processes are then independently executed for each core. Once all subtasks from the main process have been executed, the final result is output.

Modern high-level languages (HLLs) support parallel processing through implementations of multiprocessing libraries which offer methods for creating and managing threads and synchronizing processes without blocking the main process. Furthermore, despite a variation in handling, parallelism is supported through enabling multiprocess spawning and distributing processes in such (Prabhat et al., 2012). This suggests a new paradigm of parallel programming that could be implemented to enhance brute-force operations on multi-core CPUs.

2.4.2 Defining "Increasing Efficiency"

Computing efficiency is the amount of computational resources an algorithm requires. Efficiency varies as different systems have a differing amount of computational resources. It was established earlier that complexity is the amount of time needed per iteration, thus efficiency regards time varying with resources, complexity regards general time.

Clock speed is defined as the rate at which a CPU performs operations. Measured in Hertz (Hz), a 1 GHz CPU processes 1 billion clock cycles per second. Clock cycles can execute n instructions per cycle (IPC) where $n \in \mathbb{R}_{>0}$, therefore higher clock speeds means higher IPC, suggesting that as clock speed increases, the time taken for a program to iterate through M decreases. So as clock speed increases per core, efficiency increases.

Processing speed, measured in Floating point operations per second (FLOPs), represents the computational capability of processors, including CPUs (Fernandez, 2015). A standardized method to calculate the GigaFLOPs (GFLOPs) of all CPU is through the Single Precision General Matrix Multiplication (SGEMM) (IBM, 2021). Its expected that an increase in cores and cycles per second, processing speed increases, signaling increasing efficiency.

3 Methodology

3.1 Method Rationale

The method will involve running various attack codes against increasing M lengths hashed in different CHFs on computers with increasing GFLOPs.

HLLs which offer low-level memory management such as Rust offer relatively fast speeds compared to other HLLs and offer architectural agnosticism and a standard rather than working with pure machine code - as Rust requires compilation directly into machine code with LLVM in order to be executed (Ng, 2023). Thus, throughout the experiment, Rust ensures CPU-direct access and speed compilation. To satisfy the **extent of increasing processor speeds impacting the efficiency of brute-force attacks**, the CPU should be the only **hardware** independent variable. Furthermore, as this EE attempts to calculate said efficiency of brute-force attacks, factors that could throttle performance; RAM (Random Access Memory) speed and size, storage read-write speed and GPU must stay consistent throughout to ensure a fair experiment - GPU usage must be prevented during attack code execution.

In terms of CHFs, time taken will be calculated for a brute-force attack to be executed given $H(M)$ for MD4, MD5, SHA-1 and SHA-2. A tail-first simple brute-force attack will be utilized in order to generate M while a concurrent hashing function will hash M, then convert into hexadecimal, in order to generate the corresponding $H(M)$. $H(M)$ will then be compared to the user-input $H(M)$ to get the value.

To harvest primary data, `cargo run --release` will be executed for Rust code to compile source code files. Its compiler, `rustc`, optimizes Rust code if the parameter `--release` is declared during compilation (Huss, 2023). As time taken of program execution is the dependent variable, the Linux/UNIX command `time` to display execution time of a compiled program. `time`'s `real` output will be taken to signify the real-world time taken (Kerrisk & Colomar, 2023) for $H(M)$ to be generated. There will be 3 random, pre-generated, alphanumeric 6, 8, 10 & 12-character $H(M)$.

To ensure a fair experiment, hardware factors such as benchmarking machine should have the same specifications. As it's physically infeasible to have the same system support different CPU form factors from 2011 to 2021, adjustments will be made per system to ensure only CPU is the independent factor while using BLAS to calculate GFLOPs. The same set of data will also be used for brute-force where the hash is pre-calculated and used as the input then compared to a continuously-being-hashed value.

3.2 Hypothesis

If processing power increases, then an attack against a message decreases where M hashing and $H(M)$ matching time decreases. Given a set of CPUs with exponentially increasing speeds, it's expected that compile program execution times increase exponentially as the CPU's GFLOPs decreases and vice versa. In terms of the 4 separate CHFs, it's predicted that in the set of different M length, SHA-256 will take the longest to brute-force while MD4 will take the least amount of time, followed closely by MD5 as both have a digest of 128-bits, leaving SHA-1 being only slightly faster than MD5 due to its 160-bit digest size and proven, but slight, computational speed advantage. Although, its 80 operational rounds could push its time closer to SHA-256 especially that Latinov (2016) was carried out though JVM bytecode and not directly onto the CPU machine code. This length-basis hypothesis also extends to the generated Ms where its predicted that the 6-character length Ms will take significantly less time than 12-character Ms throughout the different CPUs and CHFs.

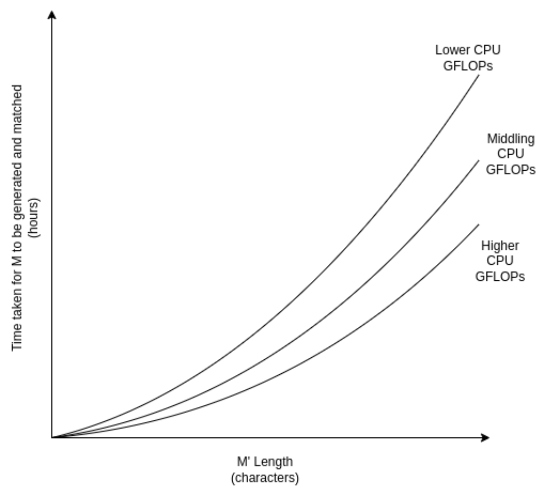


Figure 3: The hypothetical graph of results.

3.3 Variables

3.3.1 Independent Variables

CPU:

- All CPUs are x86-64 Intel & AMD.
- Clock speed will be set at base clock speeds.
- Mobile CPUs will only be used if their laptops allow for RAM upgrades.

Message Length:

- Variations in M with lengths 6, 8, 10 & 12 and their hashes.

Tail/Head ordering:

- All tests will be run both tail-first and head-first code.

3.3.2 Dependent Variable

Time taken for a successful attack (in hours)

3.3.3 Control Variables

1. Cryptographic Hashing Algorithm
2. Message and Hash Value
3. Attack Code
    - 5 tail-first and head-first brute-force code times of the procedure, will be taken per hash function to receive a mean time per length.
4. GPU and Memory Specifications
    - The GPU used throughout will vary by CPU though any GPU calls will be blocked by the program.
    - Memory used will be either DDR3L or DDR4 dual-channel 3200Mhz 8GBx2 RAM.
5. Operating System
    - All tests will be performed on a fresh, headless install of Arch Linux (kernel 6.4.1) on a 128GB Live USB Environment, with `rustc`.

