Middle East Technical University

Informatics Institute

A COMPREHENSIVE SURVEY ON PASSWORD STORAGE SECURITY

Advisor Name: Cihangir TEZCAN, Assoc. Prof. Dr.

(METU)

Student Name: Murat Can ORUNTAK

(Cyber Security, MSc without Thesis)

June 2025

TECHNICAL REPORT

METU/ II-TR-2025-

Orta Doğu Teknik Üniversites

Enformatik Enstitüsü

GÜVENLİ PAROLA SAKLAMA HAKKINDA KAPSAMLI BİR İNCELEME

Danışman Adı: Cihangir TEZCAN, Doç. Dr.

(ODTÜ)

Öğrenci Adı: Murat Can ORUNTAK

(Siber Güvenlik, Tezsiz Yüksek Lisans)

Haziran 2025

TEKNİK RAPOR

METU/ II-TR-2025-

REPORT DOCUMENTATION PAGE

1. AGENCY USE ONLY (Internal Use)	**2. REPORT DATE** 12.06.2025

3. TITLE AND SUBTITLE

A COMPREHENSIVE SURVEY ON PASSWORD STORAGE SECURITY

4. AUTHOR (S) Murat Can ORUNTAK	**5. REPORT NUMBER (Internal Use)**

6. SPONSORING/ MONITORING AGENCY NAME(S) AND SIGNATURE(S)

Non-Thesis Master's Programme, Department of Cyber Security, Informatics Institute, METU

Advisor: Cihangir TEZCAN, Assoc. Prof. Dr. Signature:

7. SUPPLEMENTARY NOTES

8. ABSTRACT

Weak password storage practices still are a major vulnerability in modern systems. Despite the rise of alternative authentication methods, passwords remain the most dominant form of access control across various sectors which makes their secure storage essential. This report provides a comprehensive analysis of password storage security, with a focus on common malpractices, attack vectors, established industry standards, and potential future directions. It examines the historical evolution of password storage, highlighting notable breaches and vulnerabilities caused by weak or outdated hashing mechanisms and poor implementation decisions. Key concepts such as hashing and salting are discussed in the context of secure storage architecture, alongside additional measures like peppering and honeywords. Widely accepted standards including NIST guidelines, OWASP recommendations, and ISO/IEC frameworks are reviewed to outline best practices. Finally, the report explores emerging solutions such as passkeys and two factor authentication, aiming to align secure storage mechanisms with modern authentication needs. Through this structured exploration, the study aims to serve as a comprehensive and practical guide to password storage security.

9. SUBJECT TERMS	**10. NUMBER OF PAGES** 67

A COMPREHENSIVE SURVEY ON PASSWORD STORAGE SECURITY

Abstract

Weak password storage practices still are a major vulnerability in modern systems. Despite the rise of alternative authentication methods, passwords remain the most dominant form of access control across various sectors which makes their secure storage essential. This report provides a comprehensive analysis of password storage security, with a focus on common malpractices, attack vectors, established industry standards, and potential future directions. It examines the historical evolution of password storage, highlighting notable breaches and vulnerabilities caused by weak or outdated hashing mechanisms and poor implementation decisions. Key concepts such as hashing and salting are discussed in the context of secure storage architecture, alongside additional measures like peppering and honeywords. Widely accepted standards including NIST guidelines, OWASP recommendations, and ISO/IEC frameworks are reviewed to outline best practices. Finally, the report explores emerging solutions such as passkeys and two factor authentication, aiming to align secure storage mechanisms with modern authentication needs. Through this structured exploration, the study aims to serve as a comprehensive and practical guide to password storage security.

Table of Contents

List of Figures

List of Tables

List of Abbreviations

API	Application Programming Interface
DoS	Denial of Service
FIPS	Federal Information Processing Standards
GDPR	General Data Protection Regulation
HMAC	Hash-based Message Authentication Code
ISO	International Organization for Standardization
KDF	Key Derivation Function
MD5	Message Digest 5
MFA	Multi-Factor Authentication
NIST	National Institute of Standards and Technology
OWASP	Open Worldwide Application Security Project
OPT	One Time Password
PBKDF2	Password-Based Key Derivation Function 2
PCI-DSS	Payment Card Industry Data Security Standard
RFC	Request for Comments
SHA-1	Secure Hash Algorithm 1
SHA-2	Secure Hash Algorithm 2
TLS	Transport Layer Security
TOTP	Time-based One-Time Password
URL	Uniform Resource Locator
W3C	World Wide Web Consortium

1.Introduction

Authentication is considered as one of the most critical security aspects of information systems. It helps to verify user identities and ensures that accessing a specific digital resource is restricted to only authorized individuals [1]. Even though alternative authentication like passkeys, one-time passwords (OTPs), and keychain-based solutions have been developed in recent years, password-based authentication remains as the most widely used method [2]. From personal accounts to enterprise-level systems, passwords serve as the first line of defense in verifying user identities and protecting digital resources. Despite their commonness, using passwords as the main authentication method comes with a significant security liability if not managed and stored properly [3].

One of the most common challenges in this area is ensuring secure storage of passwords, particularly on servers and databases that may be compromised during security breaches [3]. In modern systems, plain text passwords should never be stored directly; instead, they need to be transformed to a hash output using cryptographic hash functions which are basically one-way algorithms that generate fixed length outputs regardless of the input size [4]. While storing passwords, it is critical to use a secure and computationally intensive hash function. However, the strength of a password storage mechanism is not only about selecting the appropriate hash function but also about correctly following security procedures at every step, including the proper implementation of additional measures such as salting and peppering. Improper use of outdated and unsecure algorithms like MD5, SHA-1, along with salting misconfigurations such as using reused, static, or easy to predict salts can significantly decrease the resistance against offline attacks. In such attacks attackers get access to the hashed passwords and attempt to reverse the hashes using precomputed tables which are known as rainbow tables or by leveraging different brute force strategies such as dictionary attacks [5].

1.1 Research Questions and Objectives

This research aims to systematically investigate how to store passwords safely from start to finish. First the question of why secure storage matters is reviewed and some of the well-known data breaches in recent years is examined. Next hash functions are described briefly and their security properties and the reason why they are difficult to break explained. Common hashes such as MD5, SHA-1, and SHA-256 are then compared, and their strengths and weaknesses are presented. Special password-hashing tools like bcrypt, scrypt, and Argon2 are also covered, and the ways how they slow attackers are noted.

