Master's thesis

Master's Programme in Computer Science

# Optimizing the computation of password hashes

Markus Andersson

August 20, 2023

Faculty of Science
University of Helsinki

**Contact information**

P. O. Box 68 (Pietari Kalmin katu 5)
00014 University of Helsinki,Finland

Email address: info@cs.helsinki.fi
URL: http://www.cs.helsinki.fi/

HELSINGIN YLIOPISTO – HELSINGFORS UNIVERSITET – UNIVERSITY OF HELSINKI

| Tiedekunta — Fakultet — Faculty | Koulutusohjelma — Utbildningsprogram — Study programme |
|---|---|
| Faculty of Science | Master's Programme in Computer Science |


Tekijä — Författare — Author

Markus Andersson


Työn nimi — Arbetets titel — Title

Optimizing the computation of password hashes

Ohjaajat — Handledare — Supervisors

Prof. V. Niemi

| Työn laji — Arbetets art — Level | Aika — Datum — Month and year | Sivumäärä — Sidoantal — Number of pages |
|---|---|---|
| Master's thesis | August 20, 2023 | 58 pages |

Tiivistelmä — Referat — Abstract


Using password hashes for verification is a common way to secure users' passwords against a potential data breach. The functions that are used to create these hashes have evolved and changed over time. Hackers and security researchers constantly try to find effective ways to derive the original passwords from these hashes.

This thesis focuses on cryptographic hash functions that get passwords as inputs and on the different methods an attacker may use to deduce a password from a hash. The research questions for the thesis are:

1. What kind of password hashing techniques have evolved from the viewpoints of a defender and an attacker?
2. What kind of observations can be made when studying the implementations of the hashing algorithms and the tools that the attackers use against the hashes?

The thesis examines some commonly used hash functions for passwords and common attack strategies that are used against them. Hash functions developed especially for passwords such as PBKDF2 and Scrypt will be explained. The password recovery tool Hashcat is introduced and different ways to use the tool against password hashes are demonstrated. Tests are done to show off differences in hash functions, as well as what kind of effect offensive and defensive techniques have against password hashes. These test results are explained and reviewed.



**ACM Computing Classification System (CCS)**
Security and privacy → Cryptography → Cryptanalysis and other attacks
Security and privacy → Cryptography → Symmetric cryptography and hash functions → Hash functions and message authentication codes
Security and privacy → Cryptography → Symmetric cryptography and hash functions → Block and stream ciphers

Avainsanat — Nyckelord — Keywords

Security, Passwords, Hash functions, Hashcat

Säilytyspaikka — Förvaringsställe — Where deposited

Helsinki University Library

Muita tietoja — övriga uppgifter — Additional information

Networking study track

HELSINGIN YLIOPISTO – HELSINGFORS UNIVERSITET – UNIVERSITY OF HELSINKI

| Tiedekunta — Fakultet — Faculty | Koulutusohjelma — Utbildningsprogram — Study programme |
|---|---|
| Faculty of Science | Master's Programme in Computer Science |

Tekijä — Författare — Author

Markus Andersson

Työn nimi — Arbetets titel — Title

Optimizing the computation of password hashes

Ohjaajat — Handledare — Supervisors

Prof. V. Niemi

| Työn laji — Arbetets art — Level | Aika — Datum — Month and year | Sivumäärä — Sidoantal — Number of pages |
|---|---|---|
| Master's thesis | 20.8.2023 | 58 pages |

Tiivistelmä — Referat — Abstract

Användning av lösenordshashar för verifiering är ett vanligt sätt att säkra användarnas lösenord i fall av eventuella informationsläckage. De funktioner som används för att skapa dessa hashar har utvecklats och förändrats under tidens gång. Både hackare och säkerhetsforskare försöker ständigt hitta effektiva sätt att härleda de ursprungliga lösenorden från dessa hashar.

Denna avhandling fokuserar på kryptografiska hashfunktioner som tar lösenord som inmatning, samt på de olika metoder en hackare kan använda för att härleda ett lösenord från en hash. Forskningsfrågorna för avhandlingen är:

1. Vilka typer av tekniker för lösenordshashning har utvecklats från synvinkeln av både en försvarare och en hackare?
2. Vilka observationer kan göras då man studerar verktygen som hackarna använder emot hashar och hur hashalgoritmerna har implementeras?

Avhandlingen undersöker hashfunktioner som i allmänhet används för lösenord och vanliga anfallstrategier som används emot dem. Hashfunktioner som har utvecklats speciellt för lösenord, såsom PBKDF2 och Scrypt, kommer att förklaras. Återhämtningsverktyget för lösenord Hashcat introduceras och olika sätt att använda verktyget mot lösenordshashar demonstreras. Tester utförs för att visa skillnader i hashfunktioner. Dessutom demonstreras effekten av offensiva och defensiva tekniker mot lösenordshashar. Testresultaten kommer att förklaras och granskas.

**ACM Computing Classification System (CCS)**
Security and privacy → Cryptography → Cryptanalysis and other attacks
Security and privacy → Cryptography → Symmetric cryptography and hash functions → Hash functions and message authentication codes
Security and privacy → Cryptography → Symmetric cryptography and hash functions → Block and stream ciphers

Avainsanat — Nyckelord — Keywords

Security, Passwords, Hash functions, Hashcat

Säilytyspaikka — Förvaringsställe — Where deposited

Helsinki University Library

Muita tietoja — övriga uppgifter — Additional information

Networking study track

# Contents


**1 Introduction** **1**

**2 Background** **2**
2.1 Motivation . . . 2
2.2 History . . . 3
2.3 Terminology . . . 4

**3 Common building blocks for password encryption** **9**
3.1 Secure Hash Algorithm 2 . . . 9
3.2 Hash-based Message Authentication Code . . . 14
3.3 PBKDF2 . . . 16
3.4 Salsa20 . . . 19

**4 Scrypt** **21**
4.1 Background . . . 21
4.2 Description of the scrypt function . . . 22
4.3 Usage in cryptocurrencies . . . 24

**5 Exploits and potential attacks** **25**
5.1 Hashcat . . . 25
5.2 SHA-256 weaknesses and exploits . . . 28
5.3 Optimizations for password cracking . . . 33
5.4 Hashcat kernels . . . 37
5.5 Scrypt strengths and defence strategies . . . 42
5.6 The effects of password length . . . 44
5.7 Cracking speed differences for hash functions . . . 48

**6 Discussions** **50**

**7 Conclusions** **53**

# 1 Introduction

Most people use passwords to secure and access different kinds of services and data. Traditionally, authorization to a system would be done by comparing an entered password to a previous stored correct password. If the entered and the stored passwords match, the user would be authorized to access the system. This kind of authorization has been deemed by many security experts to not be secure nowadays, which is why another system has been developed. A password hash, which is a seemingly random string of characters, is generated, by entering a password into a hash function. With the new password authorization system, the entered password is first entered into a hash function, whereafter the output hash is compared to exiting hashes. If the output hash and a stored hash match, the user would be authorized to access the system.

When the media talks about data breaches and how passwords have been leaked, they usually mean that password hashes have been compromised. The actual process of deriving a password from a password hash is not a simple one, but if successful, a hacker would get access to a user's password. This is why the computation of password hashes interest hackers and security researchers. Hackers want to find out the most effective way to compute these hashes, while researchers want to invent the best way to generate hashes that are as secure as possible.

This thesis aims to find out what kind of techniques both the attackers and the defenders have developed throughout the years. In addition to this, observations will be made on the implementation of these different approaches and techniques.

# 2 Background

In this chapter we will briefly go through the history of password usage, why we use passwords and what are the challenges with them. Explanations and definitions of different concepts and terms that will be used throughout the thesis will be given.

## 2.1 Motivation

Nowadays, the average person will have countless accounts and services to manage, in both their professional and private life, e.g., email, online banking, streaming services and games. Some services can share accounts, but most require that the user creates new ones. When these accounts are created, the user typically has to give a username and password. The ability to keep these accounts safe is crucial, since a data breach could have serious consequences. A severe scenario for a user would be, e.g., not being able to access their bank account. Since we use passwords in various places and losing them to an attacker can lead to serious consequences, one can start to appreciate the importance of password security.

Even though there has been a lot of advancements in password security throughout the years, people and companies still tend to repeat common mistakes. Even though people might claim that they are smart and use good passwords, data claim otherwise. Taking NordPass's 2022 research as an example, the top 10 most used passwords that year include: "password", "123456", "guest" and "querty" [22]. When looking for examples of companies being careless when it comes to security you do not have to look far. In 2019, Facebook and Instagram had a security breach, which leaked hundreds of millions of passwords, which were stored in plaintext, meaning that the passwords themselves were not protected in any way [21]. Another recent example of companies storing user credentials in plaintext would be the Thomson Reuters leak from 2022 [26].

In this thesis, we seek to study past and current password security methods, more specifically hash functions used on passwords. Points of interest would be to see what techniques have worked and not worked to secure passwords in the past and how these techniques have evolved or changed.

## 2.2 History

A password or passcode is something that a user typically uses to identify themselves. Passwords have been used in various ways throughout history, e.g., the Roman military used rotating watchwords which a person would have to know if they wanted to enter an area [27]. Ever since then, the usage of passwords and the need to keep secrets have changed and evolved. This has led to the development of more sophisticated and more complex security systems in order to keep our secrets safe. However, for a system to be widely accepted and used, it has to also be easy and convenient to use. The usage of passwords continues today and has stood the test of time.

When computers started to become more common and their usage increased, naturally the need for computer security emerged. When the first general-purpose time-sharing operating system CTSS was developed in the early 1960s, a problem emerged where multiple users files and data was available to all other users. The solution for this problem was to lock users' profiles and their files with a password. This worked well at the time, but did have its flaws as one of the first data breaches happened to this system. The system had a feature that allowed any user to physically print out any file from the system, which a user abused by printing out the file containing all the passwords that were stored on the system. Since the passwords where stored in plaintext on a database, this meant that when the attacker gained access to the database, they had effectively gained all passwords for that system and now had access to all functions, including administrator functions [15].

Even though when computers began to be more common, there was initially not much concern for computer security in the 1970s, since the use of the internet had not become common. The way password functionality was done originally in Unix systems, was that passwords, along with other login information were initially stored in a "/etc/passwd" file. Passwords could in this case be encrypted with the DES algorithm. This file could still be accessed and read by anyone, meaning that if the passwords were not encrypted, anyone could make a new account on the computer and then proceed to look up all the other users' passwords. Still, even if the passwords were encrypted, there had already been studies proving that one could break a DES encryption in about 12 hours of computation time in the 1970s [6]. These security concerns of the Unix systems were addressed when the "/etc/shadow" file was introduced. When in use, passwords were removed from the "/etc/passwd" file, where instead were references to the "/etc/shadow" file. The passwords

were encrypted here and allowed multiple different encryption algorithms to be used to encrypt the passwords. Additionally, the file could only be accessed by a user with root permissions [8]. The success of this system can be stated by the fact that it is still used today in, e.g., Linux systems.

Nowadays, the amount of cyberattacks on systems and databases is huge, and since quite a few of these cyberattacks are very sophisticated, having passwords saved as plaintext in your database is completely unacceptable. This is why it is very important that we add additional layers of security to our passwords when they are saved to databases. A common type of password security technique today is that instead of storing passwords in a database, hashes (will be explained in the next section) of passwords are stored instead. One of the earliest examples of using password hashes is the login passwords on Unix operating systems in the 1970s [17].

## 2.3 Terminology

In this section we will go through concepts and terminology of security and cryptography, that will be used in the thesis. The goal is to give the reader basic understanding of these concepts.

**Character:** When we talk about characters in this thesis, we mean symbols, such as letters, numbers, dots or exclamation marks, which can be typed on a keyboard. The relevance of this, is that passwords are made out of characters.

**Password:** A password is a string of characters chosen by a user which is used to usually secure a account or data. Generally, when trying to access a computer system, the user is asked to give a password, whereafter they are granted access if they input the correct password, otherwise they are denied access. Depending on the system, there may be requirements or restrictions on passwords, such as having a certain length, or requiring or disallowing certain characters.

**Bitwise XOR $\oplus$:** A bitwise XOR operation takes two bit patterns of the same length and returns 1 when the bits are different and 0 when the bits are the same. In Figure 2.1, we can see that when comparing 1 to 1 or 0 to 0, we get 0 and when we compare 1 to 0, we get 1.

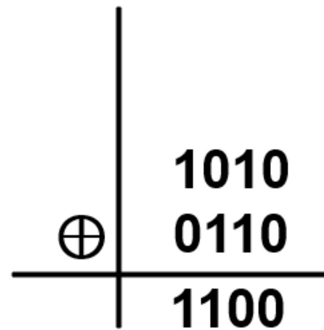



**Figure 2.1:** Bitwise XOR operation on two binary numbers.

**Concatenation ||:** The operation where two groups of characters are glued together. For example, if we have two strings $A =$ "crypto" and $B =$ "graphy" and we concatenate them together we get $A||B =$ "cryptography".

**Hash function:** A hash function $H$ takes as input a bit string $M$, and produces as output a fixed-size hash value $h = H(M)$. These hash values are called *hash values, hash codes* or *digests*, but in this thesis, we will mainly call them *hashes*. The goal with the functions is that given different types and sizes of data as input, they should give outputs that are evenly distributed and seemingly random. For example, if the input was a password, the output would be a random string of characters having no visible connection to the original password. Another desired property is that the slightest change in the input $M$, should change the output $h$ drastically [32].