3.4 CPU Units

CPU	**Cores**	**Threads**	**Clock** (GHz)	**Die Size** (mm^2)	**Year**	**GFLOPs**
Intel i7 2600K	4	8	3.40	216	2011	163.4
Intel i5 4200M	2	4	2.50	131	2013	94.7
Intel i7 5600U	2	4	2.60	82	2015	88.2
AMD Ryzen 5 1500X	4	8	3.50	213	2017	179.5
AMD Ryzen 5 3600	6	12	3.60	74	2019	484
AMD Ryzen 7 5800H	8	16	3.20	180	2021	378.7

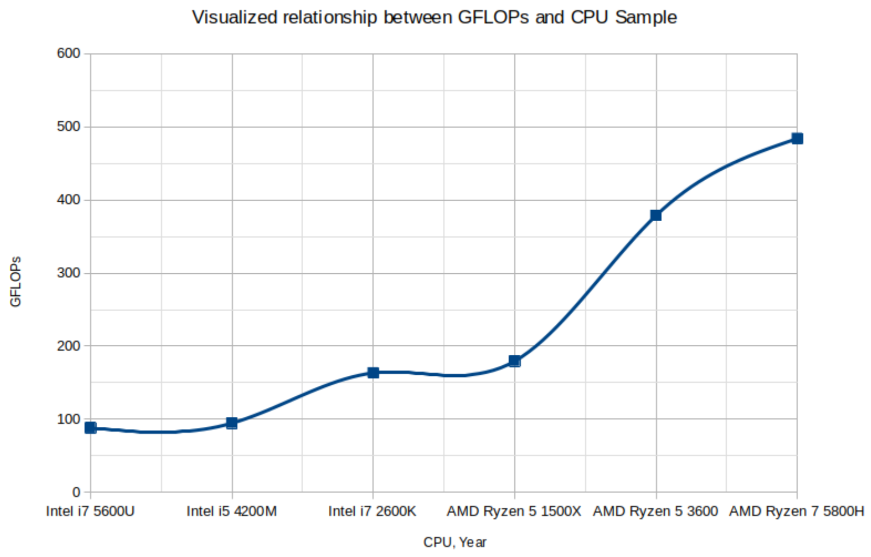
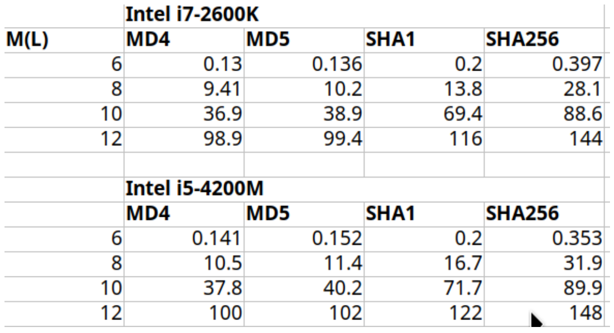


Figure 4: Visualized relation between GFLOPs and CPU test sample

4 Results

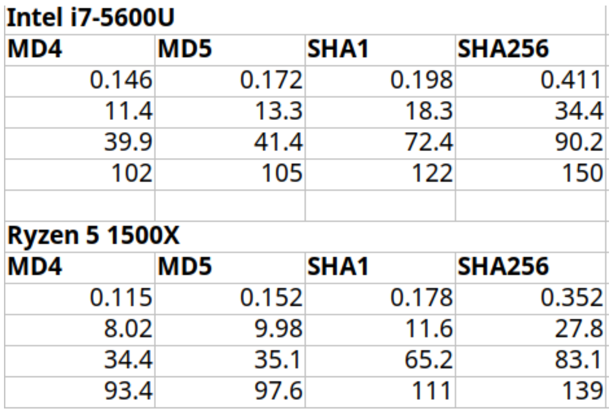
4.1 Formatted Data



	Intel i7-2600K			
M(L)	MD4	MD5	SHA1	SHA256
6	0.13	0.136	0.2	0.397
8	9.41	10.2	13.8	28.1
10	36.9	38.9	69.4	88.6
12	98.9	99.4	116	144

	Intel i5-4200M			
	MD4	MD5	SHA1	SHA256
6	0.141	0.152	0.2	0.353
8	10.5	11.4	16.7	31.9
10	37.8	40.2	71.7	89.9
12	100	102	122	148

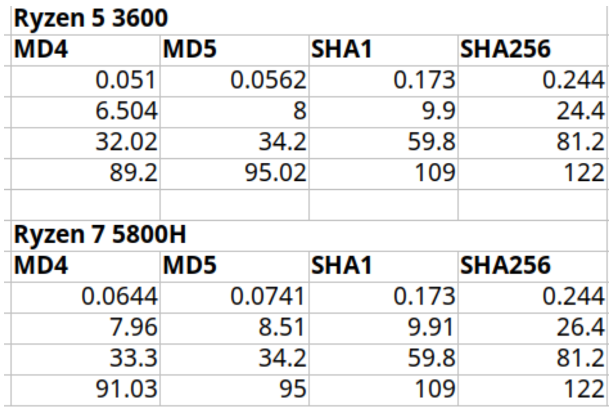
Figure 5: Average tail-first times (3 s.f.) [1/3]



Intel i7-5600U			
MD4	MD5	SHA1	SHA256
0.146	0.172	0.198	0.411
11.4	13.3	18.3	34.4
39.9	41.4	72.4	90.2
102	105	122	150

Ryzen 5 1500X			
MD4	MD5	SHA1	SHA256
0.115	0.152	0.178	0.352
8.02	9.98	11.6	27.8
34.4	35.1	65.2	83.1
93.4	97.6	111	139

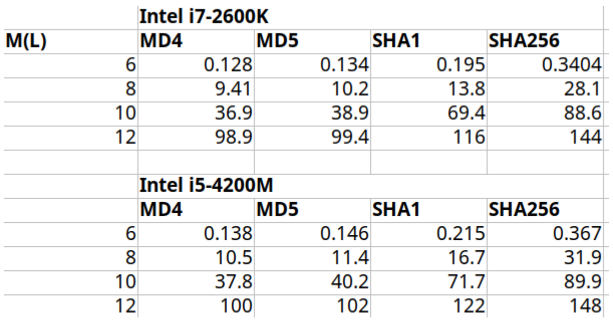
Figure 6: Average tail-first times (3 s.f.) [2/3]



Ryzen 5 3600			
MD4	MD5	SHA1	SHA256
0.051	0.0562	0.173	0.244
6.504	8	9.9	24.4
32.02	34.2	59.8	81.2
89.2	95.02	109	122

Ryzen 7 5800H			
MD4	MD5	SHA1	SHA256
0.0644	0.0741	0.173	0.244
7.96	8.51	9.91	26.4
33.3	34.2	59.8	81.2
91.03	95	109	122

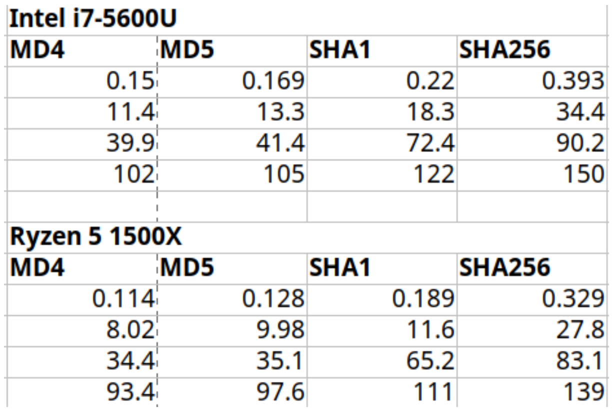
Figure 7: Average tail-first times (3 s.f.) [3/6]



	Intel i7-2600K			
M(L)	MD4	MD5	SHA1	SHA256
6	0.128	0.134	0.195	0.3404
8	9.41	10.2	13.8	28.1
10	36.9	38.9	69.4	88.6
12	98.9	99.4	116	144

	Intel i5-4200M			
	MD4	MD5	SHA1	SHA256
6	0.138	0.146	0.215	0.367
8	10.5	11.4	16.7	31.9
10	37.8	40.2	71.7	89.9
12	100	102	122	148

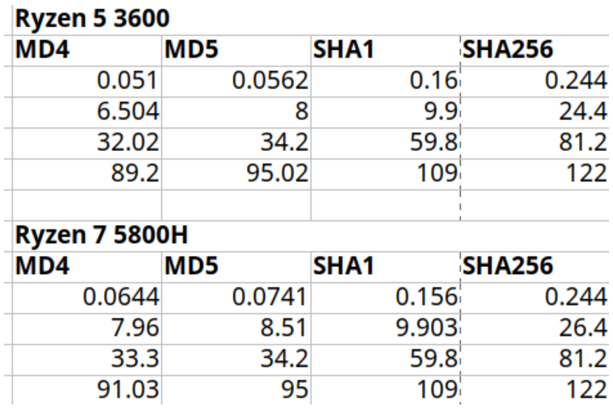
Figure 8: Average head-first times (3 s.f.) [1/3]