Afterward, the use of additional measures such as salting and peppering is examined in order to explain how they make passwords harder to crack, and the main attack types like offline and online are assessed along with appropriate countermeasures. Key guidelines issued by organizations like NIST and OWASP are then summarized, and newer ideas such as passwordless logins and multi-factor authentication are briefly evaluated.

To summarize, this paper brings all these topics together, to provide a clear step by step guideline for secure password storage in modern systems.

1.2 History of Password and Password Storage

Passwords have served as the main tools for verifying identity since ancient times and they have faced similar challenges like we are facing today. Throughout the time people have used vocal passwords, written passwords and hard to forge seals to confirm their identity, confirm other people's identity and verify critical messages. Even the concept of automated authentication dates back thousands of years. For instance, key-based locks, were utilized by ancient Egyptians to secure access with minimal human involvement. In today's digital age, where computer systems operate at massive scale, authentication with an unattended and scalable system has become crucial and relying on automated mechanisms for security becomes a necessity. Interestingly, the idea of password-based, automated access control has deep cultural roots as well. A notable example appears in the classic Middle Eastern tale of *Ali Baba and the Forty Thieves*, where a magical phrase "Open, Sesame" served as a password to unlock a hidden treasure cave, illustrating an early metaphor for access control via secret credentials [6]. Besides the folkloric tales the earliest

documented use found in the Roman military, where a "watchword" (tessera) was passed along the guards to verify identity (documented by Greek historian Polybius around 150 BC) to control access to military camps[1], but these were vulnerable to various attack vectors similar to the first digital passwords at MIT in 1961, which were stored in plaintext and easily stolen by a doctoral student "Allan Scherr" in 1962 [6].

Just as watchwords needed constant rotation to stay secure, early computer systems struggled with similar challenges and repeated breaches happened until cryptography offered solutions we know today. In the 1970's, researchers introduced one-way hashing with the UNIX password hashing mechanism, employing crypt(3), a one-way hash function based on DES, to securely transform plaintext into fixed-length strings [7] which scrambled passwords into unreadable codes. And later they produced the concept of salting [7] which added random data to each password before hashing to prevent brute force attacks. These innovations solved some of the ancient goals by ensuring secrets could not be reused or reversed if intercepted. Yet human tendencies to insecure applications persist and people still choose weak passwords, and responsible entities still follow the inappropriate practices to store passwords securely [8].

While today's shift toward passphrases and biometrics follows up an ancient problem which is: "Authenticating identity without burdening memory" these methods are still facing critical issues like single point of failure, efficient deployment, and usability. Even though passwords have some known weaknesses they remain as the most widely used form of authentication due to their balance, usability and deployability [2].

1.3 Examples of Weak Storage Practices

The importance of secure password storage has evolved significantly over the past two decades, particularly in response to large-scale data breaches that exposed millions of user credentials due to poor storage practices. In the early days of the internet, it was common for systems to store user passwords in plain text. As awareness of security risks grew, systems began to implement cryptographic hashing to store passwords more securely. However, the choice of

[1] https://en.wikipedia.org/wiki/Password

hash function and the correct use of salting have remained ongoing challenges that are mostly overlooked or improperly implemented by developers [3]. When recent breaches have been examined, countless critical incidents can be seen which include:

RockYou Breach (2009)

RockYou, a social media application provider, exposed 32 million user credentials stored entirely in plain text in 2009. This devastating failure became one of the definitive case studies in password storage malpractices. Attackers exploited a decade old SQL injection vulnerability[2] to extract the database, revealing not only RockYou account passwords but also credentials for linked platforms (e.g., Facebook, MySpace) due to unencrypted partner integrations. Moreover, RockYou not only stored plaintext user credentials without any cryptographic operation but also user passwords were emailed in plaintext during account recovery and special characters were disallowed while creating passwords and the company took days to notify users after the incident[3].

Analysis of the leaked data also revealed systemic user behavior flaws such as:

- 30% of passwords were ≤ 6 characters.
- The password "123456" appeared 290,000 times (0.9% of total) [9].

Under Armour - MyFitnessPal Data Breach (2018)

The 2018 breach of Under Armour's nutrition-tracking platform, MyFitnessPal, affected approximately 150 million user accounts which made it one of the largest credential leaks of this decade.

Attackers gained access to the database in February 2018 via an undisclosed vulnerability (potentially SQL injection or credential exploitation) and exfiltrated usernames, email addresses, IP addresses, and hashed passwords. Crucially, MyFitnessPal used a hybrid hashing system due to a recent migration. While they use bcrypt for new passwords, legacy accounts remained protected by unsalted SHA-1 which is a deprecated function with known flaws since 2005. This

[2] https://www.w3schools.com/sql/sql_injection.asp
[3] https://en.wikipedia.org/wiki/RockYou

inconsistency created a critical attack surface: SHA-1's lack of salting allowed attackers to reverse engineer 60 million passwords rapidly via rainbow tables, even though bcrypt's computational intensity slowed cracking success but still not completely prevented the incident[4].

The failures of the institution also have amplified the damage. Instead of migrating to bcrypt for newer accounts, Under Armour retained SHA-1 hashes for inactive users rather than forcing system wide resets or cryptographic upgrades. Cryptographer Matthew Green assessed the situation as: “Amateur hour,” noting that Under Armour seemed to have migrated “from something terrible, SHA-1, to something less terrible, bcrypt,” but kept the previous hashes for users who had not logged in during the transition[5].

Even though Under Armour’s segmentation prevented theft of financial or biometric data, the incident showed how partial security upgrades create a false sense of security. As concluded in the wired article[6], the breach was "worse than it had to be".

Key takeaways from this breach can be listed as:

1) Legacy hash function usage should be prevented.

2) Single and standardized hashing policy should be maintained.

3) Password resets during cryptographic upgrades should be enforced.

LinkedIn Data Breach (2012)

The LinkedIn data breach happened in June 2012 stands as one of the most important cases in the history of password security failures. Almost 6.5 million hashed passwords leaked on a Russian hacker forum. However, it was later revealed that the total number of affected accounts reached nearly 117 million. Attackers exploited a SQL injection vulnerability (a well-documented attack vector since the early 2000s) to extract password hashes from production databases[7]. Even though LinkedIn did use hashing with SHA-1, they did not use salting. The absence of salting

[4]https://www.forbes.com/sites/tonybradley/2018/03/30/security-experts-weigh-in-on-massive-data-breach-of-150-million-myfitnesspal-accounts/

[5] https://www.wired.com/story/under-armour-myfitnesspal-hack-password-hashing/

[6] https://www.wired.com/story/under-armour-myfitnesspal-hack-password-hashing/

[7] https://www.sentinelone.com/blog/blast-past-2012-linkedin-breach-dumps-100m-additional-records/

rendered the stored credentials highly vulnerable to offline attacks using rainbow tables and brute force methods[8]. Security researchers confirmed that 90% of the hashed passwords were reversed within 72 hours due to this flaw. The breach's severity was worsened by credential reuse patterns. Since many users reused the same credentials in different platforms, attackers were then able to log in to other services (Dropbox, Gmail etc.). This incident demonstrates how vulnerabilities in one system can cascade across the wider authentication ecosystem[9].