**Plaintext:** Readable text. The original, unaltered text before it is entered into an algorithm.

**Cipher:** A algorithm that either encrypts or decrypts its inputs. When encrypting, a cipher takes a plaintext as input and encrypts it. The encrypted output is called a **ciphertext**. For decryption, the reverse is done, by entering the ciphertext into the cipher, which then outputs the plaintext.

**Stream cipher:** A stream cipher is a encryption algorithm that reads a stream of input bit-by-bit or byte-by-byte to produce a ciphertext. Let's present a simple example stream cipher: the example stream cipher takes takes as input 4 bits 1001, XORs this with 4 pseudorandom bits, which gives the output 0101, i.e., $1001 \oplus 1100 = 0101$ [32].

**Block cipher:** A block cipher $B$, takes as input a block of $n$ bits and a key of $k$ bits, which the cipher uses to output a output block of $n$ bits. For example, if the algorithm takes a 128-bit block as input, the output will be a ciphertext block of 128 bits [32].

**Salt:** A random value used together with the original input data in hash functions in order to make all hashes unique. One reason for their use is to protect the password hashes in databases if a breach were to occur, since without salts, the attacker would be able to find out whether some users have identical passwords, because identical passwords produce identical hashes. For example, if user1 and user2 both use "abc123" as their password, the hash of their passwords will be the same, but if salts are used, their hashes are different, making it more difficult for an attacker to guess the passwords. The salts themselves should never be reused (this would defeat the purpose of them), but they do not need to be hidden and are usually stored in the same place where the passwords are stored [10].

**MAC:** A message authentication code (MAC) is a function that allows a user to verify the authenticity of a message. When sending a message and using MAC, the receiver is able to verify that the message has not been tampered with and is sent from a trusted party. The use of MAC is as follows: Two users $A$ and $B$ have a shared key $K$. When $A$ wants to send a message $M$ to $B$, $A$ uses a MAC function $C$ to calculate a checksum by using $K$ and $M$ as input, so MAC = $C(K, M)$. $A$ then send the message $M$ and the MAC to $B$, so that $B$ can calculate the same MAC. Since $B$ has the same key as $A$, $B$ can be sure that the message came from $A$ if the two MACs match, otherwise the message (or the MAC) has been tampered with [32].

**Preimage attack:** When talking about hashes, a preimage is the input given to a hash function, i.e., given a hash $h = H(x)$ (were H is a hash function), $x$ would be the preimage of $h$. A preimage attack is an attack where a attacker is able to compute the original preimage of a hash, even though the function is supposed to be one way. The way this is done varies based on the hash function, but essentially this means that an attacker has found a flaw in the hash function, which makes it possible to derive a preimage from a hash [32].

**Second-preimage attack:** This attack raises from the fact that hash values may have multiple preimages. Given the same hash $h = H(x)$, one can find the same hash with a different preimage $h = H(y)$, meaning $H(x) = H(y)$. The second-preimage attack can be performed in various ways, but the main point will be to find a preimage that gives the correct hash, meaning that an attacker could, e.g., access some data with the wrong password because it resulted in the same hash as the correct password would have [32].

**Collision attack:** With a collision attack, the attacker gets to choose inputs to a hash function and tries to get a collision. Getting to choose inputs $x$ and $y$, the attacker tries

to get a result where the hashes will be the same, i.e., $H(x) = H(y)$.

**Cryptographic hash functions:** When talking about hash functions that are used with security applications, they are referred to as cryptographic hash functions. An expected property with these cryptographic hash functions is that they should be able to resist known cryptanalytic attacks [32]. Ways to measure this is to see if a function has the following properties:

- Given a hash $h$, it should be computationally infeasible to find an input $M$ where $h = H(M)$ for a cryptographic hash function $H$. This is commonly called *the one-way property* [32].
- Given two different inputs $M1$ and $M2$, so $M1 \neq M2$, it should be computationally infeasible to find hashes for them where $H(M1) = H(M2)$. This is commonly called *the collision-free property* [32].

**Avalanche effect:** When in the context of hash functions, the avalanche effect means that the smallest change to the input should make the output hash completely different. This means that it should not be possible to determine if the preimages of two hashes are similar, based solely on observing the hashes. Since it will be impossible for an attacker to compare similar preimages to their hashes, it means that our function will be significantly more secure. An example of the effect with the SHA-256 algorithm can be seen below:

```
SHA−256(''The quick brown fox jumps over the lazy dog'')
= d7a8fbb307d7809469ca9abcb0082e4f8d5651e46d3cdb762d02d0bf37c9e592
SHA−256(''The quick brown fox jumps over the lazy dog.'')
= ef537f25c895bfa782526529a9b63d97aa631564d5d789c2b765448c8635fb6c
```

*Even though we only added "." to the end of the second input, one can see that the hashes are completely different, meaning that the avalanche effect has occurred.*

**Secure hash algorithms (SHA):** The secure hash algorithms (SHAs) are cryptographic hash functions published as federal information processing standards (FIPS) by the National Institute of Standards and Technology (NIST). SHA-0, SHA-1 and SHA-2 were created by the United States National Security Agency (NSA), while the newest version SHA-3 was chosen during a public competition held by NIST. The algorithms have had a wide user base throughout the years [32].

**Key derivation function:** A key derivation function (KDF) creates one or more derived keys from an input key and possibly more parameters. In the context of password hashing, the derived key would be the password hash. The original use for the functions were to create keys of specific format, e.g., making them longer or more secure. KDFs are however used frequently for password hashing, in which case they get passwords as input as well as salts [16]. Another use case for KDFs involving passwords, would be generating an encryption key, which in turn is used to encrypt a database of passwords.

**Password cracking:** Password cracking is the process of recovering passwords from data that is usually protected, or encrypted in some way [11]. In this thesis, when using the term cracking, breaking or password recovery, we mean essentially the same thing.

**Hashcat:** Hashcats own website states, that “Hashcat is the world’s fastest and most advanced password recovery utility, supporting five unique modes of attack for over 300 highly-optimized hashing algorithms”. The tool is used to recover/break password hashes in order to get a actual text password as a output [11]. The use of the tool in this thesis will be explained later.

# 3 Common building blocks for password encryption

In this chapter we will present some commonly used building blocks, when encrypting passwords. Brief history and use cases of each building block will be presented together with general explanations of how they work. The algorithms that will be presented are: The Secure Hash Algorithm 2, the message authentication code HMAC, the key derivation function PBKDF2 and the stream cipher Salsa20.

## 3.1 Secure Hash Algorithm 2

Secure Hash Algorithm 2 (SHA-2) is a collection of hash algorithms developed by the NSA and standardized by NIST. A newer version of the standard called SHA-3 is also available, but it has not been widely used yet. NIST has said that if SHA-2 is found not to be secure, they would be ready to update their standards and substitute SHA-2 for SHA-3 [13]. We will focus on the SHA-2 algorithms in this thesis, since they are still considered secure unlike the earlier versions of SHA (SHA-0 and SHA-1), which have been cryptographically broken and are susceptible to, e.g., collision attacks [14]. The SHA-2 algorithms produce hashes with lengths varying from 224 bits to 512 bits. The algorithms were published in 2001 in the draft FIPS PUB 180-2 [19]. The FIPS PUB 180-2 became the secure hash standard in August 2002, meaning that these algorithms were deemed secure by the United States government. NIST also informed federal agencies in 2006 that they should start adopting SHA-2 and concluded that SHA-1 could only be used in special cases after 2010 [20]. The SHA-2 algorithms have seen a wide use throughout the years, e.g., in TLS, SSL, SSH and Bitcoin [32] [23].

We will explain how the SHA-256 algorithm works by going through it step by step. The input message that the algorithm gets will be called $M$. In this explanation, addition (+) is performed modulo $2^{32}$. Additionally, the big-endian convention is used when expressing 32-bit and 64-bit words. A detailed presentation of how the compression function (step 5c) works can be seen in Figure 3.1.

We will first introduce some functions that the algorithm uses:

$SHR^n(x)$ is a right shift binary operator. It shifts the bits of a binary number $x$, $n$ steps to the right. For example, $SHR^2(01101001) = 00011010$.

$SHL^n(x)$ is a left shift binary operator. It shifts the bits of a binary number $x$, $n$ steps to the left. For example, $SHR^2(01101001) = 10100100$.

$ROTR^n(x)$ is a circular right shift operation, where $x$ is a $w$-bit word and $n$ is an integer. The operation works as follows: $ROTR^n(x) = SHR^n(x) \vee SHL^{w-n}(x)$

For the following functions, the variables $X$, $Y$ and $Z$ are bit strings. Each function also outputs a new bit string.

$Ch(X, Y, Z) = (X \wedge Y) \oplus (\neg X \wedge Z)$
$Maj(X, Y, Z) = (X \wedge Y) \oplus (X \wedge Z) \oplus (Y \wedge Z)$

$\sigma_0(X) = ROTR^7(X) \oplus ROTR^{18}(X) \oplus SHR^3(X)$
$\sigma_1(X) = ROTR^{17}(X) \oplus ROTR^{19}(X) \oplus SHR^{10}(X)$
$\sum_0(X) = ROTR^2(X) \oplus ROTR^{13}(X) \oplus ROTR^{22}(X)$
$\sum_1(X) = ROTR^6(X) \oplus ROTR^{11}(X) \oplus ROTR^{25}(X)$

The SHA-256 algorithm works as follows:

1. Initialize the $H_i$ values, which were obtained by taking the first 32 bits of the fractional parts of the square roots of the first eight prime numbers:

$$
\begin{aligned}
H_0 &= 0x6a09e667 \\
H_1 &= 0xbb67ae85 \\
H_2 &= 0x3c6ef372 \\
H_3 &= 0xa54ff53a \\
H_4 &= 0x510e527f \\
H_5 &= 0x9b05688c \\
H_6 &= 0x1f83d9ab \\
H_7 &= 0x5be0cd19
\end{aligned}
$$

2. Initialize the $K$ array with the following constants, which represent the first 32 bits of the fractional parts of the cube roots of the first sixty- four prime numbers:

```
0x428a2f98 , 0x71374491 , 0xb5c0fbcf , 0xe9b5dba5 ,
0x3956c25b , 0x59f111f1 , 0x923f82a4 , 0xab1c5ed5 ,
0xd807aa98 , 0x12835b01 , 0x243185be , 0x550c7dc3 ,
0x72be5d74 , 0x80deb1fe , 0x9bdc06a7 , 0xc19bf174 ,
0xe49b69c1 , 0xefbe4786 , 0x0fc19dc6 , 0x240ca1cc ,
0x2de92c6f , 0x4a7484aa , 0x5cb0a9dc , 0x76f988da ,
0x983e5152 , 0xa831c66d , 0xb00327c8 , 0xbf597fc7 ,
0xc6e00bf3 , 0xd5a79147 , 0x06ca6351 , 0x14292967 ,
0x27b70a85 , 0x2e1b2138 , 0x4d2c6dfc , 0x53380d13 ,
0x650a7354 , 0x766a0abb , 0x81c2c92e , 0x92722c85 ,
0xa2bfe8a1 , 0xa81a664b , 0xc24b8b70 , 0xc76c51a3 ,
0xd192e819 , 0xd6990624 , 0xf40e3585 , 0x106aa070 ,
0x19a4c116 , 0x1e376c08 , 0x2748774c , 0x34b0bcb5 ,
0x391c0cb3 , 0x4ed8aa4a , 0x5b9cca4f , 0x682e6ff3 ,
0x748f82ee , 0x78a5636f , 0x84c87814 , 0x8cc70208 ,
0x90befffa , 0xa4506ceb , 0xbef9a3f7 , 0xc67178f2
```

3. Pre-process and pad the message $M$ so that it is in the right format:
   (a) We note that the length of $M$ is $l$ bits.
   (b) Append at the end of $M$ the bit "1".
   (c) Append $k$ "0" bits at the end of $M$, where $k$ is the smallest, non-negative solution to the equation $l + 1 + k \equiv 448 \pmod{512}$.
   (d) Append at the end of $M$ the binary representation of $l$ in a a 64-bit block.
4. Break $M$ into 512-bit blocks. The number of blocks that $M$ is broken into is noted by $N$. The different blocks are labeled with a subscript notation, in the following way: $M_1$, $M_2$, ... , $M_N$.
5. For $i = 1$ to $N$:
   (a) Prepare a message schedule array $W_t$ of 64 32-bit words, where $t$ indicates the index of the array:
      i. Divide $M_i$ into 16 32-bit words and copy these words to $W_0 - W_{15}$.
      ii. For $t$ from 16 to 63:
      $W_t = \sigma_1(W_{t-2}) + W_{t-7} + \sigma_0(W_{t-15}) + W_{t-16}$

(b) Initialize eight working variables with the current hash values:

$$\begin{aligned} \mathrm{a} &= H_0 \\ \mathrm{b} &= H_1 \\ \mathrm{c} &= H_2 \\ \mathrm{d} &= H_3 \\ \mathrm{e} &= H_4 \\ \mathrm{f} &= H_5 \\ \mathrm{g} &= H_6 \\ \mathrm{h} &= H_7 \end{aligned}$$