Intel i7-5600U			
MD4	MD5	SHA1	SHA256
0.15	0.169	0.22	0.393
11.4	13.3	18.3	34.4
39.9	41.4	72.4	90.2
102	105	122	150

Ryzen 5 1500X			
MD4	MD5	SHA1	SHA256
0.114	0.128	0.189	0.329
8.02	9.98	11.6	27.8
34.4	35.1	65.2	83.1
93.4	97.6	111	139

Figure 9: Average head-first times (3 s.f.) [2/3]

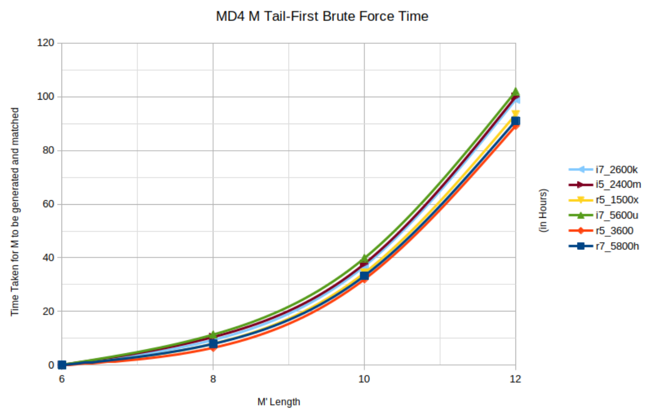


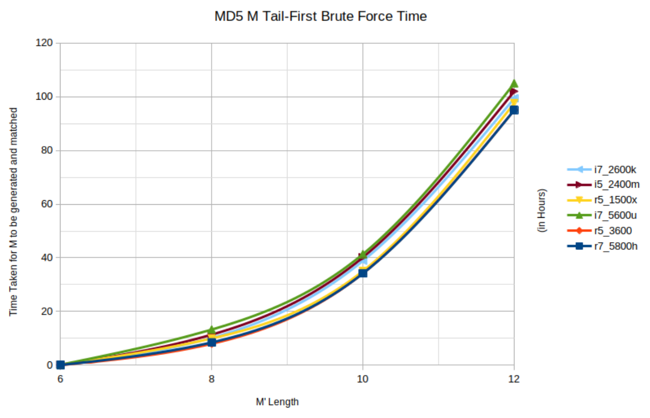
Ryzen 5 3600			
MD4	MD5	SHA1	SHA256
0.051	0.0562	0.16	0.244
6.504	8	9.9	24.4
32.02	34.2	59.8	81.2
89.2	95.02	109	122

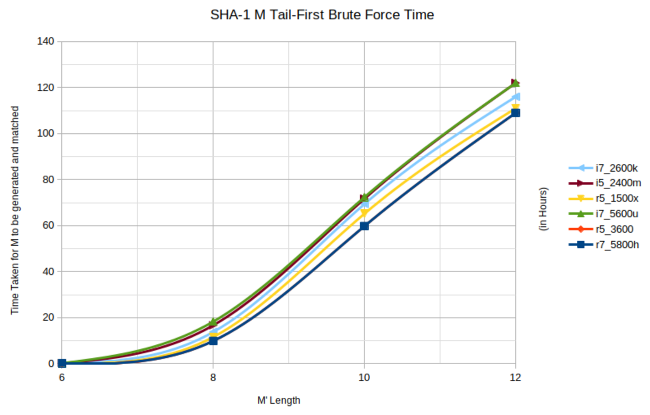
Ryzen 7 5800H			
MD4	MD5	SHA1	SHA256
0.0644	0.0741	0.156	0.244
7.96	8.51	9.903	26.4
33.3	34.2	59.8	81.2
91.03	95	109	122

Figure 10: Average head-first times (3 s.f.) [3/3]

4.2 Formatted Data Graphs







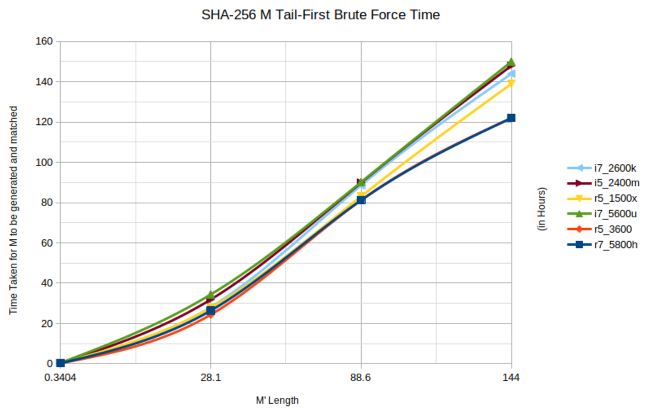
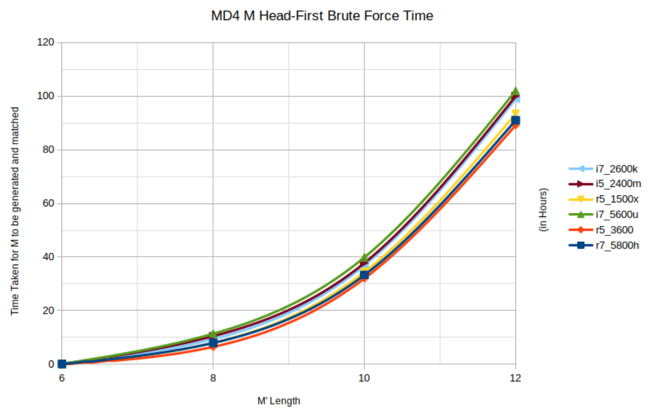
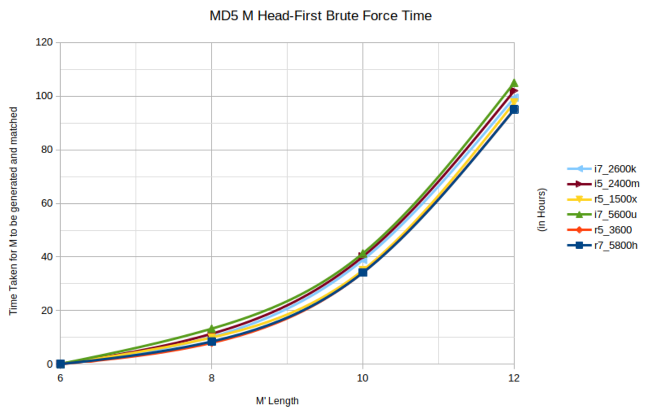