In summary the 2012 LinkedIn breach stands as a key example in cybersecurity area, highlighting the critical importance of applying fundamental protection measures while storing user credentials. It demonstrates how overlooking basic security practices can lead to devastating consequences and emphasizes the shifting responsibilities shared by users, platforms, and regulators in an increasingly connected digital environment [10].

1.4 User Behavior and Common Malpractices

Creating a secure password authentication system depends on two main parameters: How service providers store passwords and end users credential management practices. Even though now we have known for many years which password storage practices are secure, and which are dangerous (like improperly used MD5 hashing or plaintext) many services providers still make the same mistakes due to lack of awareness or underestimating potential consequences.

While users certainly play a role in password security, the first line of defense lies with service providers and their password storage practices. Unfortunately, various organizations continue to implement dangerously outdated methods that leave user credentials vulnerable. There are some essential findings from recent studies:

- 40% of organizations use a spreadsheet or Word document to store user credentials in a fully readable format[10].

[8]https://www.bitdefender.com/en-us/blog/hotforsecurity/6-5-million-linkedin-hashed-passwords-exposed-change-your-login-credentials-now
[9] https://www.informationweek.com/cyber-resilience/linkedin-hack-why-breach-is-a-wake-up-call-for-users
[10] https://passcamp.com/blog/dangers-of-storing-and-sharing-passwords-in-plaintext/

- 28% of organizations use a shared server or a USB stick to store user credentials[11].
- 11 websites in Alexa's top 500 list were storing passwords in plaintext, highlighting that even prominent sites can have insecure practices [11] These websites are shown in the Table 1.

Table 1. 11 websites from Alexa's top 500 list which stores user credentials as plaintext.

Site Address	Rank	Category	Country
fc2.com	58	Business Services	Japan
lists.wikimedia.org	135	Cooperatives	United States
badoo.com	164	Cooperatives	Italy
espncricinfo.com	188	Arts & Entertainment	India
liveinternet.ru	192	Arts & Entertainment	Russia
rutracker.org	301	Arts & Entertainment	Russa
corrierre.it	380	Arts & Entertainment	Italy
extratorrent.cc	415	Arts & Entertainment	India
jrj.com.cn	456	Investing	China
kooora.com	464	Arts & Entertainment	Saudi Arabia
xywy.com	484	Healthcare	China

- In 2019, Meta disclosed that it had stored millions of user passwords in fully readable format, leading to a 91 million euro fine by the EU in 2024[12].
- In April 2018, T-Mobile Austria publicly admitted via Twitter that it stored customer passwords in plaintext and defended this practice with: “We store your passwords in plain text, but don't worry, our security is amazingly good!”[13].
- A study evaluating open-source Content Management Systems (CMS) found that 14.29% of the relevant CMS did not apply salt while hashing which leaves their password hashes susceptible to rainbow table attacks [8].
- The same study observed that %36.73 of the relevant CMS did not iterate their hashes, making them more susceptible to password guessing attacks [8].

[11] https://onmsft.com/news/survey-says-40-of-organizations-store-admin-passwords-in-a-word-document

[12]https://www.reuters.com/technology/eu-privacy-regulator-fines-meta-91-million-euros-over-password-storage-2024-09-27

[13]https://www.linkedin.com/pulse/t-mobile-austria-we-store-your-passwords-plain-text-martin-st%C3%B6fler

- The remaining CMS platforms that do apply iterations mostly tend to choose the number of iterations randomly or without a consistent method [8].

While institutional failures in password storage create systemic vulnerabilities, user behavior also remains as a critical weak link in the system. Even with following best practices while storing passwords, human tendencies toward convenience such as password reuse, predictable patterns or weak password choices undermines the security of password authentication. Studies reveal that 68% of breaches involve human errors or misuse [12] highlighting how user malpractices amplify risks. To prevent this type of weakness, institutions should enforce their users to secure password choices.

Results from the study conducted by Passcape Software [9] about RockYou breach are shown in Figure 1. Figure 2. and Figure 3.

- The most popular password was “123456” used by nearly 290,000 users amongst 32 million users.

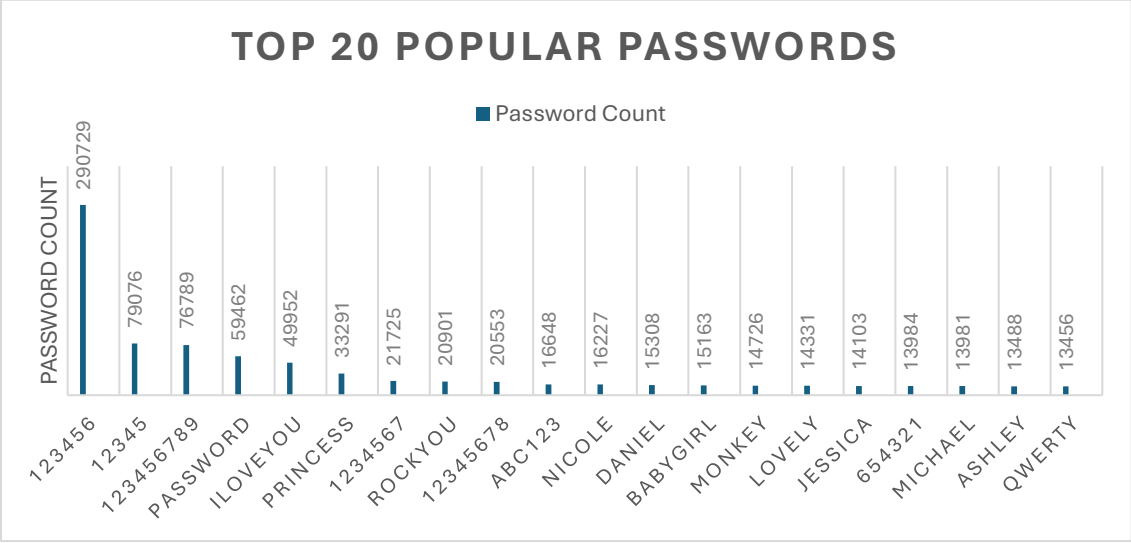

Figure 1. Top 20 password choice of users from RockYou breach [9].