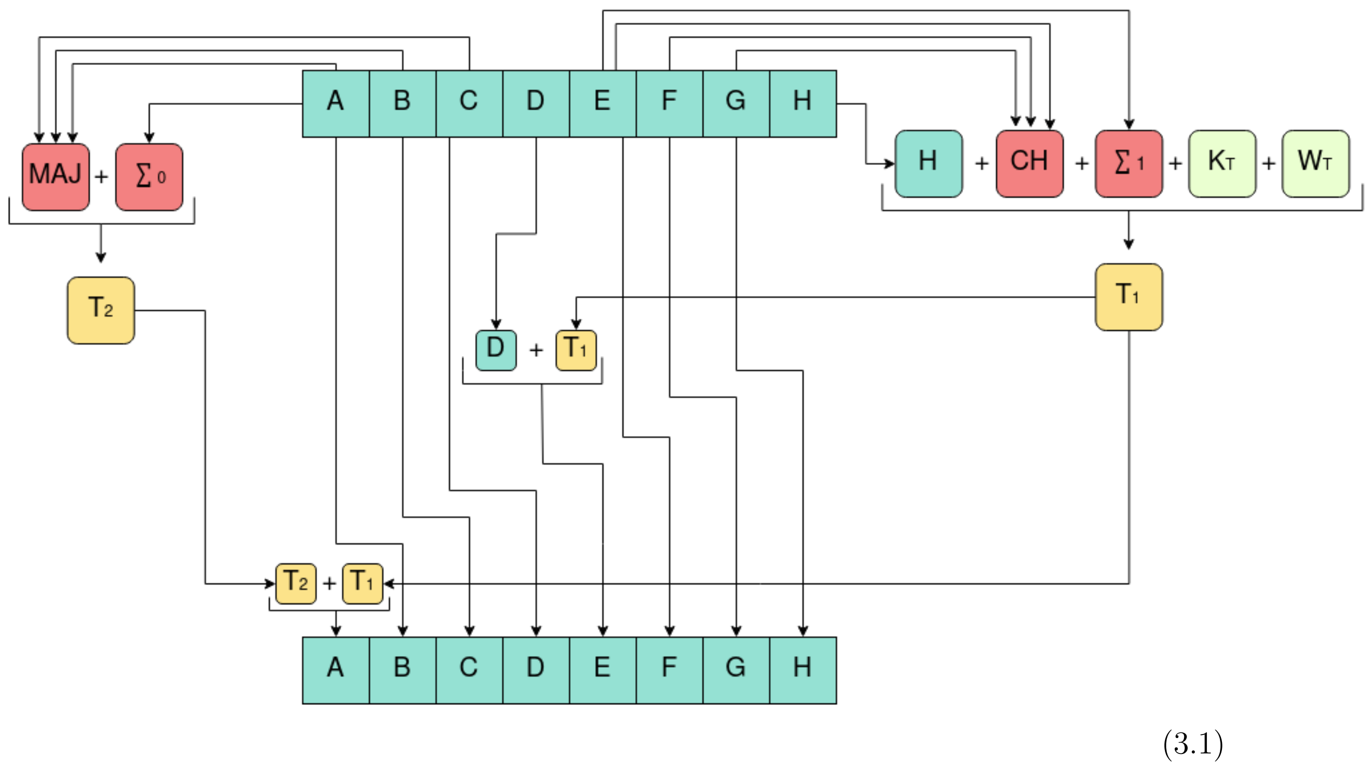
(c) For $t = 0$ to 63:

$$\begin{aligned} T_1 &= \mathrm{h} + \textstyle\sum_1(e) + \mathrm{Ch}(\mathrm{e}, \mathrm{f}, \mathrm{g}) + K_t + W_t \\ T_2 &= \textstyle\sum_0(a) + \mathrm{Maj}(\mathrm{a}, \mathrm{b}, \mathrm{c}) \\ \mathrm{h} &= \mathrm{g} \\ \mathrm{g} &= \mathrm{f} \\ \mathrm{f} &= \mathrm{e} \\ \mathrm{e} &= \mathrm{d} + T_1 \\ \mathrm{d} &= \mathrm{c} \\ \mathrm{c} &= \mathrm{b} \\ \mathrm{b} &= \mathrm{a} \\ \mathrm{a} &= T_1 + T_2 \end{aligned}$$

(d) Add the working variables to the current hash values:

$$\begin{aligned} H_0 &= \mathrm{a} + H_0 \\ H_1 &= \mathrm{b} + H_1 \\ H_2 &= \mathrm{c} + H_2 \\ H_3 &= \mathrm{d} + H_3 \\ H_4 &= \mathrm{e} + H_4 \\ H_5 &= \mathrm{f} + H_5 \\ H_6 &= \mathrm{g} + H_6 \\ H_7 &= \mathrm{h} + H_7 \end{aligned}$$

6. We get the final hash by appending all our hash values together:
   $Hash = H_0||H_1||H_2||H_3||H_4||H_5||H_6||H_7$




(3.1)

**Figure 3.1:** Figure demonstrating one iteration of SHA-256 compression function. The $T$ stands for the iteration count.

As the name SHA-2 suggests, it is not the first version of the SHA algorithms. Even though SHA-2 was a new improved version for the Secure Hash Algorithm family, it still shares similarities with the previous versions, SHA-0 and SHA-1. More specifically, SHA-2 shares the same structure and uses the same mathematical operations as the previous versions. This connection has raised some legitimate concerns about SHA-2 security. This is because multiple vulnerabilities and exploits have been found for SHA-0 and SHA-1 and they have been deemed cryptographically broken [32]. SHA-2 itself has not been broken, but experts do not recommend using it for securing passwords. Some exploits and vulnerabilities have though been found for SHA-2, some of which will be explained later in the thesis. These are however not too critical, which is why the SHA-2 algorithms are still in active use today.

## 3.2 Hash-based Message Authentication Code

Hash-based Message Authentication Code (HMAC) is a type of MAC, which employ the strengths of hash algorithms. HMAC was released in 1997 and was created by IBM and the University of California San Diego. As with MAC, its main purpose is to verify the authenticity of some message. One benefit with HMAC, is that it can use any hash function and can swap out the hash function as needed, e.g., if the one currently used is deemed insecure, or if a new better one is developed. This is possible, since HMAC handles the hash function as a black box, i.e., it only cares about what output the function gives, when given an input [32].

In addition to the upcoming explanation, we can also see how HMAC works in Figure 3.2.

HMAC will get as input a bit string $K$, a message $M$ and a hash function $H$. The block size of $H$ is noted with $S$. HMAC has the constants $ipad$ = 0x36 and $opad$ = 0x5c which are both repeated until they are the length of $S$. The function is called in the following way: $HMAC(K, M)$. HMAC works as follows:

1. If $K$ is not the size of $S$:
   (a) If the size of $K > S$, create a string $L$ by hashing $K$ with $H$, $L = H(K)$ and set $K = L$.
   (b) If the size of $K < S$, append zeros to the end of $K$ until it is of the desired block size $S$.
2. $iKeypad = ipad \oplus K$.
3. $oKeypad = opad \oplus K$.
4. HMAC = $H(\ oKeypad \ || \ H(\ iKeypad \ || \ M\ )\ )$

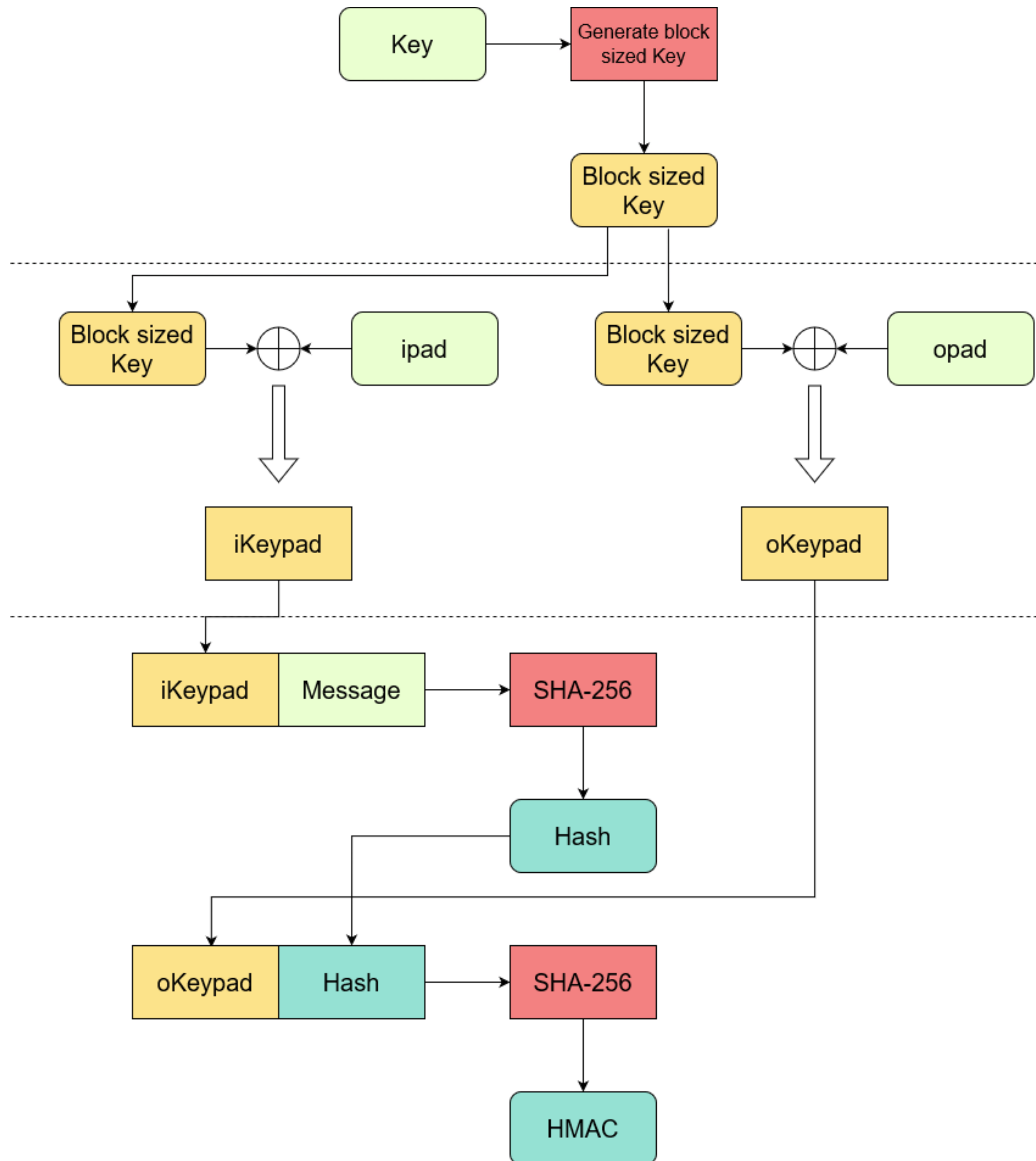



**Figure 3.2:** Figure demonstrating how a HMAC is generated using SHA-256 as the hash function.

As one can see from the explanations, HMAC itself is not a complicated function. During a typical execution of HMAC, three runs of the embedded hash function is performed. This means that HMAC's run time will be close to the hash function's for long messages. This simplicity and dependency on the embedded hash function has also worked in HMAC's favor when thinking about its security. For any hash function, the designers of HMAC were able to prove an exact relationship between the strength of that hash function and

the strength of HMAC [32].

## 3.3 PBKDF2

The password based key derivation function 2 (PBKDF2), is a key derivation function which has the desirable property of resisting brute-force attacks. This resistance is mainly accomplished by allowing the user to configure the function's iteration counts, which in turn can make the function relatively expensive to run [28]. The function was created by RSA laboratories and was part of their Public-Key Cryptography Standards (PKCS) series. It was released 2000, in both PKCS #5 and RFC 2898 [16].

PBKDF2 is of interest to us in this thesis, since it is able to resists brute-force attacks quite well compared to other more traditional hash functions (like SHA-256). The way this resistance is accomplished can be divided into multiple steps, but one of the main factors would be that the function goes through a lot of iterations before outputting the final derived key. This means that when the amount of time it takes to calculate one password hash is long enough, it becomes impractical to calculate multiple keys in quick succession, as one does with a brute-force attack. A potential drawback of configuring too many iteration counts would be that multiple sign in attempts could put unnecessary strain on a server. Another result of this is that authentication could become impractical for a user, i.e., a user would have to potentially wait a frustratingly long time to log in to a system.

We will now go through the steps of PBKDF2, where we use HMAC and SHA-256 for our pseudorandom function (PRF). PBKDF2 takes as input the following parameters:

- **PRF:** The pseudorandom function that will be used, which is in this case HMAC-SHA-256 (which is HMAC that uses SHA-256 as its hash function).
- **Password:** The password that we want to secure.
- **Salt:** The salt that will be used.
- **c:** The number of the desired iterations.
- **dkLen:** The desired length wanted for the derived key, which fulfills:
  $dkLen < (2^{32} - 1) \cdot hLen$, where $hLen$ is the length of our PRF output in octets.

PBKDF2 will use the following functions:

$CEIL(X)$ returns the smallest integer greater than, or equal to x.

$INT(X)$ returns a four-octet and big-endian encoding of the integer *X*.

$F(Password, salt, c, X) = U_1 \oplus U_2 \oplus ... \oplus U_c$, where the $U_i$ are generated in the following way:

$U_1 = PRF(Password, salt||INT(X))$,
$U_2 = PRF(Password, U_1)$,
...
$U_c = PRF(Password, U_{c-1})$

When the function is applied, it will output the derived key *DK*. In other words, the function is called in the following way: PBKDF2(PRF, Password, Salt, c, dkLen).