Figure 11: Tail-first brute-force algorithm results





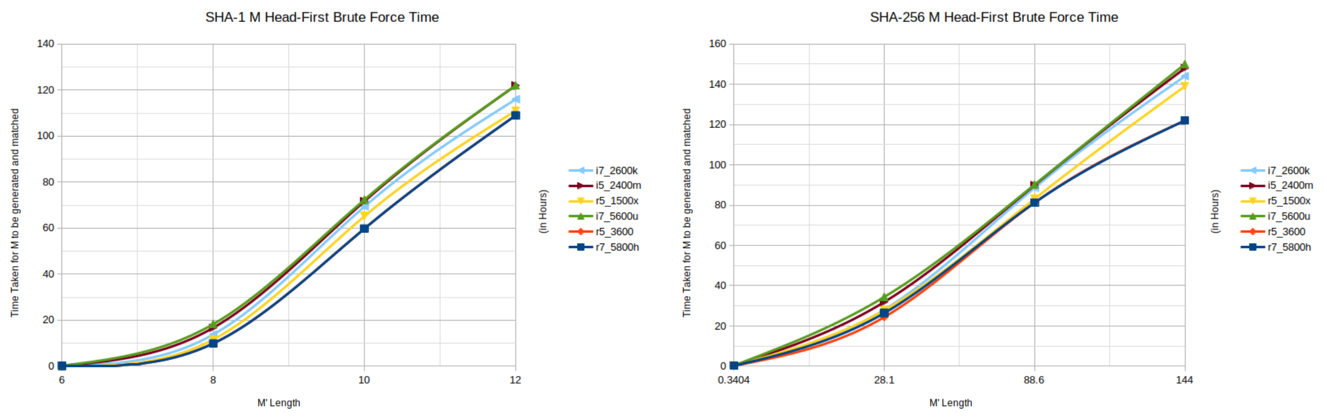


Figure 12: Head-first brute-force algorithm results

5 Analysis

Figures 5 to 10 represent average tail and head-first times of all $M'(L)$s along with their $H(M)$ and CPU while figures 11 and 12 represent plotted and graphed values of 5 and 6 per CHF. A recurring trend is seen where MD4 takes the shortest time, followed by MD5, SHA-1 (in contrast to SHA-1 purported as being slower than MD5 and SHA-256, regardless of $M'(L)$ or CPU. As $M'(L)$ increases though, the time taken for $H(M)$ to be generated and matched increases as well. For MD4, each CPU follows an exponential pattern. From $M'(L) = 6$, tail and head-first results are clustered around 0 hours. Looking at figure 5 and 6, MD4 with $M'(L) = 6$ does in fact vary from 0.051 hours to 0.138 hours head-first and 0.051 hours to 0.141 hours tail-first. Like with the aforementioned recurring trend, the exponential patten for time follows the trend for every subsequent CHF.

The significance between average result time also seems to increase as length increases. on a head-first MD4 test, with $M'(L) = 6$ and an i7-2600K, standard deviation of average time by length (σ_L) equals to 0.0856 while when $M'(L) = 12$, σ_{12} equals to 18.3. Tail-first tests do seem to follow the standard deviation increase as on the corresponding tail-first test set, $\sigma_6 = 0.108$ and $\sigma_{12} = 18.3$. Noting that the values for later $M'(L)$ for standard deviation do not vary much at all, with almost directly similar results for head-first and tail-first tests. The 2 most powerful CPUs in the experiment; the Ryzen 7 5800H and Ryzen 5 3600, appears to also have similar results. In their respective averages in figure 5 and 6's are equal when $M'(L) \geq 10$.

Graphically, figures 11 and 12 highlight another pattern between CPU units and CHFs. For MD4 and MD5, both tail-first and head-first follow an exponential pattern, where for every increasing $M(L)$ corresponds with time taken for $H(M)$ to be generated by $M'(L)^{(}L)$. For SHA-1 and SHA-256, however, there seems to be a non-exponential pattern for both tail-first and head-first SHA operations. Figure 11c and 12c highlight that for SHA-1, $M'(L)$ and time taken for $H(M)$ to be generated do have an exponential relationship up until $M'(L) = 10$. After, time for generation seems to plateau, in almost a logarithmic form, for the 3 Intel CPUs and Ryzen 5 1500X while the 2 remaining Ryzen CPUs converge and form a linear relationship. For SHA-256, CPUs with a recorded GFLOPs greater than 150 follow an exponential model up until around $M'(L) = 9$.

The Ryzen 5 1500X once again stands out as an outlier, forming a linear relationship. For the other 2 Ryzen CPUs, a more defined plateau formed. While the Ryzen 5 3600 undercut its fellow Ryzen 7 5800H at $M'(L) = 8$ in terms of time, both did converge and plateau together - confirming the aforementioned pattern. Between the i7-2600K and Ryzen 5 1500X, while boasting similar GFLOPs - at 163.4 and 179.5 respectively - the Ryzen 5 1500X did significantly better compared to its counterpart, despite varying by only 16.1 GFLOPs.

Lower clock speeds and fewer cores seem to demonstrate slower processing times across the board. This is evident in consistently longer results for slower CPU as $H(M)$ bit length increases, regardless of CHF. As $M'(L)$ increases, so does the time required for generating and matching $H(M)$. (σ_L) for head-first MD4 tests increase significantly from $M'(L) = 6$ to $M'(L) = 12$, highlighting a widening range of results. The more powerful CPUs - Ryzen 7 5800H and Ryzen 5 3600 - display similar performance levels for all but MD4. Additionally, outliers like the Ryzen 5 1500X in SHA-256 operations suggest performance variations even among CPUs with similar GFLOPs. However, lower clock speeds do not necessarily mean faster hashing speeds: CPUs with lower base clock and more logical cores perform better than higher base clock with less logical cores.

6 Evaluation

6.1 Method Weaknesses and Limitations

- Other Merkle-Damgård construction-based CHFs have not been tested. Such CHFs may yield varying levels of computational difficulty or ease - based on the respective calculation method, unveiling distinct performance traits not yet explored.

- The Ryzen 5 1500X results were tested on an external system and could have introduced an external variable, potentially influencing the results compared to other results.

- DDR3L RAM in the two lower-end Intel CPUs, may have influenced instruction execution speed, potentially impacting overall processing times.

- The target Ms given in the experiment table are randomly generated strings with no semblance of synonymity to any word. That said, a good portion of leaked passwords are based off nouns in the English dictionary (Wodinsky, 2021).

- Mobile, Desktop, and Server CPUs differ in processing power due to usage constraints. Mobile CPUs prioritize battery life, with lower clock speeds and cores (2-8). Desktop CPUs have higher clock speeds and more cores (4+), without power throttling. Server CPUs have lower base clock speeds (around 3.0 GHz) but boast a higher core count, typically 16 or more. Modern server CPUs also feature assembly instructions to expedite intense calculations and ensure even system process distribution (Zhu et al., 2016).

6.2 Possible Future Improvements

- No multiprocessing functionalities have been implemented in the code. Rust's prevents multiprocessing by default for stability (Huss, 2023). Rust libraries which force multiprocessing and threading, such as Rayon, could be used in further experimentation.

- Special characters and non-Latin characters have not been taken into consideration in hash generation.
- Salts and peppers, added before hashing, enhance security. A pepper is a shared secret suffix while a salt is unique per input. This extra complexity deters brute-force attacks, as attackers would need to know s and/or p to decipher M from $H(M)$. This personalization renders attacks fruitless when s and p are unknown (Rosulek, 2021).

7 Conclusion

Ultimately, the experiment sought to evaluate the relationship between processing power and the time taken for brute-force attacks against CHFs; this investigation has proved that an increase in processor speed does improve the efficiency of brute-force attacks based on Merkle-Dåmgard construction-based CHFs. In other words, as CPU processing power increased, the time taken for brute-force attacks decreased.

Method-wise, using Rust along with Linux significantly benefited the experiment, both ensuring a stable and reliable testing platform. However, the limitations are clear; the experiment focused on a specific set of CPUs following a two-year release schedule of varying CPU type and brand, which may not fully represent the diversity of CPU architectures. Further retests must be undertaken to explore why SHA functions exhibit a non-exponential pattern, potentially shedding light on unique computational characteristics within these algorithms or could have proved the capabilities of AES-NI. The SHA-2 family could also be tested separately with its varying digest sizes. When comparing tail-first and head-first approaches, it was observed that there was no significant difference in the time taken for the algorithms to generate $H(M)$ and match the target hash. This suggests that the choice of brute-force strategy may not be a critical factor in the efficiency of attacks against Merkle-Damgård-based CHFs.