- The top 10 passwords accounted for over 2% of the entire dataset, a strong indicator of poor security behavior.

- Around 65% of all passwords was only 6–8 characters long which means 2 out of 3 passwords were easily susceptible to brute-force attacks, especially with modern GPUs.

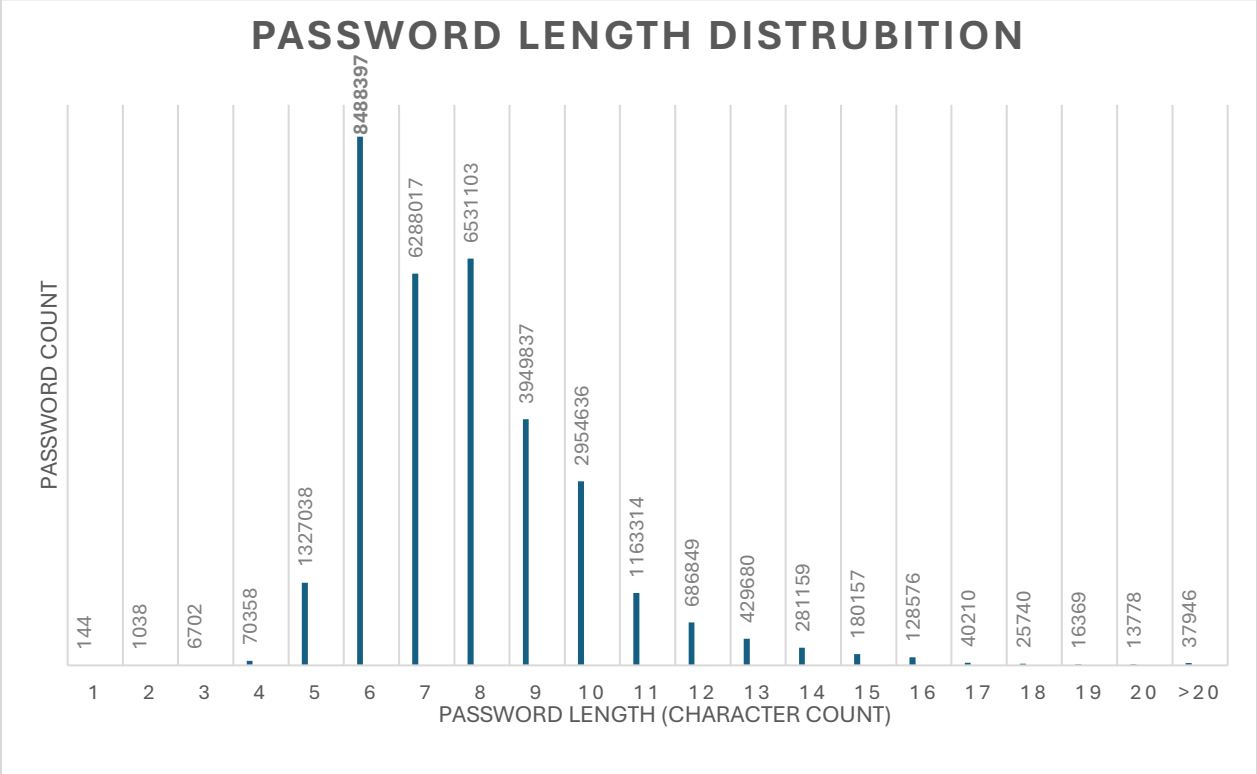

Figure 2. Distribution of password lengths from RockYou breach [9].

- Over 96.5% of users used only one or two types of characters (e.g., just lowercase letters or just digits).
- Only less than 3.5% use three or more-character types (e.g., lowercase + uppercase + symbols), which is vital for strong passwords.

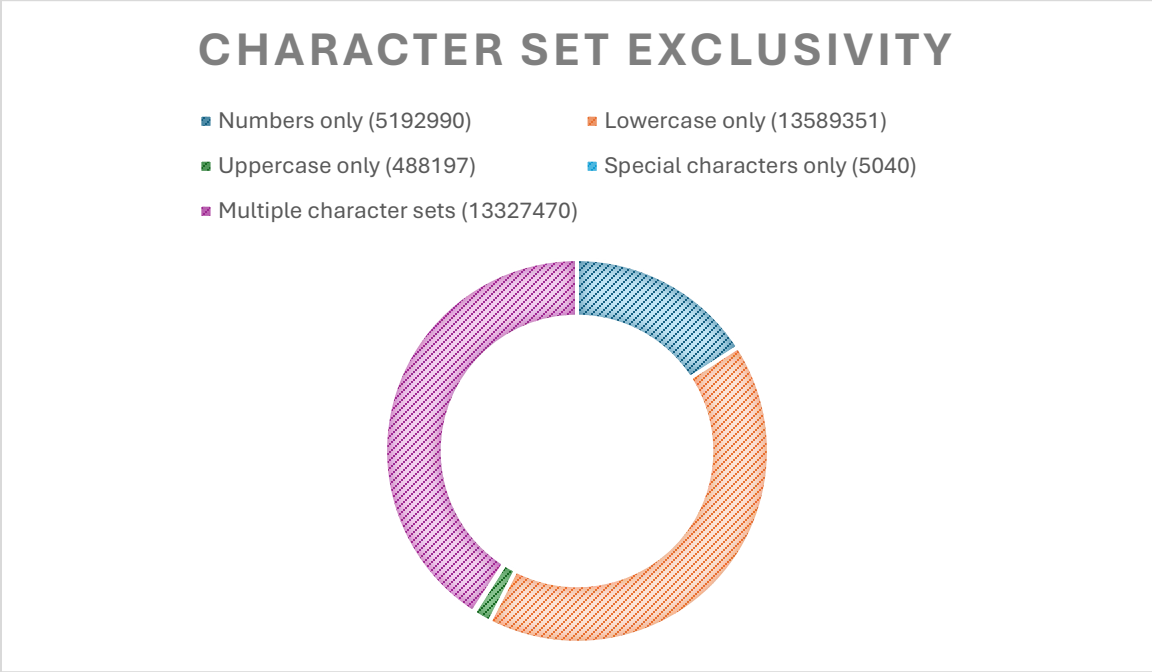

Figure 3. Character set exclusivity from all passwords of RockYou breach [9].

2. Online and Offline Attacks

Password-based systems are continuously targeted by adversaries employing a range of techniques to gain unauthorized access. These attacks can be conventionally categorized as online attacks and offline attacks, depending on whether the attacker interacts with the authentication interface in real time or operates independently on leaked data. Understanding both types is essential for designing effective defense strategies and reducing security risks.