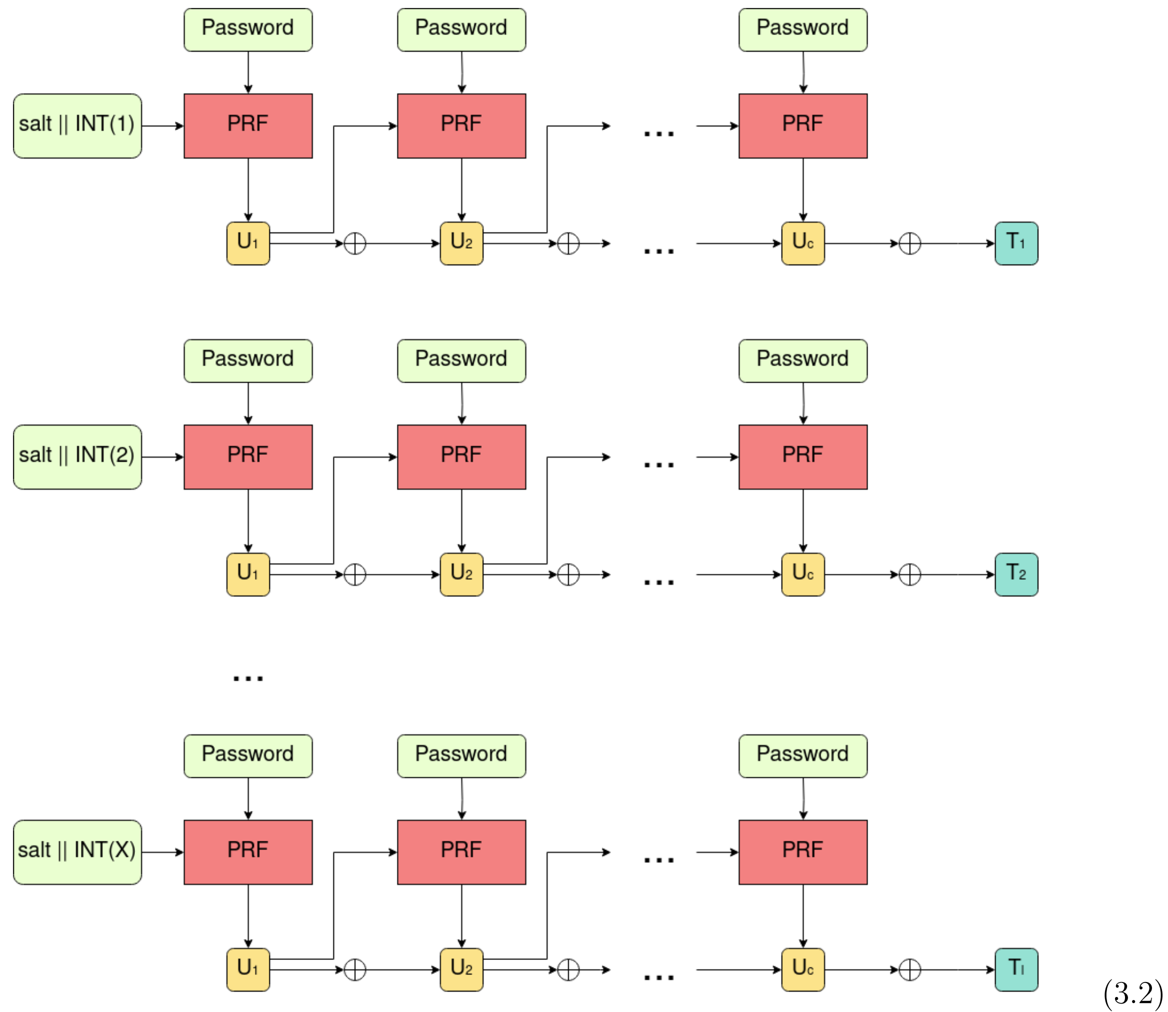
PBKDF2 works as follows:

1. Let $l = CEIL(dkLen/hLen)$.

2. Initialize $l$ amount of *hLen*-octet blocks to make up the derived key. The blocks will be defined as $T_1$, $T_2$, ..., $T_l$. A more detailed look on how the blocks $T$ are generated can be seen in Figure 3.3.

3. For $i = 1$ to $l$:
   $T_i = F(Password, salt, c, i)$.

4. $DK = T_1||T_2||...||T_l$




(3.2)

**Figure 3.3:** Figure demonstrating how PBKDF2 generates its blocks $T$.

As explained above, PBKDF2 utilizes our previous explained algorithms SHA-256 and HMAC. It is a good design choice to build on older algorithms, since one can utilize their strengths to make the new algorithm even more secure. SHA-256 is a desirable choice here as a hash function, since it has been used for a long time without being broken. Combining SHA-256 with a salt using HMAC makes the PRF in this case more secure compared to normal SHA-256. While also taking into account the iteration count modifier, PBKDF2 has been able to establish itself as a popular choice to secure passwords.

## 3.4 Salsa20

Salsa20 is a stream cipher that was created by Daniel J. Bernstein in 2005. It was submitted to eSTREAM, which was a project that wanted to "identify new stream ciphers suitable for widespread adoption" [3]. Salsa20 was later added to eSTREAMs portfolio of ciphers. The default version of the cipher performs 20 rounds of mixing the input, but there exist other versions, such as Salsa20/8 and Salsa20/12 which perform fewer rounds, which claim to be "among the fastest 256-bit stream ciphers available" [3].

Only the core function of the algorithm interest us in this thesis, since it is the part that is used in scrypt. We will now go through how an implementation of Salsa20s core function would work:

1. Initialize the function *ROTL(a, b)*, which takes a 32-bit word $a$ and rotates it by $b$ bits to the left. The bits that are shifted out from the left side are wrapped around to the right side. For example, $ROTL(11010101, 2) = 01010111$.

2. Initialize the function *QR(a, b, c, d)*:
   $b = b \oplus ROTL(a + d, 7)$
   $c = c \oplus ROTL(b + a, 9)$
   $d = d \oplus ROTL(c + b, 13)$
   $a = a \oplus ROTL(d + c, 18)$

3. Take as input an array *IN* containing 16 32-bit integers.

4. Copy *IN* to *X*.

5. For $i = 0$ to 20, do:

   (a) $QR(X_0, X_4, X_8, X_{12})$

   (b) $QR(X_5, X_9, X_{13}, X_1)$

   (c) $QR(X_{10}, X_{14}, X_2, X_6)$

   (d) $QR(X_{15}, X_3, X_7, X_{11})$

   (e) $QR(X_0, X_1, X_2, X_3)$

   (f) $QR(X_5, X_6, X_7, X_4)$

   (g) $QR(X_{10}, X_{11}, X_8, X_9)$

   (h) $QR(X_{15}, X_{12}, X_{13}, X_{14})$

6. Initialize array $OUT$.

7. For $i = 0$ to 16, do:
   $OUT_i = X_i + IN_i$

8. return $OUT$.

When studying the security of Salsa20, some concerns may come up. Cryptanalysis has been done on the stream cipher by multiple researchers, which has led to the development of many successful attacks against it. An attack by Aumasson et al., which improves on an previous attack (by Shi et al.) had a time complexity of $2^{109}$ against the 20/7 version and a time complexity of $2^{250}$ against the 20/8 version [31]. Even though the cipher has these security flaws, it still has uses in other areas. In scrypt, the 20/8 version of the cipher is used in order to make randomly ordered requests to the RAM.

# 4 Scrypt

In this chapter we will discuss the key derivation function scrypt: a bit of its history, the motivations behind it, how it works and how it is used in cryptocurrencies.

## 4.1 Background

Scrypt is a key derivation function created by Colin Percival in 2009 and it was originally developed for Tarsnap's online backup system [34]. It was designed to be more secure and to resist significantly better brute-force attacks than other (at the time) popular key derivation functions such as PBKDF2 and bcrypt [34].

As it was stated in chapter 3, the more time it takes to compute a function, the more resistant it is against brute-force attacks. If one run of a function takes a second or two, this time is negligible to a user who tries to, e.g., log in to a system, but it is much more difficult for a malicious user to use a brute force attack against the function. Colin Percival points out how previous key derivation functions have been designed to use multiple iterations of the function to improve the security of them [24]. Some functions like PBKDF2 allow the user to specifically define how many iterations the function should perform [16]. The idea of this strategy is that as computer hardware improves and gets faster, one can simply increase the iteration counts of a function (like PBKDF2) to uphold its security. This is because increasing the iteration counts will make brute-force attacks more expensive. However, Percival points out in his paper that widely used key derivation functions that use the strategy of iteration counts have used "constant amounts of logic and memory". With scrypt he intends to also increase the memory usage, thus limiting the number of calculations that can be done in parallel[24].

The concept of memory-hard algorithms[24] is something that scrypt adopts in its design. The aim of memory-hard algorithms is to require a significant amount of memory to compute their operations, making it challenging for attackers to parallelize the computations efficiently. The memory requirements are typically proportional to the algorithm's time complexity, making it difficult for attackers to speed up computations by using specialized hardware.

## 4.2 Description of the scrypt function

We will now explain how an example implementation of scrypt works. This particular implementation uses PBKDF2 with HMAC and SHA-256 as its pseudo random function. When finished, scrypt outputs a derived key *DK* [25]. Recall that PBKDF2, HMAC, Salsa20 and SHA-256 were explained in chapter 3. We will first introduce *BlockMix* and *ROMix*, since they are functions that scrypt uses.

**BlockMix** takes the following parameters:

- ***B:*** The input array of size $128 \cdot r$ octets. Each index contains a $2r$ 64-byte block.
- ***r:*** Block size.

BlockMix uses Salsa20/8, which is a 8-round version of Salsa20.

The **BlockMix** function works as follows:

1. Initialize an array $Y$ of size $2r$.
2. $X = B_{2r-1}$
3. For $i = 0$ to $2r - 1$ do:
   (a) $X = Salsa20/8(X \oplus B_i)$
   (b) $Y_i = X$
4. return $Y_0||Y_2||...||Y_{2r-2}||Y_1||Y_3||...||Y_{2r-1}$

**ROMix** takes the following parameters:

- **r:** Block size.
- **N:** CPU/memory cost expressed as an integer.
- **B:** The input array.

The **ROMix** function works as follows:

1. $X = B$
2. Initialize an array $V$ of size $N$.
3. For $i = 0$ to $N - 1$ do:
   (a) $V_i = X$
   (b) $X = BlockMix(X)$
4. For $i = 0$ to $N - 1$ do:
   (a) $j = Integrify(X) \, mod \, N$, where *Integrify* returns the last 64 bytes of X interpreted as a little-endian integer.
   (b) $T = X \oplus V_j$
   (c) $X = BlockMix(T)$
5. Return $X$.

**Scrypt** takes the following parameters:

- **P:** The password that we want to protect.
- **S:** The salt that will be used.
- **N:** CPU/memory cost.
- **r:** Block size.
- **p:** Parallelization parameter, which is a positive integer that satisfies the inequality $p \leq (2^{32}-1) \cdot hLen/128r$, where:
  - **hlen:** The output length in octets of the hash function, which is 32 for SHA-256.
- **dkLen:** The desired length we want for our derived key.

**Scrypt** itself works as follows:

1. Initialize an array $B$ with $p$ blocks with size of $128r$ octets.
2. $B_0||B_1||...||B_{p-1} = PBKDF2_{HMAC-SHA256}(P, S, 1, p \cdot 128 \cdot r)$.

3. For $i = 0$ to $p - 1$ do:
   $B_i = ROMix(r, B_i, N)$

4. $expensiveSalt = B_0||B_1||...||B_{p-1}$.

5. Our final derived key $DK$ is then generated with $PBKDF2_{HMAC-SHA256}$, so $DK = PBKDF2_{HMAC-SHA256}(P, expensiveSalt, 1, dkLen)$.

Note that, scrypt uses multiple cryptographic building blocks to ensure that the derived key is securely generated. As a sidenote, in order to have a good grasp of how scrypt works, one need to understand also how PBKDF2, HMAC, SHA-256 and Salsa20 works.

## 4.3 Usage in cryptocurrencies

Scrypt and variations of scrypt can be found in various cryptocurrencies. They are used as proof-of-work algorithms. Proof-of-work algorithms are generally difficult computational tasks. Their purpose is to "reach consensus regarding the ledger history, thereby synchronizing the transactions and making the users secure against double-spending attacks" [2].

One of the main reasons why cryptocurrencies prefer scrypt over other hash functions is the time-memory trade off property of scrypt. The property would allow a user to configure scrypt to require demanding hardware to perform a calculation. This means that it would be quite expensive to perform multiple calculations of proof-of-work algorithms consecutively. With some implementations of scrypt, users have been able to make the computations themselves fast, but still requiring lots of memory. This is a desirable feature for several cryptocurrencies, since there still is a hardware requirement for proof-of-work calculations, but they can be done in reasonable time. The better security and adjustment options of scrypt has meant that scrypt has been a desirable option as a hash function when developing cryptocurrencies [18]. Examples of cryptocurrencies that use scrypt are Dogecoin, Litecoin, Verge and Einsteinium [9].