Looking ahead, further investigations could encompass non-Merkle-Damgård-based functions, expanding the scope of analysis and providing valuable insights into a wider range of CHFs. Additionally, exploring multiprocessing functionalities through specialized libraries could enhance efficiency of brute-force attacks and offer new perspectives on cryptographic security. Future experimentation should consider different form factors (desktop, server, mobile) and explore the impact of varying clock speeds and core counts. Additionally, integration of more CHFs beyond the Merkle-Damgård construction would offer a more comprehensive understanding of the relationship between processing power and cryptographic security in general.

In conclusion, my results somewhat aligned with the initial hypothesis, affirming that an increase in CPU speed enhances the efficiency of brute-force attacks against Merkle-Damgård-based CHFs in an exponential format. However, not all Merkle-Damgård CHFs followed the expected hypothesis; highlighting the need for nuanced considerations, such as CPU architecture diversity and the exploration of different CHFs, to comprehensively address the evolving landscape of cryptographic security.

A Bibliography

.

Bosnjak, L., Sres, J., & Brumen, B. (2018). Brute-force and dictionary attack on hashed real-world passwords. International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO), 41(41), 1161–1166. https://doi.org/10.23919/MIPRO.2018.8400211

Chaves, R., Sousa, L., Sklavos, N., Fournaris, A., Kalogeridou, G., Kitsos, P., & Sheikh, F. (2016). Secure hashing: SHA-1, SHA-2, and SHA-3 (p. 382).

Intel (2001). Intel® Celeron® processor 1.00 GHz, 256K cache, 100 MHz FSB. Intel. https://ark.intel.com/content/www/us/en/ark/products/27165/intel-celeron-processor-1-00-ghz-256k-cache-100-mhz-fsb.html

Intel (2022). Intel® CoreTM i9-12900KS processor (30M cache, up to 5.50 GHz) - product specifications. Intel. https://www.intel.com/content/www/us/en/products/sku/225916/intel-core-i912900ks-processor-30m-cache-up-to-5-50-ghz/specifications.html

Dimitriou, K., & Hatzitaskos, M. (2015). Core computer science for th IB diploma program. Express Publishing.

Fernandez, M. R. (2014). Dell technologies ”nodes, sockets, cores and FLOPS, oh my archive mirror. In Dell. https://mdotfernandez.wordpress.com/2014/02/03/nodes-sockets-cores-and-flops-oh-my/

Göthberg, D. (2007). Merkle-damgard hash big. In Wikimedia. https://commons.wikimedia.org/wiki/File:Merkle-Damgard_hash_big.svg?uselang=en#Licensing

Gueron, S. (2010). Intel advanced encryption standard (AES) new instructions set. Intel. https://www.intel.com/content/dam/doc/white-paper/advanced-encryption-standard-new-instructions-set-paper.pdf

”H2g2bob”. (2007). SHA-1.svg. In Wikimedia. https://commons.wikimedia.org/wiki/File:SHA-1.svg

Halpern, M., Zhu, Y., & Reddi, V. J. (2016). Mobile CPU’s rise to power: Quantifying the impact of generational mobile CPU design trends on performance, energy, and user satisfaction. IEEE International Symposium on High Performance Computer Architecture (HPCA), 1(1), 64–76. https://doi.org/10.1109/HPCA.2016.7446054

Huss, E. (2023). The rust reference (Nightly Release: July 8th 2023). The Rust Foundation. https://doc.rust-lang.org/reference/

IBM. (2021). SGEMM, DGEMM, CGEMM, and ZGEMM (combined matrix multiplication and addition for general matrices, their transposes, or conjugate transposes). IBM.com.

Knudsen, L. R., & Robshaw, M. J. B. (2011). Brute force attacks. In The Block Cipher Companion (pp. 95–108). Springer Berlin Heidelberg. https://doi.org/10.1007/978-3-642-17342-4\%E2\%82\%85

"kockmeyer". (2007). SHA-2.svg. In Wikimedia. https://commons.wikimedia.org/wiki/File:SHA-2.svg

"KTC". (2008). MD4.svg. In Wikimedia. https://commons.wikimedia.org/wiki/File:MD4.svg

Kundalakesi, M. (2015). Overview of Modern Cryptography. International Journal of Computer Science and Information Technologies, 6(1), 1. https://www.researchgate.net/publication/320517674_Overview_of_Modern_Cryptography

Lyudmil Latinov. (2016). MD5, SHA-1, SHA-256 and SHA-512 speed performance — automation rhapsody. Automation Rhapsody. https://automationrhapsody.com/md5-sha-1-sha-256-sha-512-speed-performance/

Merkle, C. (1979). SECRECY, AUTHENTICATION, AND PUBLIC KEY SYSTEMS. http://www.ralphmerkle.com/papers/Thesis1979.pdf

Ng, V. (2023). Rust vs C++, a battle of speed and efficiency [Review of Rust vs C++, a battle of speed and efficiency]. https://doi.org/10.36227/techrxiv.22792553.v1

NIST. (2001). Descriptions of SHA-256, SHA-384, and SHA-512. National Security Agency. https://web.archive.org/web/20130526224224/https://csrc.nist.gov/groups/STM/cavp/documents/shs/sha256-384-512.pdf

Rhodes, D. (2019). Cryptographic hash functions explained: A beginner's guide. Komodo Academy — En. https://komodoplatform.com/en/academy/cryptographic-hash-function/#an-intro-to-one-way-hash-functions

Rivest, R. L. (1990). MD4 message digest algorithm (No. 1186) [RFC]. RFC Editor. https://doi.org/10.17487/RFC1186

Rivest, R. L. (1992). The MD5 message-digest algorithm (No. 1321) [RFC]. RFC Editor. https://doi.org/10.17487/RFC1321

Rivest, R., Shamir, A., & Adleman, L. (1978). A method for obtaining digital signatures and public-key cryptosystems. Communications of the ACM, 21(2). ACM Digital Library. https://dl.acm.org/doi/pdf/10.1145/359340.359342

Rosulek, M. (2021). Hash functions. In The Joy of Cryptography. National Science Foundation. `https://joyofcryptography.com`

Schauer, B. (2008). Multicore processors - a necessity (archived). In Internet Archive's Wayback Machine. ProQuest. `https://web.archive.org/web/20111125035151/http://www.csa.com/discoveryguides/multicore/review.pdf`

Specops Software. (2023). Weak password report 2023. Specops Software. `https://specopssoft.com/wp-content/uploads/2023/06/Specops-Software-Weak-Pwd-report-2023.pdf`

"Surachit". (2005). MD5.svg. Wikimedia. `https://en.wikipedia.org/wiki/MD5#/media/File:MD5_algorithm.svg`

Swanson, B. (2015). Moore's law at 50. American Enterprise Institute.

Colomar, A., Kerrisk, M. (2023). Linux user's manual (L. Torvalds , Ed.). Kernel.org. `https://www.man7.org/linux/man-pages/man1/man.1.html`

Totoo, P., Deligiannis, P., & Loidl, H.-W. (2012). Haskell vs. F vs. Scala: A high-level language features and parallelism support comparison. Heriot-Watt University.