2.1 Online Attacks

Online attacks are password-guessing attempts made by interacting directly with a live authentication service over the network typically through login forms, APIs, or mobile interfaces. Since the attacker is usually restricted by the system's response time and protective mechanisms, such attacks differ fundamentally from offline attacks, which occur entirely outside the control of the server.

Well-known variants of online attacks include:

- Brute-force attacks: Trying every password against a single account.
- Password spraying: Trying common passwords (e.g., "123456") across different accounts to avoid triggering lockouts.
- Credential stuffing: Reusing previously leaked username-password pairs from unrelated breaches to gain access to other services.

While credential stuffing relies on data from past offline breaches, its execution is distinctly online, as it involves submitting login attempts directly to active systems. According to NIST Special Publication 800-63B [13], authentication systems "shall implement a rate-limiting mechanism" to defend against such online guessing attacks [13]. These include limiting the number of failed authentication attempts and delaying or locking accounts under repeated failure conditions.

These attacks can be highly effective, especially in systems lacking rate-limiting, multi-factor authentication, or breached credential monitoring. For instance, a study in 2018 [14]

conducted an analysis about password reuse. They used a dataset derived from 107 services that includes 28.8 million users and their 61.5 million passwords. They found that 38% of users reused passwords across different sites, and 21% modified existing passwords to create new ones. Moreover, their study revealed that more than 16 million password pairs could be cracked within just 10 attempts [14], which highlights the real-world effectiveness of credential stuffing attacks.

Beyond generic guessing, targeted online attacks represent a more personalized and underestimated threat. In 2016 Wang *et al*. [15] Introduced the *TarGuess* framework, which shows that an attacker with limited personal information (e.g., full name, birthday, or past password patterns) can guess a user's password with high success rates in as few as 100 attempts. Their findings indicate that up to 73% of accounts belonging to typical users and 32% against security aware users could be compromised under such targeted models.

On the contrary with common belief NIST 800-63B [13] actually discourages mandatory periodic resets without reason, as they often lead to weaker, predictable patterns (e.g., Password1, Password2) However they also emphasize that if there is evidence to any compromise, related administrators should force users to change their passwords and also they should implement compromised password checks in order to prevent users from setting passwords known to be breached.

To mitigate online attacks effectively, systems should employ multiple strategies:

- Enforce rate limiting and account lockouts after repeated failed attempts [13]
- Require multi-factor authentication (MFA), making stolen or guessed passwords alone insufficient [13].
- Implement anomaly detection and login risk scoring (e.g., IP geolocation, device fingerprinting) [13].

Although online attacks are slower than offline attacks due to server-side controls, their success relies on user behaviors such as password reuse, weak password selection, and the inclusion of personal information in password construction. Without sufficient mitigation, these attacks can lead to large scale account compromises, even without any leaked password databases.

2.2 Offline Attacks

Offline attacks represent one of the most critical threats to password security due to their speed, efficiency, and undetectability. In contrast to online attacks where an adversary is constrained by server-side rate limiting or lockout policies, offline attacks occur once an attacker obtains access to a password database, typically through data breaches. With complete control over the hash values, attackers can perform unlimited password guesses locally without fear of detection or rate limitation.

Three main strategies dominate offline attacks:

- **Brute-Force Attacks:** Also known as the exhaustive search attack which means trying every possible combination of characters, which becomes computationally intensive for strong passwords or secure hash functions [16].
- **Dictionary Attacks:** Leveraging precompiled lists of common or leaked passwords, which drastically reduces computation time for weak or reused passwords [16].
- **Rainbow Table Attacks:** Rainbow tables are precomputed lookup optimization tables that exploit time-memory trade-off. They allow faster password recovery compared to brute-force or dictionary attacks but still require significant precomputation. By comparing the hashes in the stolen database against these tables, attackers can check reverse hashes without performing live computations. The computational burden is shifted to the precomputation stage, allowing rapid lookups at attack time [17].

It is important to understand that storing theoretically unbreakable passwords is impossible (the reason behind that will be explained later in this paper). The primary goal of a secure password storage system is to make attacks computationally infeasible for real world scenarios and with the help of secure cryptographic functions and proper additional measures like salting effectiveness for offline attacks can be significantly reduced. For instance, when a random and unique salt is added to each password before hashing, the output becomes different even for identical passwords. This destroys the feasibility of using precomputed tables, as attackers would need a separate rainbow table for every salt value. Unfortunately, studies [3] demonstrates that developers often neglected proper salting or used static salts, leaving systems vulnerable to such attacks.

Recent studies [18] show that using GPU-based and parallelized cracking methods (e.g., with Hashcat) drastically reduces cracking time. Empirical studies demonstrated up to 11.5× speedup in brute-force scenarios using NVIDIA GPUs, and dictionary attacks benefited from a 4.4× speedup when parallelized [18]. This makes it imperative that password storage techniques not only rely on hash complexity but also resist high-speed attacks through memory-hard designs (e.g., scrypt, Argon2).

Offline attacks, especially when combined with outdated hash functions, weak or reused salts, and modern hardware, can rapidly compromise millions of user credentials. Rainbow table attacks highlight the importance of implementing salting correctly preferably using secure, random, and unique values per password. When paired with strong, purpose-built password hashing algorithms or even better using memory hard Key Derivation Functions (KDFs) like scrypt or Argon2 can significantly mitigate the risks posed by powerful offline adversaries.

3. Hashing

Cryptographic hash functions are algorithms that process an input message of arbitrary length and produce a fixed-size output which is called a hash value [19]. Length of the output, also known as the message digest, varies depending on the hashing algorithm used. Common output sizes are 128 bits, 160 bits, and 256 bits. Cryptographic hash functions do not use any key and are designed to work only one way, which means that it should be easy to compute the result in one direction, but it should be extremely difficult to compute it in reverse direction [1]. The term "difficult" can be seen as ambiguous or non-formal when discussing cryptographic strength. To provide clarity, consider a hash function that produces an n-bit output referred to as n-bit hash function and assume it is secure. Under these assumptions, one can estimate the average number of hash function evaluations needed to break specific security properties as a function of n. In practice, cryptographic systems must be designed with the assumption that attackers will attempt to break them. Such attackers may engage in cryptanalysis, aiming to exploit weaknesses in the hash function. This typically involves trying to perform one of the standard attacks: finding a preimage, a second preimage, or a collision. If it can be demonstrated that any of these tasks can be accomplished more efficiently on a given hash function H than would be expected on an ideal hash function, then H is considered as broken [20]

3.1 Security Properties of Hash Functions

Security of a cryptographic hash function can be evaluated with three main properties, which are:

1. Preimage resistance: Given a specific hash output, it should be extremely difficult to determine any original input that produces that hash.
2. Second preimage resistance: For a given input message, it should be computationally infeasible to find a different input that generates the identical hash value.
3. Collision resistance: It should be computationally infeasible to identify two distinct inputs that result in the same hash output.