# 5 Exploits and potential attacks

In this section, we present known exploits and attacks on hash functions and cryptographic functions. In addition to this, we will also introduce the password recovery tool hashcat, which employs many different techniques and optimizations to break password hashes. We will focus on exploits that have been used on the algorithms and functions that were explained in the previous chapters. In addition to this, examples, tests and benchmarks will be shown.

## 5.1 Hashcat

Hashcat describes itself to be “the world’s fastest and most advanced password recovery tool” [11]. The tool was originally created by Jens Steube in 2009. Motivations behind creating the tool was that at the time, popular password recovery tools (like “PasswordsPro” or “John The Ripper”) did not support multi-threading. Different versions of hashcat were developed in the beginning with different features, e.g., hashcat (now called hashcat-legacy) originally only supported CPU-based cracking, while the oclhashcat versions supported GPU-based cracking. But with the release of version 3.00 in 2016, the versions were fused into one program called hashcat, which supports both CPU- and GPU-based cracking. Hashcat was originally proprietary software, but was later in 2015 released as open source software. Hashcat is published under the MIT License [11].

Hashcat allows the user to try to crack a password in many different ways, by the use of different attack modes. The core attack modes that hashcat offers are:

**Dictionary Attack:** Also known as “straight mode” or a “wordlist attack”, is one of the most simple attack modes. This mode is activated with the “-a 0” flag. The mode takes as input a textfile containing plaintext password candidates. Hashcat tries these candidates sequentially by hashing them with the chosen hash function and then comparing it to the password hash that the user wants to break [11]. An example dictionary attack would be initiated with the following command:

```
hashcat -m 0 -a 0 md5Example.hash testDict.dict
```

**Combinator Attack:** A combinator attacks gets as input two dictionaries containing

plaintext password candidates. This attack mode is activated with the “-a 1” flag. Each candidate gets appended to another candidate from the other dictionary to generate new password candidates [11]. For example, we have the two following dictionaries:

```
dict1.txt:   |    dict2.txt:
abc          |    123
password     |    admin
secret       |    0000
```

An example combinator attack with these dictionaries would be initiated with the following command:

```
hashcat -m 0 -a 1 combTestMd5.hash dict1.txt dict2.txt
```

With these dictionaries, hashcat would create and try the following password candidates:

```
abc123
abcadmin
abc0000
password123
passwordadmin
password0000
secret123
secretadmin
secret0000
```

**Brute-Force Attack:** Also called the mask attack, is a classic attack trying all possible password combinations, in accordance to user specifications. Hashcat’s original version of the Brute-force attack simply tried all possible combinations and did not allow much user specifications for the attack. The new brute-force attack, commonly called as the mask attack, replaced the old brute-force attack and allows the user the make the attack more specific, thus saving a lot of time. Since humans usually follow some patterns when creating passwords, we can make educated guesses, instead of wasting time on unreasonable guesses. For example, if a password requires a capital letter, it is most often found at the start, rarely in the middle of the password. To start a mask attack, the user will enter a series of “masks”, or charsets in the form of a “?” followed by a letter, in order to specify a attack. The attempts will be of the length that the user inputted, or will gradually increment the

size if the "-i" flag is given. The possible masks that a user can input and what charset they contain are:

- ?l = abcdefghijklmnopqrstuvwxyz
- ?u = ABCDEFGHIJKLMNOPQRSTUVWXYZ
- ?d = 0123456789
- ?h = 0123456789abcdef
- ?H = 0123456789ABCDEF
- ?s = «space»!"#$%&'()*+,-./:;<=>?@[\]^_‘|
- ?a = ?l?u?d?s
- ?b = 0x00 - 0xff

So, if we, e.g., wanted to crack the password "Robert1990", we would use the following attack:

```
hashcat -m 0 -a 3 bruteTestMd5.hash ?u?l?l?l?l?l?d?d?d?d
```

So, one uppercase letter, five lowercase letters and four digits, which equals to a 10 character long password. The actual order that hashcat goes though its password candidates in a brute-force attack is not a simple ordering scheme, such as alphabetical order. The candidate order is chosen by utilizing Markov chains [11].

**Hybrid Attack:** The hybrid attack, as the name suggests is a hybrid of previous attacks. It works like a combinator attack, but taking as input a dictionary and masks (like in the mask attack). It then either prepends or appends (based on user choice) the masks to the words in the provided dictionary. The attack that appends the masks is started with the "-a 6" flag and the prepend version starts with "-a 7". For example, if we have the following dictionary:

```
dict1.txt:
abc
password
secret
```

We can make the following attack:

```
hashcat -m 0 -a 6 hybridTestMd5.hash dict1.txt ?d?d?d?d
```

Which would generate the following candidates:

```
abc0000
abc0001
abc0002
.
.
.
password9999
secret0000
secret0001
.
.
.
secret9999
```

As one can see, hashcat allows the user to do various attacks and specify them as needed. This means that an attacker can adjust their password cracking approach, depending on the situation. In other words, instead of making broad brute attacks, one can employ smart strategies in order to make specific and far more effective attacks, based on preexisting knowledge.

## 5.2 SHA-256 weaknesses and exploits

SHA-256 in itself is not a bad hash function, it is still leagues better compared to other commonly used hash functions (like MD5) when it comes to security. The algorithm is over 20 years old and has stood the test of time. One might wonder why is it then that security researchers do not deem it good for password encryption and recommend other hash functions.

The answer is not simple, but a central thing to keep in mind is that SHA-256 was not designed to be used for password encryption. When reading the FIPS PUB 180-4 Secure

Hash Standard (SHS) document, one can note the following parts: “This Standard specifies secure hash algorithms - SHA-1, SHA-224, SHA-256, SHA-384, SHA-512, SHA-512/224 and SHA-512/256 - for computing a condensed representation of electronic data (message)” and “Secure hash algorithms are typically used with other cryptographic algorithms, such as digital signature algorithms and keyed-hash message authentication codes, or in the generation of random numbers (bits)”. In other words, SHA-256 is designed to be able to create a hash from large amounts of data and be able to do so in reasonable time. This means its designed to be fast, which is practical for its intended use, but when considering password encryption, this characteristic becomes less desirable [5].

As mentioned before, cryptographic hash functions are designed to have the one-way property, which implies that reversing hashes is not an option. Since reversing is not an option, one of the most popular ways to break hashes is to simply run the hash function with varying inputs in hope of having a match. One could either run a naïve brute-force attack trying every possible combination as input or use more informed attacks based on background information of the password. Either way, these attacks will most likely require the user to perform enormous amounts of runs of hash functions, in order to have a possibility of breaking a password hash. Since SHA-256 is a fast algorithm, it is quite susceptible to brute-force attacks.

To demonstrate the speed of SHA-256 runs, we will show some benchmarks made with hashcat. Hashcats built-in benchmark mode is described on their website as follows: “This mode is simply a brute force attack with a big-enough mask to create enough workload for your GPUs against a single hash of a single hash-type. It just generates a random, uncrackable hash for you on-the-fly of a specific hash-type.” To put SHA-256 speed into perspective, we have done benchmarks on SHA-256, PBKDF2-HMAC-SHA256 and scrypt. The built-in benchmark mode sets in this case the iteration count for PBKDF2-HMAC-SHA256 as 999 and for scrypt 16384. The benchmarks were generated with the following input:

```
hashcat -m x -b
```

The *x* here determines the hash type we want to test (e.g., 1400 would be for SHA-256) and the “-b” flag enables the benchmark mode. The benchmarks can be seen in Figure 5.1, Figure 5.2 and Figure 5.3. The benchmarks were run on a NVIDIA T550 Laptop graphics processing unit (GPU), using nvidia-510 proprietary Linux drivers.

SHA-256

```
---------------------------
* Hash-Mode 1400 (SHA2-256)
---------------------------


Speed.#1.........:  1503.8 MH/s (87.61ms) @ Accel:32 Loops:1024 Thr:256
Vec:1
```

**Figure 5.1:** SHA-256 hashcat benchmark test.

PBKDF2-HMAC-SHA256

```
--------------------------------------------------------
* Hash-Mode 10900 (PBKDF2-HMAC-SHA256) [Iterations: 999]
--------------------------------------------------------


Speed.#1.........:   510.5 kH/s (51.43ms) @ Accel:128 Loops:62 Thr:256
Vec:1
```

**Figure 5.2:** PBKDF2 hashcat benchmark test.

scrypt

```
---------------------------------------------
* Hash-Mode 8900 (scrypt) [Iterations: 16384]
---------------------------------------------


Speed.#1.........:      314 H/s (88.03ms) @ Accel:16 Loops:1024 Thr:32
Vec:1
```

**Figure 5.3:** Scrypt hashcat benchmark test.

The speed here is measured in hashes calculated per second (H/s). When observing SHA-256, we can see that we are able to calculate 1503.8 MH/s, so 1 503 800 000 hashes per second, meaning that brute-forcing passwords does not seem too infeasible. When comparing this to PBKDF2-HMAC-SHA, we can see that it fares far better by only having

the speed 510.5 kH/s, meaning that we have a speed decrease of around 99.97% compared to SHA-256. An even starker contrast can be seen when observing that scrypt's speed was only 314 H/s.

Even though we could see from the previous examples that it is very fast to run SHA-256, we can employ different tricks and exploits, to make it run even faster. When trying to break a hash, usual strategies end up trying a lot of similar passwords, e.g., "password1", "password2", "password3" and so on. This is something one can exploit when doing password attempts on SHA-256. If we recall how SHA-256 works, we note that the same initial variables are always used in the beginning. This means that depending on the length of our message we can precompute the initial runs of the inner for loop and practically skip these steps and go straight to the steps where new values comes into play. By doing this over a lot of runs we can save quite a lot of time over time. By using the example input "password1" we get the hash:

`0b14d501a594442a01c6859541bcb3e8164d183d32937b851835442f69d5c94e`

and with the almost similar input "password2" we get the hash:

`6cf615d5bcaac778352a8f1f3360d23f02f34ec182e259897fd6ce485d7870d4`

The hashes themselves are completely different, but if we debug SHA-256 and output what our a, b,... h values are during a run, we can see that a remarkable portion are identical in the beginning. We did this by using RFC 6234s reference implementation of SHA-256 [7]. The code snippet 5.2 corresponds to step 5c from our SHA-256 explanation in chapter 3. A slight modification has been made to the code, so that it prints out the a, b, ... h values at the end of each iteration. We did two runs with the same inputs as earlier ("password1" and "password2") and the resulting values during iteration 1, 2, 3, and 4 can be seen in 5.4 and 5.5.

```
for (t = 0; t < 64; t++) {
    temp1 = H + SHA256_SIGMA1(E) + SHA_Ch(E,F,G) + K[t] +W[t];
    temp2 = SHA256_SIGMA0(A) + SHA_Maj(A,B,C);
    H = G;
    G = F;
    F = E;
    E = D + temp1;
    D = C;
    C = B;
    B = A;
    A = temp1 + temp2;
    printf("%d %x%x%x%x%x%x%x%x\n", t, A, B, C, D, E, F, G, H);
}
```

1. 6c69fbc06a09e667bb67ae853c6ef37209295615510e527f9b05688c1f83d9ab
2. 6ac5a1066c69fbc06a09e667bb67ae85d9d489a509295615510e527f9b05688c
3. c3397c226ac5a1066c69fbc06a09e66790511f9ad9d489a509295615510e527f
4. a46bc18fc3397c226ac5a1066c69fbc0efe2e12590511f9ad9d489a509295615

...

**Figure 5.4:** Each line is the a, b,...h variables concatenated of a iteration.

1. 6c69fbc06a09e667bb67ae853c6ef37209295615510e527f9b05688c1f83d9ab
2. 6ac5a1066c69fbc06a09e667bb67ae85d9d489a509295615510e527f9b05688c
3. c4397c226ac5a1066c69fbc06a09e66791511f9ad9d489a509295615510e527f
4. 28280993c4397c226ac5a1066c69fbc06fdf012591511f9ad9d489a509295615

...

**Figure 5.5:** Each line is the a, b,...h variables concatenated of a iteration. Values that are different compared to the earlier output is marked with red and underlined.

Since the initial values are identical, we can make adjustments in our code to keep these values in memory and skip these steps, in future runs. Even though this is a minuscule improvement in speed, its help will only increment for each run we make with the algorithm. Over time, small improvements like this can make a noticeable difference in cracking speeds.