Wodinsky, S. (2021). The 200 worst passwords of 2021 are here and oh my god. Gizmodo; Gawker Media. `https://gizmodo.com/the-200-worst-passwords-of-2021-are-here-and-oh-my-god-1848073946`

B Referential Images for Section 2

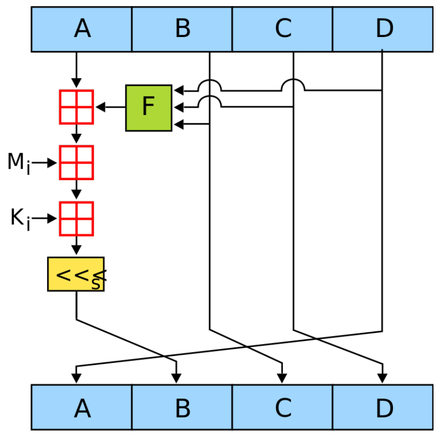


Figure 13: Illustration of MD4 (”KTC”, 2008)

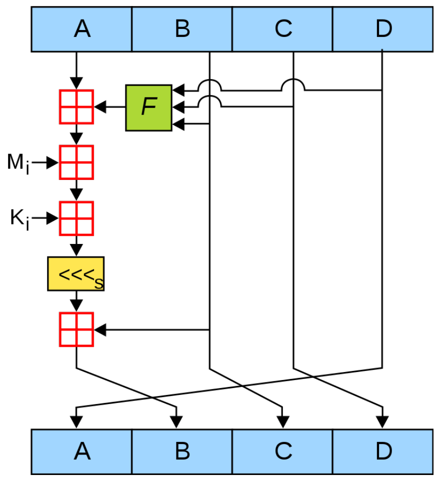


Figure 14: Illustration of MD5 (”Surachit”, 2005)

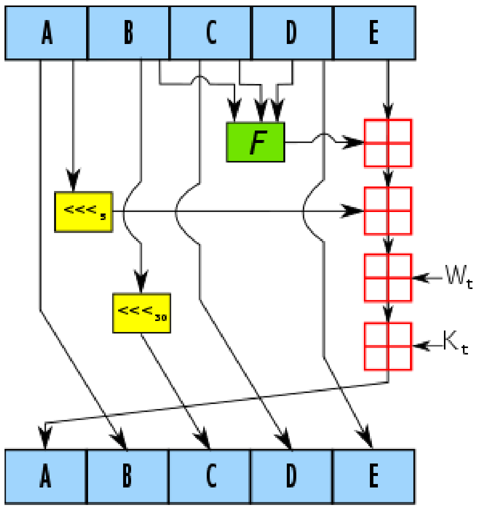


Figure 15: Illustration of SHA-1 ("H2g2bob", 2007)

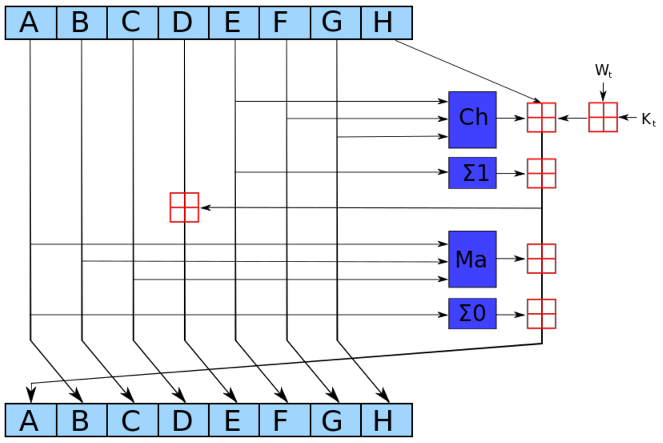


Figure 16: Illustration of SHA-2 family functions ("kockmeyer", 2007)

C Glossary

Compression Function	Maps variable-length input to a fixed-length output, preserving essential properties and ensuring a uniform output length.
Concatenation	The combination of two or more values
CHF	Cryptographic Hash Function.
Initial Value (IV)	The input of a CHF. Also known as a Message or M. For the sake of this extended essay, all Initial Values will be referred as M.
ROT	Rotation Function. An operation that shifts bits in a block by a given amount, varying per CHF
WORD	A 16-bit value. A 32-bit value could also be referred to as 32-bit WORD or DWORD. For the sake of this extended essay, all WORDs and DWORDs will be referred to a n-bit value.

D Code For Attack Algorithms

MD4 Tail-First Brute-force

```
use md4::{Md4, Digest};
use hex;

fn hash_password<D: Digest>(password: &str, output: &mut [u8]) {
    let mut hasher = D::new();
    hasher.update(password.as_bytes());
    output.copy_from_slice(&hasher.finalize());
}

fn increment_password(password: &mut Vec<char>, charset: &[char]) {
    let mut i = password.len() - 1;
    loop {
        let index = charset.iter().position(|&c| c == password[i]).unwrap();
        if index == charset.len() - 1 {
            password[i] = charset[0];
            if i == 0 {
                password.insert(0, charset[0]);
                break;
            }
            i -= 1;
        } else {
            password[i] = charset[index + 1];
            break;
        }
    }
}

fn main() {
    let target_hash = ""; \\INSERT TARGET HASH HERE
    let mut current_hash = String::new();
    let mut password = vec!['0'];

    let charset: Vec<char> = "0123456789abcdefghijklmnopqrstuvwxyz
    ABCDEFGHIJKLMNOPQRSTUVWXYZ".chars().collect();

    while current_hash != target_hash {
        let password_str: String = password.iter().collect();
        let mut buf = [0u8; 16];
        hash_password::<Md4>(&password_str, &mut buf);

        current_hash = hex::encode(&buf);
        println!("{}: {}", password_str, current_hash);

        if current_hash == target_hash {
            println!("hash found");
            break;
        }

        increment_password(&mut password, &charset);
    }
}
```

MD4 Head-First Brute-force

```
use md4::{Md4, Digest};
use hex;

fn hash_password<D: Digest>(password: &str, output: &mut [u8]) {
    let mut hasher = D::new();
    hasher.update(password.as_bytes());
    output.copy_from_slice(&hasher.finalize());
}

fn increment_password(password: &mut Vec<char>, charset: &[char]) {
    let mut i = 0;
    loop {
        let index = charset.iter().position(|&c| c == password[i]).unwrap();
        if index == charset.len() - 1 {
            password[i] = charset[0];
            if i == password.len() - 1 {
                password.push(charset[0]);
                break;
            }
            i += 1;
        } else {
            password[i] = charset[index + 1];
            break;
        }
    }
}

fn main() {
    let target_hash = ""; // INSERT TARGET HASH HERE
    let mut current_hash = String::new();
    let mut password = vec!['0'];
    let charset: Vec<char> = "0123456789abcdefghijklmnopqrstuvwxyz
    ABCDEFGHIJKLMNOPQRSTUVWXYZ".chars().collect();

    while current_hash != target_hash {
        let password_str: String = password.iter().collect();
        let mut buf = [0u8; 16];
        hash_password::<Md4>(&password_str, &mut buf);

        current_hash = hex::encode(&buf);

        println!("{}: {}", password_str, current_hash);

        if current_hash == target_hash {
            println!("hash found");
            break;
        }

        increment_password(&mut password, &charset);
    }
}
```

MD5 Tail-First Brute-force

```
use md5::{Md5, Digest};
use hex;

fn hash_password<D: Digest>(password: &str, output: &mut [u8]) {
    let mut hasher = D::new();
    hasher.update(password.as_bytes());
    output.copy_from_slice(&hasher.finalize());
}

fn increment_password(password: &mut Vec<char>, charset: &[char]) {
    let mut i = password.len() - 1;
    loop {
        let index = charset.iter().position(|&c| c == password[i]).unwrap();
        if index == charset.len() - 1 {
            password[i] = charset[0];
            if i == 0 {
                password.insert(0, charset[0]);
                break;
            }
            i -= 1;
        } else {
            password[i] = charset[index + 1];
            break;
        }
    }
}

fn main() {
    let target_hash = ""; \\INSERT TARGET HASH HERE
    let mut current_hash = String::new();
    let mut password = vec!['0'];

    let charset: Vec<char> = "0123456789abcdefghijklmnopqrstuvwxyz
    ABCDEFGHIJKLMNOPQRSTUVWXYZ".chars().collect();

    while current_hash != target_hash {
        let password_str: String = password.iter().collect();
        let mut buf = [0u8; 16];
        hash_password::<Md5>(&password_str, &mut buf);

        current_hash = hex::encode(&buf);

        println!("{}: {}", password_str, current_hash);

        if current_hash == target_hash {
            println!("hash found");
            break;
        }

        increment_password(&mut password, &charset);
    }
}
```