1) Preimage Resistance

A preimage refers to any input that maps to a specific output when processed by a hash function. In the context of a preimage attack, it is generally assumed that there exists at least one such input corresponding to the given hash output. Usually, the attacker is presented with a hash value $x = H(m)$ where is a randomly selected message unknown to the attacker [20]. Visual representation of Preimage Resistance is shown in the Figure 4.

In other words, if there is a given value of (m)) and hash function of $f()$ it should be easy to compute $f(a)$ but with given $f()$ function and result of $f(m)$ it should be extremely hard to compute the value of m [21].

In terms of password storage security preimage resistance is the most critical security measure and the only measure directly linked to this topic in practical applications.

One of the most common and fundamental strategies for attempting to find a preimage is brute force attack, which involves hashing a large number of randomly chosen messages until the desired hash value is encountered. For a hash function producing outputs of length n bits, this method typically requires 2^n trials. Of course, an attacker can choose to save the results of all previous hash attempts to effectively perform a precomputation. But this approach will have a time complexity of 2^n and requires storing roughly 2^n hash message pairs. Once this is done, the attacker can retrieve a matching input for any hash in constant time. However, that is not the complexity we are concerned with in terms of preimage resistance because if computing and storing 2^n values are already impractical, this strategy offers no advantage in practice and remains as a theoretical attack. But since computing capabilities continue to advance, the key takeaway is that such attacks demonstrate the need for 2^n operations to remain infeasible not only today but for at least the foreseeable future [20].

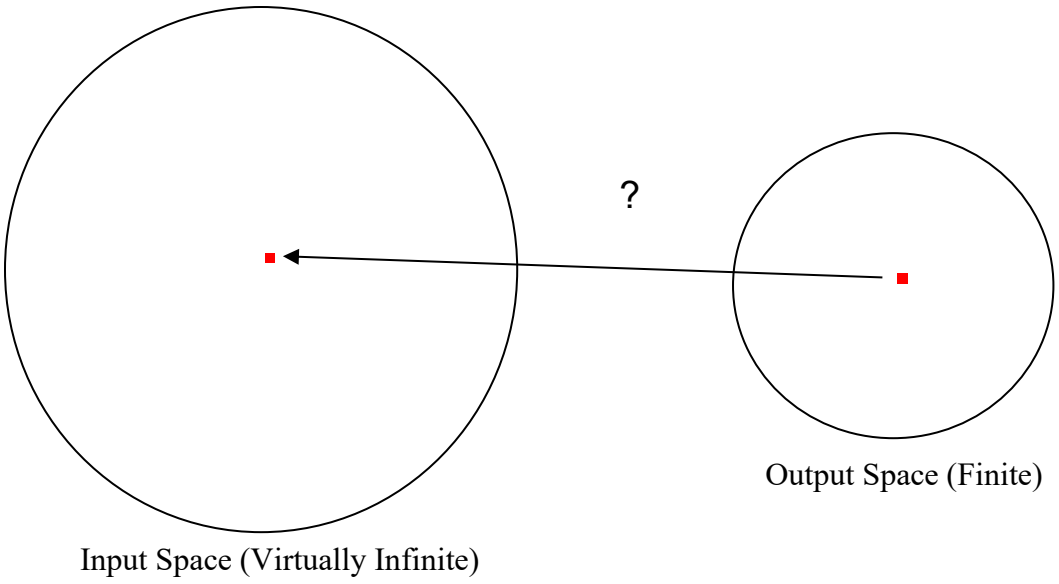

Figure 4. Visual representation of preimage resistance attack.

2) Second Preimage Resistance

A second preimage refers to a message that is different from a given input (unlike the first preimage) but produces the same output with the same hash function [20]. A visual representation of Second Preimage Resistance is shown in Figure 5.

In a typical second preimage attack scenario, the attacker is provided with a fixed message and its corresponding hash value, and the goal is to find a distinct input that results in the same hash. A second preimage attack can be conducted in a manner like a preimage attack, by randomly selecting input messages, computing their hash values, and checking whether they match the target hash while taking care to avoid reusing the original message. Due to the fact that the input space of a hash function is significantly larger than its output space, the probability of unintentionally reselecting the original message is negligible. As a result, the complexity of a second preimage attack is generally considered to be no greater than a preimage attack [20]. To explain more clearly, we can assume that scenario; If we store passwords hash and an attacker can generate the same hash output with a different password then the attacker can bypass the authentication system. However, since passwords usually have a significantly smaller character space than required input length to find a second preimage, it is practically infeasible in terms of password storage security.

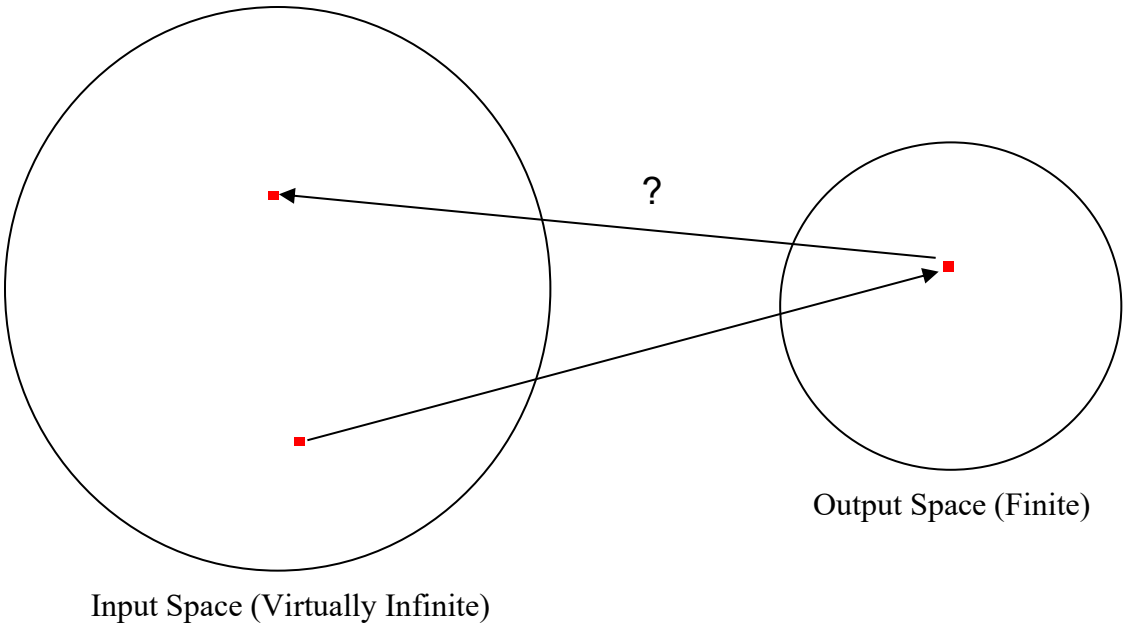

Figure 5. Visual representation of second preimage resistance attack.