## 5.3 Optimizations for password cracking

When trying to crack a password hash, an attacker will have to repeatedly run a hash function, while hoping to get a match. To speed up this process, attackers have been able to come up with different optimizations that will allow them to exit a run of a hash function sooner than expected. These optimizations usually only work on a specific hash function, meaning that new optimizations must be found for each unique hash function. We will now present some of the more common optimizations:

**Zero-based optimizations:** Passwords are usually relatively short, not taking a lot of space. Since some hash functions use data blocks to perform the hashing and these blocks are quite large in size, it means that the password has to be padded, in order to take up the space that is expected from the block. This leads to the common situation where a large chunk of the computation that is done from the hash function is done on padding bits. Since padding bits are usually zero, operations such as addition essentially do nothing. So instead of performing these unnecessary computations, one could analyze the length of the password and based on that skip a lot of steps and still be able to get to the same end result. For example, MD4 has a block size of 512-bits, but the string "password" is only 64-bits long, which means that the block needs 448-bits of padding bits to pad the block [33]. To test this optimization out, we used RFC1320s reference implementation of MD4 [29]. We slightly modified MD4s $FF$, $GG$ and $HH$ functions, adding print statements, to show that when we are dealing with padding bits, i.e., $x = 0$, addition operations with $x$ makes no difference to the calculations. The modified part of the code can be seen in 5.3. Some example results using "password" as input can be seen in figure 5.6.

```
#define FF(a, b, c, d, x, s) { \
    printf("%x\n", (F ((b), (c), (d)) + (x))); \
    printf("%x\n", (F ((b), (c), (d)))); \
    (a) += F ((b), (c), (d)) + (x); \
    (a) = ROTATE_LEFT ((a), (s)); \
  }
#define GG(a, b, c, d, x, s) { \
    printf("%x\n", (G ((b), (c), (d)) + (x) + (UINT4)0x5a827999)); \
    printf("%x\n", (G ((b), (c), (d)) + (UINT4)0x5a827999)); \
    (a) += G ((b), (c), (d)) + (x) + (UINT4)0x5a827999; \
    (a) = ROTATE_LEFT ((a), (s)); \
  }
#define HH(a, b, c, d, x, s) { \
    printf("%x\n", (H ((b), (c), (d)) + (x) + (UINT4)0x6ed9eba1)); \
    printf("%x\n", (H ((b), (c), (d)) + (UINT4)0x6ed9eba1)); \
    (a) += H ((b), (c), (d)) + (x) + (UINT4)0x6ed9eba1; \
    (a) = ROTATE_LEFT ((a), (s)); \
  }
```

...

Function call 3.

x = 80

F ((b), (c), (d)) + (x) = cb9d0c09

F ((b), (c), (d)) = cb9d0b89

Function call 4.

**x = 0**

F ((b), (c), (d)) + (x) = **27d33959**

F ((b), (c), (d)) = **27d33959**

...

Function call 17.

x = 73736170

G ((b), (c), (d)) + (x) + (UINT4)0x5a827999 = f5b5e655

G ((b), (c), (d)) + (UINT4)0x5a827999 = 824284e5

Function call 18.

**x = 0**

G ((b), (c), (d)) + (x) + (UINT4)0x5a827999 = **8f6544e2**

G ((b), (c), (d)) + (UINT4)0x5a827999 = **8f6544e2**

...

Function call 41.

x = 64726f77

H ((b), (c), (d)) + (x) + (UINT4)0x6ed9eba1 = 6ce80505

H ((b), (c), (d)) + (UINT4)0x6ed9eba1 = 875958e

Function call 42.

**x = 0**

H ((b), (c), (d)) + (x) + (UINT4)0x6ed9eba1 = **92b65c3d**

H ((b), (c), (d)) + (UINT4)0x6ed9eba1 = **92b65c3d**

**Figure 5.6:** Results showcasing why operations where $x = 0$ are inconsequential. Results that unaffected due to $x = 0$ are bolded.

When putting this optimization in practice, one could skip 36 / 128 ADD instructions with MD4 on passwords which length is less than 12 characters [33].

**Early-exit optimizations:** With the early-exit optimization, we can exploit certain hash functions to make a comparison earlier than intended to reduce the time needed to crack a password. Let us illustrate this with a example using MD5. MD5 processes its input

in 512-bit blocks and after each block is processed, the hash result so far is added to the previous hash result, but if it is going through the first block, they are added to constants instead. Since common passwords are rarely over 512-bits long, we know that MD5 will only process one 512-block and use the constants in the final steps. Since the hash results are split up into 4 parts, one could simply split our password hash (the one we want to break) and take the first part and subtract from it the first constant to essentially reverse one step of the hash function and use that to compare. This means that we can make an early comparison immediately when we get the first part of the hash result and if the hash part does not match, we can simply stop the operation and go on to the next candidate [33]. To illustrate:

1. We want to crack the hash 8743b52063cd84097a65d1633f5c74f5.
2. We take the first part and subtract from it a known constant 0x8743b520 - 0x67452301 = 0x1ffe921f.
3. When iterating candidate passwords with the MD5, we come to the last step where the first part of the hash result is modified, i.e., this step:

   ```
   II (a, b, c, d, w[ 4], S41, 0xf7537e82); /* 61 */
   ```

   , where $a$ is the first part of the hash result.
4. Instead of continuing the operation of the hash function, we do a comparison between $a$ and 0x1ffe921f.
   (a) If they match, we let the hash function continue and let it finish, whereafter we make another normal comparison between the full hashes.
   (b) If they do not match, we know that the hashes will not match and we can end the operation earlier.

**Precomputing:** When trying to break passwords, a lot of strategies tend to try very similar passwords in succession, e.g., changing characters to upper case, or adding numbers at the end of a password candidate. Depending on the hash function, these situations could allow a user to precompute or save the state of the hash function in advance and simply skip to that position for each iteration when trying for similar passwords. This means that potentially a lot of steps can be skipped early, meaning that a lot of time could be

saved over time [33]. An example of this was showed earlier in section 5.2.

## 5.4 Hashcat kernels

Another way to see the difference optimizations can do in password cracking is to use hashcats different kernels. Hashcat itself has built-in pure and optimized kernels for a lot of different hash-types. The pure kernels are a standard implementation of hash functions, without any added shortcuts or optimizations. The pure kernels just perform the hash function on the inputted password which then allows for comparison of a inputted hash. The optimized kernels use various techniques and optimizations to allow faster password cracking.

To demonstrate the effectiveness of these kernels, test runs were performed with the following three hash functions; MD5, SHA-256 and HMAC-SHA-256. For these test runs, we used the following input parameters to generate our results:

```
hashcat -m x -a 3 test.hash ?b?b?b?b?b?b?b -O
```

The parameters here are the same that hashcats own benchmark mode would use, expect for having the "-O" parameter which allows us to specifically use the optimized kernels. Each "?b" here means that we test all hexadecimal characters from 0x00 to 0xff. The example runs with these kernels were again run on the NVIDIA T550 Laptop GPU and their results can be seen in Figure 5.7, Figure 5.8 and Figure 5.9.

MD5

**Pure kernel**

```
Hash.Mode........: 0 (MD5)
Hash.Target......: 0e9aec569aefecabea2dbed59f5eb047
Time.Estimated...: Sun Dec  3 02:30:58 2023 (226 days, 17 hours)
Kernel.Feature...: Pure Kernel
Speed.#1.........: 3678.5 MH/s (3.69ms) @ Accel:512 Loops:64 Thr:32
Vec:1
```

**Optimized kernel**

```
Hash.Mode........: 0 (MD5)
Hash.Target......: 0e9aec569aefecabea2dbed59f5eb047
Time.Estimated...: Mon Jul 17 19:16:08 2023 (88 days, 9 hours)
Kernel.Feature...: Optimized Kernel
Speed.#1.........: 9436.0 MH/s (13.40ms) @ Accel:256 Loops:128 Thr:256
Vec:1
```

**Figure 5.7:** Performance comparison between hashcat's pure and optimized kernels for MD5 hashes.

SHA-256

**Pure kernel**

```
Hash.Mode........: 1400 (SHA2-256)
Hash.Target......: 7106773e79e86ee14d037ecbebf2d0b1b88fff0f290
93b96c05...ab4e0c
Time.Estimated...: Sat Jul 26 23:54:48 2025 (2 years, 99 days)
Kernel.Feature...: Pure Kernel
Speed.#1.........: 1005.5 MH/s (7.79ms) @ Accel:256 Loops:64 Thr:32
Vec:1
```

**Optimized kernel**

```
Hash.Mode........: 1400 (SHA2-256)
Hash.Target......: 7106773e79e86ee14d037ecbebf2d0b1b88fff0f290
93b96c05...ab4e0c
Time.Estimated...: Tue Dec 10 00:53:46 2024 (1 year, 235 days)
Kernel.Feature...: Optimized Kernel
Speed.#1.........: 1388.9 MH/s (11.32ms) @ Accel:64 Loops:128 Thr:128
Vec:1
```

**Figure 5.8:** Performance comparison between hashcat's pure and optimized kernels for SHA-256 hashes.

HMAC-SHA-256

**Pure kernel**

```
Hash.Mode........: 1460 (HMAC-SHA256 (key = $salt))
Hash.Target......: 1427e16fd6413439ebc947273323d36830c6f095296
e353c416...helios
Time.Estimated...: Tue Mar  7 05:52:04 2045 (21 years, 321 days)
Kernel.Feature...: Pure Kernel
Guess.Queue......: 1/1 (100.00%)
Speed.#1.........: 104.4 MH/s (9.08ms) @ Accel:4 Loops:64 Thr:256
Vec:1
```

**Optimized kernel**

```
Hash.Mode........: 1460 (HMAC-SHA256 (key = $salt))
Hash.Target......: 1427e16fd6413439ebc947273323d36830c6f095296
e353c416...helios
Time.Estimated...: Wed Mar  3 06:27:29 2027 (3 years, 316 days)
Kernel.Feature...: Optimized Kernel
Speed.#1.........: 590.3 MH/s (13.29ms) @ Accel:128 Loops:128 Thr:32
Vec:1
```

**Figure 5.9:** Performance comparison between hashcat's pure and optimized kernels for SHA-256 hashes.

One can immediately note that the speed is significantly higher with the optimized kernel compared to the pure kernel with all hash modes, as seen in Figure 5.10. MD5 is quite a simple algorithm and a run can be performed very quickly, having a base speed of 3678.5 MH/s in our tests with the pure kernel. Still, we can see that in this case the optimized kernel gives a significant boost in speed, having a speed of 9436.0 MH/s, which is around 2,5 times faster than the pure kernel. Since SHA-256 is more secure and time intensive than MD5, we can see that our time estimate has jumped up quite a lot (from days to years). Even though the speed increase with the optimized kernel here is only around 1,4 compared to the pure, the improvement is most welcome when looking at the time estimates. Being able to break a password 229 days earlier than before is a significant detail, especially if the importance of the password being cracked is severe. When observing the results from the HMAC-SHA-256 test runs one can start to see the outstanding potential of

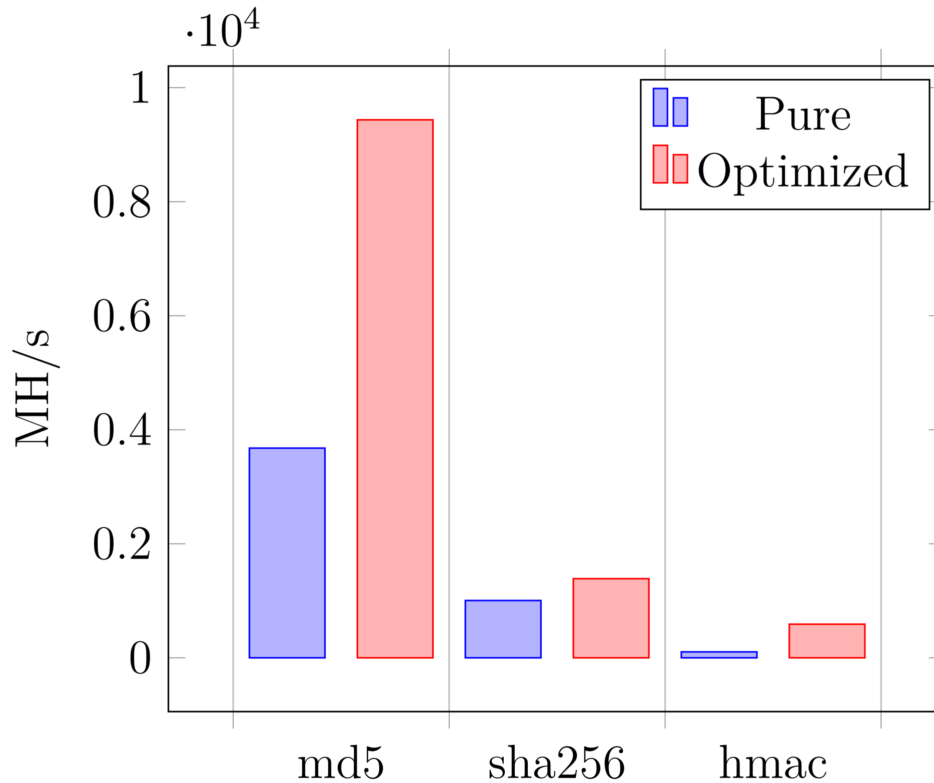



**Figure 5.10:** Bar graph comparing the speeds of the pure and optimized kernels for the different hash functions.

password cracking optimizations. With a speed increase of around 5,7 when cracking more complex passwords (like in this case), we can start seeing differences of a password recovery going from being infeasible to feasible. In most situations, security researchers would not want to waste computing resources to crack one password for over 21 years, but around 4 years sounds much more reasonable. These optimized kernels are not however the answer to all scenarios, since they do have limitations and drawbacks. Hashcat also notifies the drawbacks and benefits with the kernels with the following message when using the optimized kernel: "ATTENTION! Pure (unoptimized) backend kernels selected. Pure kernels can crack longer passwords, but drastically reduce performance." Since quite a few optimizations are dependent on password length, most of the optimized kernels have a much lower limit of password lengths they support. The pure kernel supports passwords up to 256 characters, while the optimized kernel generally supports passwords up to 55 characters, although there are exceptions, e.g., a Unix version of SHA-512crypt (hash mode 1800) only supports passwords with a maximum length of 16 characters [11].

Another thing to note about the kernels is that optimized kernels do not exist for all the supported hash types. Version 6.2.6 of hashcat was used for the tests done in this thesis.