MD5 Head-First Brute-force

```
use md5::{Md5, Digest};
use hex;

fn hash_password<D: Digest>(password: &str, output: &mut [u8]) {
    let mut hasher = D::new();
    hasher.update(password.as_bytes());
    output.copy_from_slice(&hasher.finalize());
}
```

```
fn increment_password(password: &mut Vec<char>, charset: &[char]) {
    let mut i = 0;
    loop {
        let index = charset.iter().position(|&c| c == password[i]).unwrap();
        if index == charset.len() - 1 {
            password[i] = charset[0];
            if i == password.len() - 1 {
                password.push(charset[0]);
                break;
            }
            i += 1;
        } else {
            password[i] = charset[index + 1];
            break;
        }
    }
}

fn main() {
    let target_hash = ""; // INSERT TARGET HASH HERE
    let mut current_hash = String::new();
    let mut password = vec!['0'];

    let charset: Vec<char> = "0123456789abcdefghijklmnopqrstuvwxyz
    ABCDEFGHIJKLMNOPQRSTUVWXYZ".chars().collect();

    while current_hash != target_hash {
        let password_str: String = password.iter().collect();
        let mut buf = [0u8; 16];
        hash_password::<Md5>(&password_str, &mut buf);

        current_hash = hex::encode(&buf);

        println!("{}: {}", password_str, current_hash);

        if current_hash == target_hash {
            println!("hash found");
            break;
        }

        increment_password(&mut password, &charset);
    }
}
```

SHA-1 Tail-First Brute-force

```
use sha1::{Sha1, Digest};
use hex;

fn hash_password<D: Digest>(password: &str, output: &mut [u8]) {
    let mut hasher = D::new();
    hasher.update(password.as_bytes());
    output.copy_from_slice(&hasher.finalize());
}

fn increment_password(password: &mut Vec<char>, charset: &[char]) {
    let mut i = password.len() - 1;
    loop {
        let index = charset.iter().position(|&c| c == password[i]).unwrap();
        if index == charset.len() - 1 {
            password[i] = charset[0];
            if i == 0 {
                password.insert(0, charset[0]);
                break;
            }
            i -= 1;
        } else {
            password[i] = charset[index + 1];
            break;
        }
    }
}

fn main() {
    let target_hash = ""; \\INSERT TARGET HASH HERE
    let mut current_hash = String::new();
    let mut password = vec!['0'];

    let charset: Vec<char> = "0123456789abcdefghijklmnopqrstuvwxyz
    ABCDEFGHIJKLMNOPQRSTUVWXYZ".chars().collect();

    while current_hash != target_hash {
        let password_str: String = password.iter().collect();
        let mut buf = [0u8; 20];
        hash_password::<Sha1>(&password_str, &mut buf);

        current_hash = hex::encode(&buf);

        println!("{}: {}", password_str, current_hash);
        if current_hash == target_hash {
            println!("hash found");
            break;
        }

        increment_password(&mut password, &charset);
    }
}
```

SHA-1 Head-First Brute-force

```
use sha1::{Sha1, Digest};
use hex;

fn hash_password<D: Digest>(password: &str, output: &mut [u8]) {
    let mut hasher = D::new();
    hasher.update(password.as_bytes());
    output.copy_from_slice(&hasher.finalize());
}

fn increment_password(password: &mut Vec<char>, charset: &[char]) {
    let mut i = 0;
    loop {
        let index = charset.iter().position(|&c| c == password[i]).unwrap();
        if index == charset.len() - 1 {
            password[i] = charset[0];
            if i == 0 {
                password.insert(0, charset[0]);
                break;
            }
            i -= 1;
        } else {
            password[i] = charset[index + 1];
            break;
        }
    }
}

fn main() {
    let target_hash = ""; \\INSERT TARGET HASH HERE
    let mut current_hash = String::new();
    let mut password = vec!['0'];

    let charset: Vec<char> = "0123456789abcdefghijklmnopqrstuvwxyz
    ABCDEFGHIJKLMNOPQRSTUVWXYZ".chars().collect();

    while current_hash != target_hash {
        let password_str: String = password.iter().collect();
        let mut buf = [0u8; 20];
        hash_password::<Sha1>(&password_str, &mut buf);

        current_hash = hex::encode(&buf);

        println!("{}: {}", password_str, current_hash);

        if current_hash == target_hash {
            println!("hash found");
            break;
        }

        increment_password(&mut password, &charset);
    }
}
```

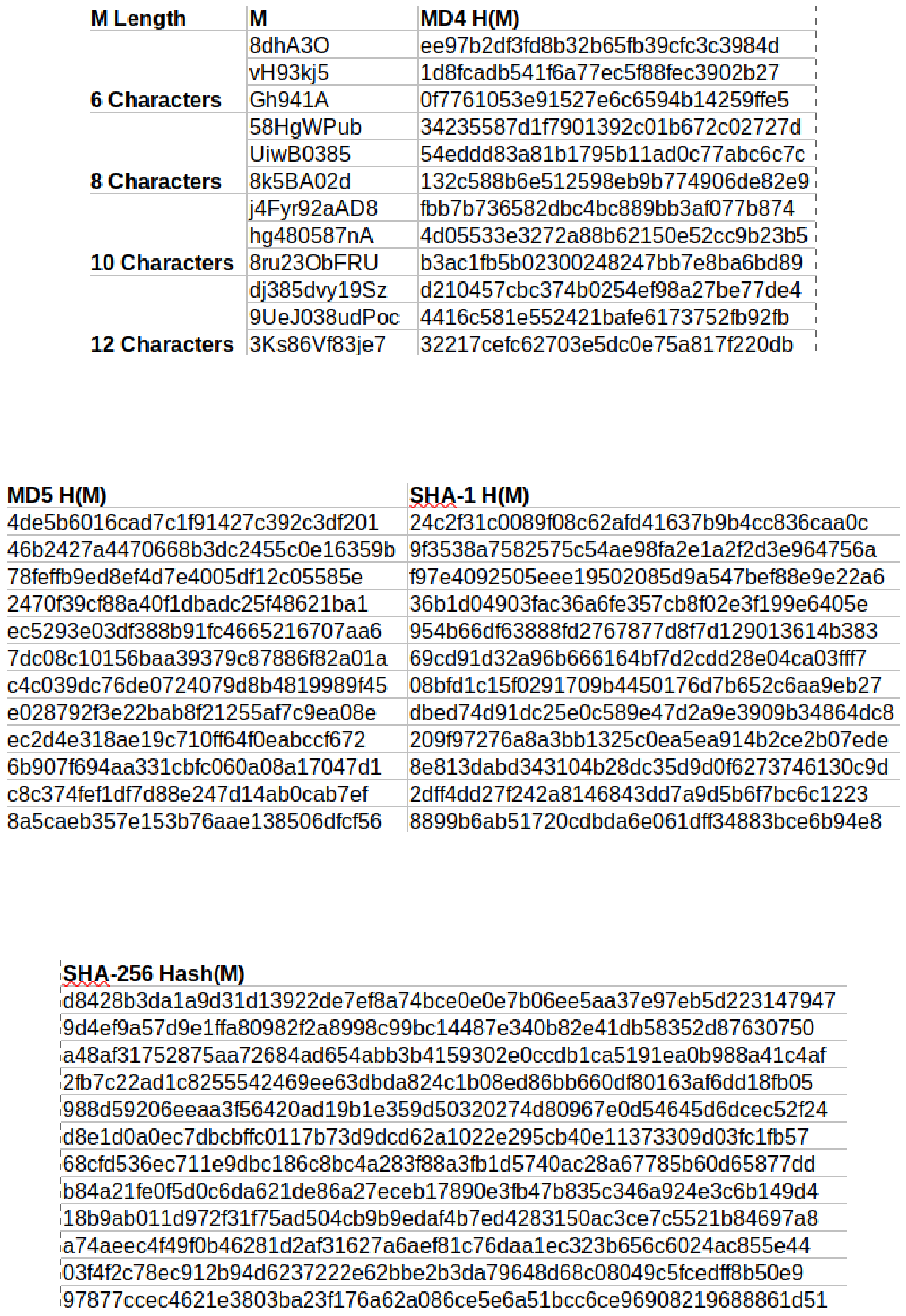
SHA-2 Tail-first Brute-force

```
use sha2::{Sha256, Digest};
use hex;

fn hash_password<D: Digest>(password: &str, output: &mut [u8]) {
    let mut hasher = D::new();
    hasher.update(password.as_bytes());
    output.copy_from_slice(&hasher.finalize());
}

fn increment_password(password: &mut Vec<char>, charset: &[char]) {
    let mut i = password.len() - 1;
    loop {
        let index = charset.iter().position(|&c| c == password[i]).unwrap();
        if index == charset.len() - 1 {
            password[i] = charset[0];
            if i == 0 {
                password.insert(0, charset[0]);
                break;
            }
            i -= 1;
        } else {
            password[i] = charset[index + 1];
            break;
        }
    }
}

fn main() {
    let target_hash = ""; \\INSERT TARGET HASH HERE
    let mut current_hash = String::new();
    let mut password = vec!['0'];

    let charset: Vec<char> = "0123456789abcdefghijklmnopqrstuvwxyz
    ABCDEFGHIJKLMNOPQRSTUVWXYZ".chars().collect();

    while current_hash != target_hash {
        let password_str: String = password.iter().collect();
        let mut buf = [0u8; 32];
        hash_password::<Sha256>(&password_str, &mut buf);

        current_hash = hex::encode(&buf);

        println!("{}: {}", password_str, current_hash);

        if current_hash == target_hash {
            println!("hash found");
            break;
        }

        increment_password(&mut password, &charset);
    }
}
```