3) Collision Resistance

A collision happens when two different input messages produce the same hash output. Since the goal is simply to find any such pair, no specific input is required to begin a collision search; we just need to know the hash function [20]. Visual representation of Collision Resistance property is shown in Figure 6.

A basic method to find a collision involves: Randomly selecting messages, hashing each one and checking whether any two produce the same hash. It might seem similar to second preimage resistance but now we are looking for any two items that create the same hash output rather than a match for a specific input.

To find a collision in any given hash function we can simply calculate the hash of a random message, check if it has been seen before, if not continue the same process.

With m messages the number of pairs is $\binom{q}{2} = q(q-1)/2 \approx q^2/2$ Since two random n bit strings have 2^{-n} probability of being equal for an n bit hash function 2^n pairs are needed for an expected collision. Therefore, when $q \approx 2^{n+1/2}$ (for acceptability large n) a collision is

expected. This is commonly simplified to a collision complexity of $2^{n/2}$. At this point, the likelihood of finding a collision is approximately $1 - e^{-1/2}$ which is around 0.39. The possibility increases with few more queries (with $m = 2^{(n+1)/2}$ queries the probability is $1 - 1/e \approx 0.63$) This type of attack is known as the birthday attack, named after the well-known birthday paradox in probability theory. Which is actually not a paradox, but the term paradox refers to the surprising fact that in a group of just 23 individuals, the likelihood of at least two people sharing the same birthday exceeds 50%. Similarly, in hash functions the birthday attack takes advantage of the fact that collisions can occur more easily than one might assume. This vulnerability applies to any hash function that significantly reduces input size. As a result, we can consider that for any n-bit hash function, the strongest achievable level of collision resistance is approximately $2^{n/2}$ [20].

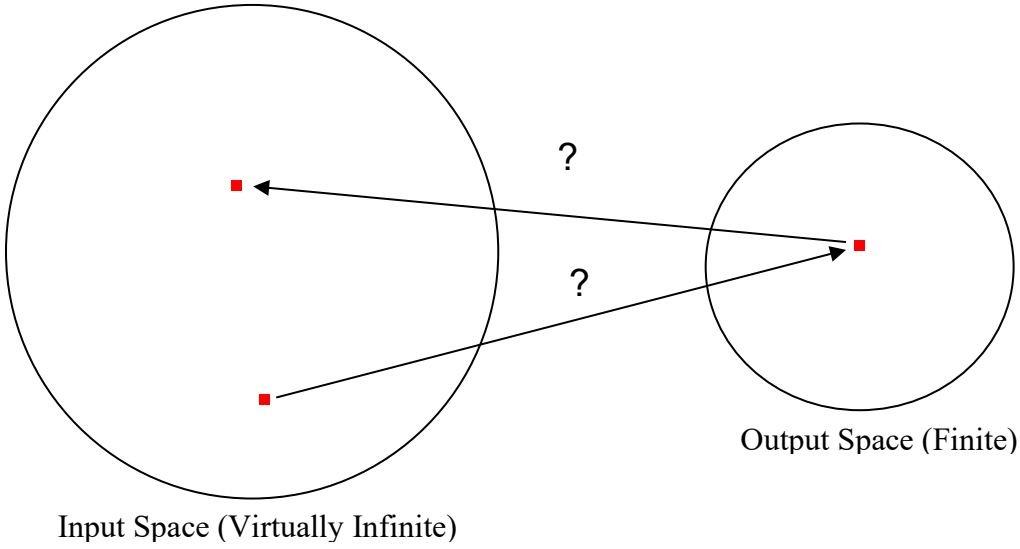

Figure 6. Visual representation of collision attack.

3.2 Hash Iterations

While choosing the right hash function is crucial, it is also important to consider how that function is applied. In order to provide secure password storage, one of the most effective techniques is to iterate the hash function multiple times. Iterations increase the computational workload required to verify each password guess which causes significantly hardening password cracking attempts, particularly in offline scenarios [13].

For instance, assume a hash function with C times iterations and t bit entropy. In this scenario iteration increases the effort of attacker from 2^t operations to $C * 2^t$ operations, which makes dictionary and brute force attacks more difficult for the attacker. However, the computational power required for a user who needs the authentication will also increase in this scenario. This creates a security performance tradeoff. Increasing iterations raises the cost for attackers on the other hand slows down legitimate users. The ideal iteration count should be the maximum value the system can support without compromising usability for related functions [22].

Crucially it is important to understand that the impact of iteration count is not symmetric, a slight delay (e.g., 500 milliseconds) is tolerable for a legitimate user logging in a system but becomes significantly computationally expensive for an attacker attempting millions or billions of guesses.

Legitimate User: Computes C iterations once per login (e.g., 100,000 iterations * 1 guess)

Attacker: Computes C iterations for every attempt (e.g., 100,000 iterations * 1 billion)

3.3 Commonly Used Hash Functions

3.3.1 Message Digest 5 (MD5)

MD5 is a cryptographic hash function developed by Ronald Rivest as a successor and improved version of his old algorithm MD4 in 1991 and standardized in RFC 1321 [23]. It produces 128-bit output from an arbitrary length input and was widely adopted due to its simplicity and speed. MD5 was initially used in different security applications such as integrity verification,

digital signatures and password storage. However today, it is no longer recommended for use due to its severe vulnerabilities [20]. A single round function of MD5 is shown in Figure 7.

The MD5 algorithm processes the input message in 512-bit blocks and transforms them through a series of four rounds, each consisting of 16 operations. The transformation relies on non-linear functions (F, G, H, I), bitwise operations, and modular addition to achieve diffusion and confusion [23].