This version has a total of 1189 kernels, were 448 of them were optimized kernels and 701 pure kernels. One should note, that exploits and optimizations need to be found in a hash function, in order for a optimized kernel to exist. This is why you cannot find optimized kernels for more advanced hash functions, e.g., PBKDF2-HMAC-SHA256 or scrypt.

## 5.5 Scrypt strengths and defence strategies

Scrypt is a reasonably new function that can be used to secure passwords, when compared to other popular choices, like PBKDF2 or SHA-2. Since it is a newer function, it has used past solutions as building blocks to be as secure as possible. The different parts complement each other in order to tackle as many security concerns as possible.

As we remember, in addition to its own internal functions ROMix BlockMix, scrypt utilizes Salsa 20/8, PBKDF2, HMAC and SHA-256. PBKDF2 in itself uses HMAC and SHA-256, which will allow scrypt to slow down computation in general and to make precomputation costlier [25]. As mentioned in an earlier chapter, PBKDF2 allows the user to choose how many iterations of the algorithm should be performed in order to produce the derived key. This means that even though computer hardware improves and machines get faster, users can always increase the iteration count of the algorithm. In Figure 5.11 we can see what kind of speed differences there are with PBKDF2 depending on how many iteration counts we have.

| Iteration count | Time to calculate hash |
|---|---|
| 1000 | 0,007s |
| 10000 | 0,010s |
| 100000 | 0,041s |
| 1000000 | 0,336s |
| 10000000 | 3,288s |

**Figure 5.11:** Table showing the time needed to hash one password using the shown iteration counts. Tests run on a ThinkPad P14s Gen 3 (Intel core i7-1260P, 32GB RAM) using openssl implementation of PBKDF2.

When PBKDF2 released, the recommended amount of iterations was at least 1000 [12]. As one can see from Figure 5.11, having only 1000 iterations means that we can calculate a hash almost instantly. However, as we increase the iteration counts, the computation time

needed to produce one key increases. This means that as long as you have a sufficiently large iteration count (based on today's hardware), you will be able counter attackers who simply rely on fast machines to brute-force all possible combinations.

The ability to resist precomputation comes from the use of salts in this case. Salts themselves are in this case used by HMAC-SHA-256, which is used by PBKDF2, which is in turn used by scrypt. Since salts are randomly generated and applied for each password, it means that even if you would precompute a password with a given hash function, you would never get a match, since the final hash is different due to the salt. An example of how identical passwords get different hashes can be seen in Figure 5.12.

| Password | Salt | Hash |
|---|---|---|
| abc123 | qw41t2 | 86af1d3564e04f4136db23a398e46592d3786d4b03675ed379e537c8ccc4c7d5 |
| abc123 | 63ho8j | 31d9ef942b477288de1c022904c3fb7694038433e3a61672304aced42568a428 |
| secret1337 | lfu6sd | 574c0f79b7fdbacb6a42c7a7a1fb0de7915b48909b4273baf0aefababefcff5b |
| secret1337 | 5d2sdf | ae9bfd2b400cc29a5d8a35cccb838d5c851b553dd503da06ba796ed3a38c5eac |
| secret1337 | 074b67 | 8d94a3d0faf656244baf313a2955fbc73314fed5c04a9dc4ba97f62c8ccbbd13 |

**Figure 5.12:** Table demonstrating how salts can completely transform the hash of identical passwords. In this case, HMAC-SHA-256 was used.

If an attacker were to potentially generate all possible hashes up to a certain length, including certain characters, salts would make this precomputation unusable. As one could see from the table, the same password had different hashes. This means that, for a precomputation attack to work in this case, the attacker would have to generate all possible hashes for all possible salts, which simply is computationally infeasible for default password lengths or longer.

The above mentioned defence mechanisms were available for PBKDF2, but now we shall go through scrypt own unique defensive features. Even though you can increase the iteration counts of PBKDF2 which makes the algorithm slower, the algorithm itself is still not that intensive to run. In other words, no especially demanding hardware is required to run the algorithm. This means that if an attacker would have access to a lot computation hardware, they would be able to run a lot of attacks in parallel and increase their password cracking speed. This is why scrypt allows the user, in addition of configuring the iteration counts, increase the CPU and memory cost of the algorithm.

In a basic run of scrypt, a large vector of pseudorandom data is generated, which has to be

accessed in a pseudo-random order and different points need to accessed multiple times. Basic implementations would want to keep all of this data in memory, so that the data could be accessed when needed. If a user does not have enough memory, they could instead implement the algorithm in a way that generates the data points on the fly when needed. This would allow the user to generate the hash with way less memory, but this approach does have its drawbacks. The amount of computations needed to calculate the hash would be significantly higher compared to the basic implementation of the algorithm. The reason for this is that each time a data point needs to be accessed, it has to be generated again from scratch. Also, since the same data points needs to be accessed multiple times, a lot of redundant computations need to be made, effectively wasting time. Another approach on optimizing scrypt could be to instead increase the memory used. This is however not to realistic, since the memory cost to create one hash is already very high. The amount of memory needed to parallelize the computation would be exceedingly expensive. One might wonder how then to configure scrypt's CPU and memory cost. Let us recall the input parameters of scrypt, specifically **N**, **r** and **p**. These are the parameters that will allow the user to fine tune the system requirements for the algorithm, so that calculating a hash is as demanding as possible. The main adjustment can be made with **N**, which directly modifies the CPU/memory cost. The blocksize is adjusted with **r**, which can be used to fine-tune the relative memory-cost. The parallelization factor is **p**, which in turn fine-tunes the relative CPU-cost [25]. To summarize, one of the main benefits with scrypt is that the user has the ability to make the time-memory trade-off of the function as costly as possible.

## 5.6 The effects of password length

When cracking password hashes, there are many factors that affect cracking speed, e.g., hash functions, exploits or machine hardware. Another factor that can have a substantial effect is the password length itself. Naturally, shorter passwords are easier to break than longer ones. When you increase the password length more significantly, the cracking time also will be much longer, since the time needed to crack the hash will increase exponentially. This essentially means, that as long as a hash function is not cryptographically broken, your password is secure as long as it is long enough.

To demonstrate the exponential increase in time required when increasing the character count of passwords, we did two password length tests with SHA-256 and scrypt. In these

tests, we assume that the attacker does not know the properties of the password and has to guess all possible characters. The test start with assuming the password length is 6 characters and will increment once it has gone through all candidates. The SHA-256 results can be seen in Figure 5.13 and the scrypt results can be seen in Figure 5.14. In these test, scrypt's iteration count was set to 1. The tests were performed on a NVIDIA T550 Laptop GPU.

Password length SHA-256 test

**Character length 6**

```
Hash.Mode........: 1400 (SHA2-256)
Hash.Target......: f62b68d564722f20784232c573a95b9fa9f42d2e
b781eb78839...e59dae
Time.Estimated...: Mon Jun  5 13:04:10 2023 (9 mins, 1 sec)
Guess.Mask.......: ?a?a?a?a?a?a [6]
Speed.#1.........:  1356.0 MH/s (10.73ms)
```

**Character length 7**

```
Hash.Mode........: 1400 (SHA2-256)
Hash.Target......: f62b68d564722f20784232c573a95b9fa9f42d2eb781
eb78839...e59dae
Time.Estimated...: Tue Jun  6 03:55:23 2023 (14 hours, 59 mins)
Guess.Mask.......: ?a?a?a?a?a?a?a [7]
Speed.#1.........:  1294.3 MH/s (11.45ms)
```

**Character length 8**

```
Hash.Mode........: 1400 (SHA2-256)
Hash.Target......: f62b68d564722f20784232c573a95b9fa9f42d2eb781
eb78839...e59dae
Time.Estimated...: Fri Aug  4 08:44:11 2023 (59 days, 19 hours)
Guess.Mask.......: ?a?a?a?a?a?a?a?a [8]
Speed.#1.........:  1284.3 MH/s (11.30ms)
```

**Character length 9**

```
Hash.Mode........: 1400 (SHA2-256)
Hash.Target......: f62b68d564722f20784232c573a95b9fa9f42d2eb781e
b78839...e59dae
Time.Estimated...: Thu Nov 26 09:58:09 2037 (14 years, 174 days)
Guess.Mask.......: ?a?a?a?a?a?a?a?a?a [9]
Speed.#1.........:  1379.5 MH/s (11.03ms)
```

**Figure 5.13:** Character length tests results on SHA-256 hashes using hashcat.

Password length scrypt test

**Character length 6**

```
Hash.Mode........: 8900 (scrypt)
Hash.Target......: SCRYPT:1024:1:1:aGVsaW9z:kjBsaLx+aY9B9ANP+ia7
P2gMeN...HWFBA=
Time.Estimated...: Sat Aug 12 01:57:57 2023 (64 days, 12 hours)
Guess.Mask.......: ?a?a?a?a?a?a [6]
Speed.#1.........:   131.9 kH/s (1.67ms)
```

**Character length 7**

```
Hash.Mode........: 8900 (scrypt)
Hash.Target......: SCRYPT:1024:1:1:aGVsaW9z:kjBsaLx+aY9B9ANP+ia7
P2gMeN...HWFBA=
Time.Estimated...: Tue Mar  6 00:51:34 2040 (16 years, 271 days)
Guess.Mask.......: ?a?a?a?a?a?a?a [7]
Speed.#1.........:   132.2 kH/s (1.69ms)
```

**Character length 8**

```
Hash.Mode........: 8900 (scrypt)
Hash.Target......: SCRYPT:1024:1:1:aGVsaW9z:kjBsaLx+aY9B9ANP+ia7
P2gMeN...HWFBA=
Time.Estimated...: Sun Feb  3 00:29:41 3546 (1522 years, 239 days)
Kernel.Feature...: Pure Kernel
Guess.Mask.......: ?a?a?a?a?a?a?a?a [8]
Speed.#1.........:   138.1 kH/s (1.73ms)
```

**Character length 9**

```
Hash.Mode........: 8900 (scrypt)
Hash.Target......: SCRYPT:1024:1:1:aGVsaW9z:kjBsaLx+aY9B9ANP+ia7
P2gMeN...HWFBA=
Time.Estimated...: Next Big Bang (143052 years, 85 days)
Guess.Mask.......: ?a?a?a?a?a?a?a?a?a [9]
Speed.#1.........:   139.6 kH/s (1.69ms)
```

**Figure 5.14:** Character length tests results on scrypt hashes using hashcat.

We kept incrementing the password length for the attacks until hashcat gave an error on the mask length. The error message in this case was "Integer overflow detected in keyspace of mask: ?a?a?a?a?a?a?a?a?a?a", which comes from the mask being too long, thus leading to an integer overflow.

## 5.7 Cracking speed differences for hash functions

To make a final comparison between hash functions, we made some test runs to compare what kind of cracking speeds we get for different scenarios. For these tests, we performed hashcat mask attacks against password hashes which were generated with different hash functions. All hashes were derived from the same password "Pass123!". In the results table, we call the different scenarios cases. We used the following mask attack parameters to generate our cases:

```
Case1:  ?u?l?l?l?d?d?d?s
Case2:  -1 ?l?u -2 ?d?s ?1?1?1?1?2?2?2?2
Case3:  ?a?a?a?a?a?a?a?a
```

The results can be seen in Figure 5.15. The tests were performed on a NVIDIA T550 Laptop GPU.

| Pass123! | SHA256 | HMAC-SHA256 | PBKDF2 | Scrypt |
|---|---|---|---|---|
| Case1 | 12 secs | 25 secs | 9 hours, 31 mins | 1 day, 13 hours |
| Case2 | 4 hours, 54 mins | 11 hours, 47 mins | 1 year, 283 days | 6 years, 140 days |
| Case3 | 56 days, 5 hours | 143 days, 5 hours | 492 years, 167 days | 1836 years, 333 days |

**Figure 5.15:** Table demonstrating the cracking speed differences of hash functions.

Let us explain further the idea behind the different cases. In Case1, we know the exact form of the password, i.e., we know that it starts with an upper case letter, followed by three lower case letters, followed by three numbers and finally a special character. Naturally, there is a clear difference in cracking speeds, but all of the cracking speeds are quite feasible for Case1. In Case2, we know that the four first characters are letters and that the following four characters are either numbers or special characters. Here we can start see some clear differences, where SHA-256 and HMAC-SHA256 have feasible

cracking speeds, while PBKDF2 and scrypt start taking years. In Case3, we do not know anything about the password, meaning we have to try all possible character combinations. This time, SHA256 and HMAC-SHA256 start taking quite some time, while the cracking speeds of PBKDF2 and scrypt are infeasible.

# 6 Discussions

As we have seen in this thesis, passwords can be secured in many different ways. Hash functions, salts, key derivation functions, stream ciphers are just some concepts that have been listed. Even though these techniques are public knowledge and there already exists free implementations of them, we still hear about big data leaks on a regular basis.

When it comes to attacks and data leaks, a common problem that usually comes up is that security is unfortunately often not the priority when developing software, which can lead to a lot of different issues. When talking about accounts credentials, storing passwords in plaintext without any security in place is a very severe error one can make when developing a system. Although this happens less frequently nowadays, cases have been recorded recently, like the Facebook and Instagram leak[21] which was mentioned earlier in Chapter 2.