SHA-2 Head-first Brute-force

```
use sha2::{Sha256, Digest};
use hex;

fn hash_password<D: Digest>(password: &str, output: &mut [u8]) {
    let mut hasher = D::new();
    hasher.update(password.as_bytes());
    output.copy_from_slice(&hasher.finalize());
}

fn increment_password(password: &mut Vec<char>, charset: &[char]) {
    let mut i = 0;
    loop {
        let index = charset.iter().position(|&c| c == password[i]).unwrap();
        if index == charset.len() - 1 {
            password[i] = charset[0];
            if i == 0 {
                password.insert(0, charset[0]);
                break;
            }
            i -= 1;
        } else {
            password[i] = charset[index + 1];
            break;
        }
    }
}

fn main() {
    let target_hash = ""; \\INSERT TARGET HASH HERE
    let mut current_hash = String::new();
    let mut password = vec!['0'];

    let charset: Vec<char> = "0123456789abcdefghijklmnopqrstuvwxyz
    ABCDEFGHIJKLMNOPQRSTUVWXYZ".chars().collect();

    while current_hash != target_hash {
        let password_str: String = password.iter().collect();
        let mut buf = [0u8; 32];
        hash_password::<Sha256>(&password_str, &mut buf);

        current_hash = hex::encode(&buf);

        println!("{}: {}", password_str, current_hash);

        if current_hash == target_hash {
            println!("hash found");
            break;
        }

        increment_password(&mut password, &charset);
    }
}
```

E Method Data



M Length	M	MD4 H(M)	MD5 H(M)	SHA-1 H(M)	SHA-256 Hash(M)
	8dhA3O	ee97b2df3fd8b32b65fb39cfc3c3984d	4de5b6016cad7c1f91427c392c3df201	24c2f31c0089f08c62afd41637b9b4cc836caa0c	d8428b3da1a9d31d13922de7ef8a74bce0e0e7b06ee5aa37e97eb5d223147947
	vH93kj5	1d8fcadb541f6a77ec5f88fec3902b27	46b2427a4470668b3dc2455c0e16359b	9f3538a7582575c54ae98fa2e1a2f2d3e964756a	9d4ef9a57d9e1ffa80982f2a8998c99bc14487e340b82e41db58352d87630750
6 Characters	Gh941A	0f7761053e91527e6c6594b14259ffe5	78feffb9ed8ef4d7e4005df12c05585e	f97e4092505eee19502085d9a547bef88e9e22a6	a48af31752875aa72684ad654abb3b4159302e0ccdb1ca5191ea0b988a41c4af
	58HgWPub	34235587d1f7901392c01b672c02727d	2470f39cf88a40f1dbadc25f48621ba1	36b1d04903fac36a6fe357cb8f02e3f199e6405e	2fb7c22ad1c8255542469ee63dbda824c1b08ed86bb660df80163af6dd18fb05
	UiwB0385	54eddd83a81b1795b11ad0c77abc6c7c	ec5293e03df388b91fc4665216707aa6	954b66df63888fd2767877d8f7d129013614b383	988d59206eeaa3f56420ad19b1e359d50320274d80967e0d54645d6dcec52f24
8 Characters	8k5BA02d	132c588b6e512598eb9b774906de82e9	7dc08c10156baa39379c87886f82a01a	69cd91d32a96b666164bf7d2cdd28e04ca03fff7	d8e1d0a0ec7dbcbffc0117b73d9dcd62a1022e295cb40e11373309d03fc1fb57
	j4Fyr92aAD8	fbb7b736582dbc4bc889bb3af077b874	c4c039dc76de0724079d8b4819989f45	08bfd1c15f0291709b4450176d7b652c6aa9eb27	68cfd536ec711e9dbc186c8bc4a283f88a3fb1d5740ac28a67785b60d65877dd
	hg480587nA	4d05533e3272a88b62150e52cc9b23b5	e028792f3e22bab8f21255af7c9ea08e	dbed74d91dc25e0c589e47d2a9e3909b34864dc8	b84a21fe0f5d0c6da621de86a27eceb17890e3fb47b835c346a924e3c6b149d4
10 Characters	8ru23ObFRU	b3ac1fb5b02300248247bb7e8ba6bd89	ec2d4e318ae19c710ff64f0eabccf672	209f97276a8a3bb1325c0ea5ea914b2ce2b07ede	18b9ab011d972f31f75ad504cb9b9edaf4b7ed4283150ac3ce7c5521b84697a8
	dj385dvy19Sz	d210457cbc374b0254ef98a27be77de4	6b907f694aa331cbfc060a08a17047d1	8e813dabd343104b28dc35d9d0f6273746130c9d	a74aeec4f49f0b46281d2af31627a6aef81c76daa1ec323b656c6024ac855e44
	9UeJ038udPoc	4416c581e552421bafe6173752fb92fb	c8c374fef1df7d88e247d14ab0cab7ef	2dff4dd27f242a8146843dd7a9d5b6f7bc6c1223	03f4f2c78ec912b94d6237222e62bbe2b3da79648d68c08049c5fcedff8b50e9
12 Characters	3Ks86Vf83je7	32217cefc62703e5dc0e75a817f220db	8a5caeb357e153b76aae138506dfcf56	8899b6ab51720cdbda6e061dff34883bce6b94e8	97877ccec4621e3803ba23f176a62a086ce5e6a51bcc6ce96908219688861d51

Figure 13: Method Data of Message in a Hashed Value Table