A common situation one comes across with password security is that, even though it is in place, one can see that minimum effort has been given to it, e.g., passwords are only hashed with a basic hash function. This technically better than no security at all, but even a novice can break this depending on the situation. As an example, the LinkedIn hack that happened 2012 demonstrated how easily poor security can be broken. It was originally reported that 6.5 million accounts were leaked, but it turned out later in 2016 that the number of leaked accounts was 117 million instead [36]. Reportedly, LinkedIn was using at the time only simple SHA-1 to generate hashes of the passwords. SHA-1 is today considered to be cryptographically broken, being susceptible to collision attacks and more [14]. Even though the attack happened 2012, LinkedIn would have had ample time to revise their security. Reports and research had already been done in 2005, showing that collision attacks were possible and that finding collisions was faster than brute-force attacks [30]. In 2011, SHA-1 was starting to get deprecated in some contexts by NIST [1] and NIST's stance on the algorithm today is that "We recommend that anyone relying on SHA-1 for security migrate to SHA-2 or SHA-3 as soon as possible" [4]. The LinkedIn article explaining the attack [36] goes on to explain that LinkedIn has since then enhanced their protection by adding salts to their passwords. Salts will indeed increase the security of the passwords, notably adding resistance to lookup table and rainbow table attacks. How salts are implemented is not delved deeper into. Depending on how the salts are

implemented, e.g., by simply adding random salts to SHA-1 or using HMAC makes a big difference. Since we do not know the implementation, LinkedIn could either have successfully plugged the problem, or just made a minimum effort solution, which may be broken yet again.

User do not always have a clear way to protect their credentials, especially since a lot of popular websites and services have poor security. Switching to a more secure platform might not be possible for a user due to various reasons, e.g., not being able to switch bank account and being stuck using the banks same web service. Even if a site is only using a single hash function, as in LinkedIn case, a really strong password might still protect against an attack. As long as the hash function that is used is not completely broken, e.g., allowing collision or other ways to circumvent the intended security, a strong password may be enough. As it has been demonstrated in this thesis, larger character sets and longer character counts increase the number of needed guesses exponentially. Having a long password that contains uppercase, lowercase, numbers and special characters is usually a good start. A demonstration of this security was shown in the previous chapter, in Figure 5.13 and Figure 5.14.

We now know that we can secure our accounts with long and complicated passwords, but another problem may arise if a user reuses the same password for multiple services. This problem can be solved by using password managers. Password managers allow the user to generate automatically unique long and complicated passwords and save them in a vault for future use. This vault is then protected by a master password, meaning that for daily use, a user only needs to remember one password. As long as the vault stays safe, the user may enjoy increased security with the added benefit of not having to remember multiple complicated passwords. Since each service will now have a complicated password, it will be very hard to break a password and even if a service was breached, only one password would be compromised. Password managers usually divide into two types, a local version and a cloud version. These types have their benefits and drawbacks. The cloud versions have the benefit of syncing your passwords to a cloud, meaning that you can use the manager on different devices. This however implies that you must trust in the password manager provider to keep your passwords safe. This trust was shaken in 2022, when the widely popular password manager LastPass was hacked, which lead to various damage, including customer data being stolen [35]. The other option would be to use a local password manager, which saves passwords locally on a device. The passwords would also be encrypted and protected by a master password, but they could only be

accessed through the device which they are saved on. This can be quite inconvenient depending on how often a user switches devices, but at least the user would have control and responsibility of their own passwords.

# 7 Conclusions

In this thesis, we analyzed password hashes from the perspective of a defender and an attacker. From the defender's perspective, we went through common building blocks for password encryption, scrypt and compared different function and how well they fared against attacks. From an attacker's perspective, we went through hashcat, different attack types, available exploits and tested how these techniques fared against different hash functions.

To answer our research question "What kind of password hashing techniques have evolved from the viewpoints of a defender and an attacker?", we noted the following: When hash functions were quite simple, different attacking tools were developed to automate attacks. Attack strategies like brute-force attacks, lookup table or rainbow table attacks were used. To combat table attacks, salts were developed and combined to existing solutions in order to prevent pre-existing tables. In order to make attacks more effective, attackers started to create many different attack types, like dictionary or combinator attacks, which were more intelligent attacks compared to the simpler brute-force attacks. To make the attacks less effective and slower, hash functions using more iteration counts to generate a hash were created. Attackers tried to circumvent this with different parallelization techniques, but hash functions have been developed that have demanding memory requirements, which in turn makes parallelization very difficult to perform.

When it comes to our research question "What kind of observations can be made when studying the implementations of the hashing algorithms and the tools that the attackers use for the hashes?", we noted the following: There have been many different approaches to develop hash functions throughout the years, but a lot of successful ones build on previous solutions, in order to combine many desirable properties into a new one. The attackers have tried many different approaches with their attacks against password hashes. In order to make as many attacks as fast as possible, different exploits have been found and developed for different hash functions. Major vulnerabilities have been found in different hash functions, which have meant that some have been deemed cryptographically broken. Many service providers still use older hash functions, even though attacks against them can be made very quickly. The reason for this is that these older functions have stood the test of time and have not been cryptographically broken, which only shows that they were

designed exceptionally well for the time.

The scene for password security is constantly evolving. Attackers are constantly looking for any way to get an advantage. Defenders also have to be alert at all times and be ready to update or replace current security solutions, in order to keep their system secure. Users should also remember that they have to take responsibility of their passwords. It does not matter how good the security may be for a system, if the user employs bad password practices.

# References


[1] E. Barker and A. Roginsky. “Transitions: Recommendation for transitioning the use of cryptographic algorithms and key lengths”. In: *NIST Special Publication* 800 (2011), 131A.

[2] I. Bentov, A. Gabizon and A. Mizrahi. “Cryptocurrencies Without Proof of Work”. In: *Financial Cryptography and Data Security*. Ed. by J. Clark, S. Meiklejohn, P. Y. Ryan, D. Wallach, M. Brenner and K. Rohloff. Berlin, Heidelberg: Springer Berlin Heidelberg, 2016, pp. 142–157. ISBN: 978-3-662-53357-4.

[3] D. J. Bernstein. “The Salsa20 family of stream ciphers”. In: *Lecture Notes in Computer Science, volume 4986. New stream cipher designs: the eSTREAM finalists* (2008), pp. 84–97.

[4] C. Boutin. *NIST Retires SHA-1 Cryptographic Algorithm*. News article. 2022. URL: https://www.nist.gov/news-events/news/2022/12/nist-retires-sha-1-cryptographic-algorithm (visited on 1 June 2023).

[5] Q. Dang. *FIPS-180-4: Secure hash standard*. Tech. rep. Federal Information Processing Standards Publications (FIPS PUBS) 180-4. 2015. DOI: https://doi.org/10.6028/NIST.FIPS.180-4.

[6] W. Diffie and M. Hellman. “Special Feature Exhaustive Cryptanalysis of the NBS Data Encryption Standard”. In: *Computer* 10.6 (1977), pp. 74–84. DOI: 10.1109/C-M.1977.217750.

[7] D. Eastlake 3rd and T. Hansen. *US secure hash algorithms (SHA and SHA-based HMAC and HKDF)*. Tech. rep. 2011.

[8] D. C. Feldmeier and P. R. Karn. “Unix password security - ten years later”. In: *Conference on the Theory and Application of Cryptology*. Springer. 1989, pp. 44–63.

[9] M. S. Ferdous, M. J. M. Chowdhury, M. A. Hoque and A. Colman. *Blockchain Consensus Algorithms: A Survey*. 2020. arXiv: 2001.07091 [cs.DC].

[10] P. A. Grassi, J. L. Fenton and M. E. Garcia. *Digital Identity Guidelines*. Tech. rep. NIST Special Publication 800-63-3. 2017. DOI: https://doi.org/10.6028/NIST.SP.800-63-3.

[11] Hashcat. *Hashcat advanced password recovery*. Website and wiki. URL: https://hashcat.net/hashcat/ (visited on 6 Mar. 2023).

[12] B. Kaliski. *PKCS #5: Password-Based Cryptography Specification Version 2.0*. IETF RFC 2898. Sept. 2000. DOI: 10.17487/RFC2898. URL: https://www.rfc-editor.org/info/rfc2898.

[13] R. F. Kayser. "Announcing request for candidate algorithm nominations for a new cryptographic hash algorithm (SHA-3) family". In: *Federal Register* 72.212 (2007), p. 62.

[14] G. Leurent and T. Peyrin. "SHA-1 is a shambles: First chosen-prefix collision on SHA-1 and application to the PGP web of trust". In: *Proceedings of the 29th USENIX Conference on Security Symposium*. 2020, pp. 1839–1856.

[15] R. McMillan. *The World's First Computer Password? It Was Useless Too*. News article in: *Wired*. 2012. URL: https://www.wired.com/2012/01/computer-password/ (visited on 14 June 2023).

[16] K. Moriarty, B. Kaliski and A. Rusch. *PKCS #5: Password-Based Cryptography Specification Version 2.1*. IETF RFC 8018. Jan. 2017. DOI: 10.17487/RFC8018. URL: https://www.rfc-editor.org/info/rfc8018.

[17] R. Morris and K. Thompson. "Password security: A case history". In: *Communications of the ACM* 22.11 (1979), pp. 594–597.

[18] U. Mukhopadhyay, A. Skjellum, O. Hambolu, J. Oakley, L. Yu and R. Brooks. "A brief survey of Cryptocurrency systems". In: *2016 14th Annual Conference on Privacy, Security and Trust (PST)*. 2016, pp. 745–752. DOI: 10.1109/PST.2016.7906988.

[19] National Institute of Standards and Technology. *FIPS-180-2: Secure hash standard*. Tech. rep. Federal Information Processing Standards Publications (FIPS PUBS) 180-2. 2002. URL: https://csrc.nist.gov/files/pubs/fips/180-2/final/docs/fips180-2.pdf.

[20] National Institute of Standards and Technology. *NIST's Policy on Hash Functions - March 2006*. Policy update. 2006. URL: https://csrc.nist.gov/projects/hash-functions/nist-policy-on-hash-functions (visited on 14 May 2023).

[21] L. H. Newman. *Facebook Stored Millions of Passwords in Plaintext—Change Yours Now*. News article in: *Wired*. 2019. URL: https://www.wired.com/story/facebook-passwords-plaintext-change-yours/ (visited on 9 May 2023).

[22] NordPass. *Top 200 most common passwords*. Website. 2022. URL: https://nordpass.com/most-common-passwords-list/ (visited on 8 May 2023).

[23] W. Penard and T. van Werkhoven. "On the secure hash algorithm family". In: *Cryptography in Context; Wiley: Hoboken, NJ, USA,* (2008), pp. 1–18.

[24] C. Percival. *Stronger key derivation via sequential memory-hard functions*. In Proc. Tech. BSD Conf. (BSDCan'09). May 2009. URL: https://www.tarsnap.com/scrypt/scrypt.pdf.

[25] C. Percival and S. Josefsson. *The scrypt Password-Based Key Derivation Function*. IETF RFC 7914. Aug. 2016. DOI: 10.17487/RFC7914. URL: https://www.rfc-editor.org/info/rfc7914.

[26] V. Petkauskas. *Thomson Reuters collected and leaked at least 3TB of sensitive data*. News article. 2022. URL: https://cybernews.com/security/thomson-reuters-leaked-terabytes-sensitive-data/ (visited on 9 May 2023).

[27] Polybius. *The Histories of Polybius, Vol. 1*. Project Gutenberg, 2013.

[28] K. Raeburn. *Advanced encryption standard (AES) encryption for Kerberos 5*. Tech. rep. 2005.

[29] R. L. Rivest. *The MD4 Message-Digest Algorithm*. IETF RFC 1320. Apr. 1992. DOI: 10.17487/RFC1320. URL: https://www.rfc-editor.org/info/rfc1320.

[30] B. Schneier. *Schneier on Security: Cryptanalysis of SHA-1*. Blog post. 2005. URL: https://www.schneier.com/blog/archives/2005/02/cryptanalysis_o.html (visited on 1 June 2023).

[31] Z. Shi, B. Zhang, D. Feng and W. Wu. "Improved Key Recovery Attacks on Reduced-Round Salsa20 and ChaCha". In: *Information Security and Cryptology – ICISC 2012*. Ed. by T. Kwon, M.-K. Lee and D. Kwon. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 337–351. ISBN: 978-3-642-37682-5.

[32] W. Stallings. *Cryptography and Network Security: Principles and Practice*. 6th edition. USA: Prentice Hall Press, 2013. ISBN: 0133354695.

[33] J. Steube. "Optimizing computation of Hash-Algorithms as an attacker". Presentation slides, In: PasswordsCon. 2013. URL: https://hashcat.net/events/p13/js-ocohaaaa.pdf.

[34] Tarsnap. *Tarsnap: Online backups for the truly paranoid*. Website and wiki. 2009. URL: https://www.tarsnap.com/scrypt.html (visited on 4 Apr. 2023).

[35] D. Winder. "Why You Should Stop Using LastPass After New Hack Method Update". In: (3 Mar. 2023). News article in: *Forbes*. URL: https://www.forbes.com/sites/daveywinder/2023/03/03/why-you-should-stop-using-lastpass-after-new-hack-method-update/ (visited on 3 Mar. 2023).

[36] T. C. Wood. *The LinkedIn Hack: Understanding Why It Was So Easy to Crack the Passwords*. News article. 2016. URL: https://www.linkedin.com/pulse/linkedin-hack-understanding-why-so-easy-crack-tyler-cohen-wood (visited on 1 June 2023).