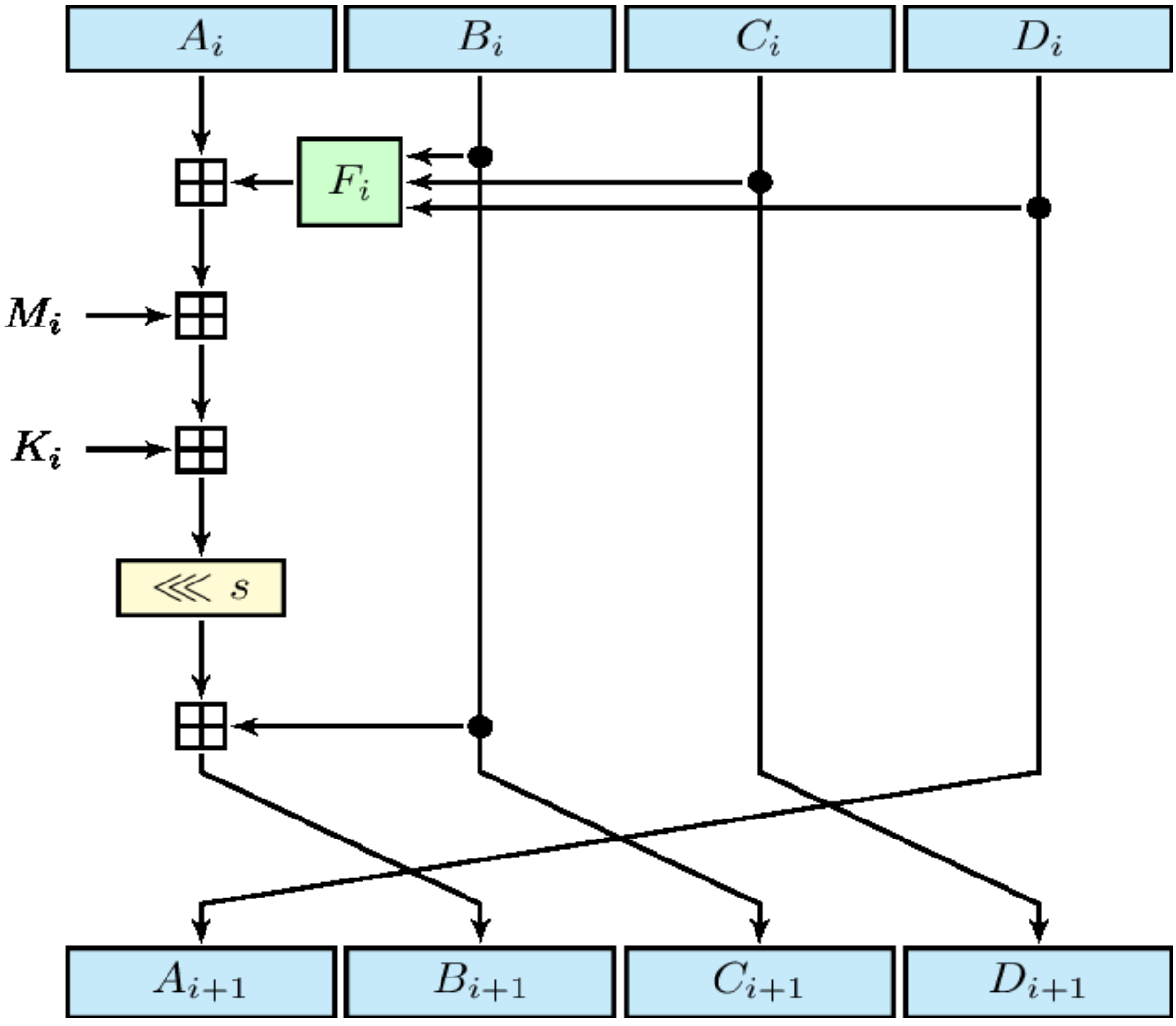

Figure 7. MD5 Round Function

The first practical collision attack against MD5, requiring only 2^{39} operations (significantly less than the 2^{64} theoretical bound for an ideal 128-bit hash function) presented in 2005 [24]. This

attack was significantly improved by using GPU Clusters to produce rogue CA certificate in 2009 [25].

Although MD5's collision resistance has been badly broken, allowing attackers to generate distinct inputs producing the same hash output [24], this vulnerability is not directly related to password security. In password hashing, the critical requirement is preimage resistance, the difficulty of reversing a hash to its original input. Moreover, while collision attacks directly affect cryptographic strength of some properties like digital signatures, they are practically non applicable in terms of password storage security.

As for preimage resistance, there is no known practical attack that has been published, but academically it is broken in numerous studies. For instance, in 2012 Aoki and Sasaki [26] developed an attack against full MD5 with a complexity of $2^{123.4}$, which is faster than exhaustive search 2^{128}. While this remains impractical with current hardware, advances in parallel computing and long-term trends like Moore's Law [27] raises concerns about this property's future feasibility.

One of the other concerns about MD5's security in context of password storage is its speed. Since it was designed for efficiency, modern hardware like GPUs and FPGAs can compute large number of MD5 hashes per second [18]. Which makes brute force and dictionary attacks significantly more effective, especially in scenarios where salting or iteration is not properly implemented. For instance, a study conducted in 2017 [28] shows that it is possible to search more than 250.000 hashes per second with a single core and search 18 billion hashes with full running the Chinese supercomputer Tianhe-1[14], which is 5.6 times faster than a CPU-only attack. Furthermore, MD5 does not support adjustable cost parameters or memory hardness, which are key features found in modern dedicated designs like Argon2[15].

Since it has several cryptographic weaknesses such as practically broken collision resistance, theoretically broken preimage security, and fast nature which makes it easily susceptible to being attacked, MD5 is no longer considered as a safe option for storing passwords. Moreover, since stronger and more secure alternatives like scrypt and Argon2 are available today

[14] https://en.wikipedia.org/wiki/Tianhe-1
[15] https://datatracker.ietf.org/doc/rfc9106/

(will be discussed in the following sections), there is no valid reason to continue using MD5 for password storage.

3.3.2 Secure Hash Algorithm 1 (SHA-1)

SHA-1 (Secure Hash Algorithm 1) is a Merkle Damgard type cryptographic hash function created by the U.S. National Security Agency in 1993 as SHA-0 and standardized as SHA1 in 1995. It produces a 160-bit output value from an input of arbitrary length and was commonly used for security applications like digital signatures, data integrity verification, and password storage. SHA-1 processes input messages in 512-bit blocks through 80 rounds of operations involving bitwise logical functions, modular additions, and rotations[16]. Single round of SHA-1 is shown in the Figure 8.

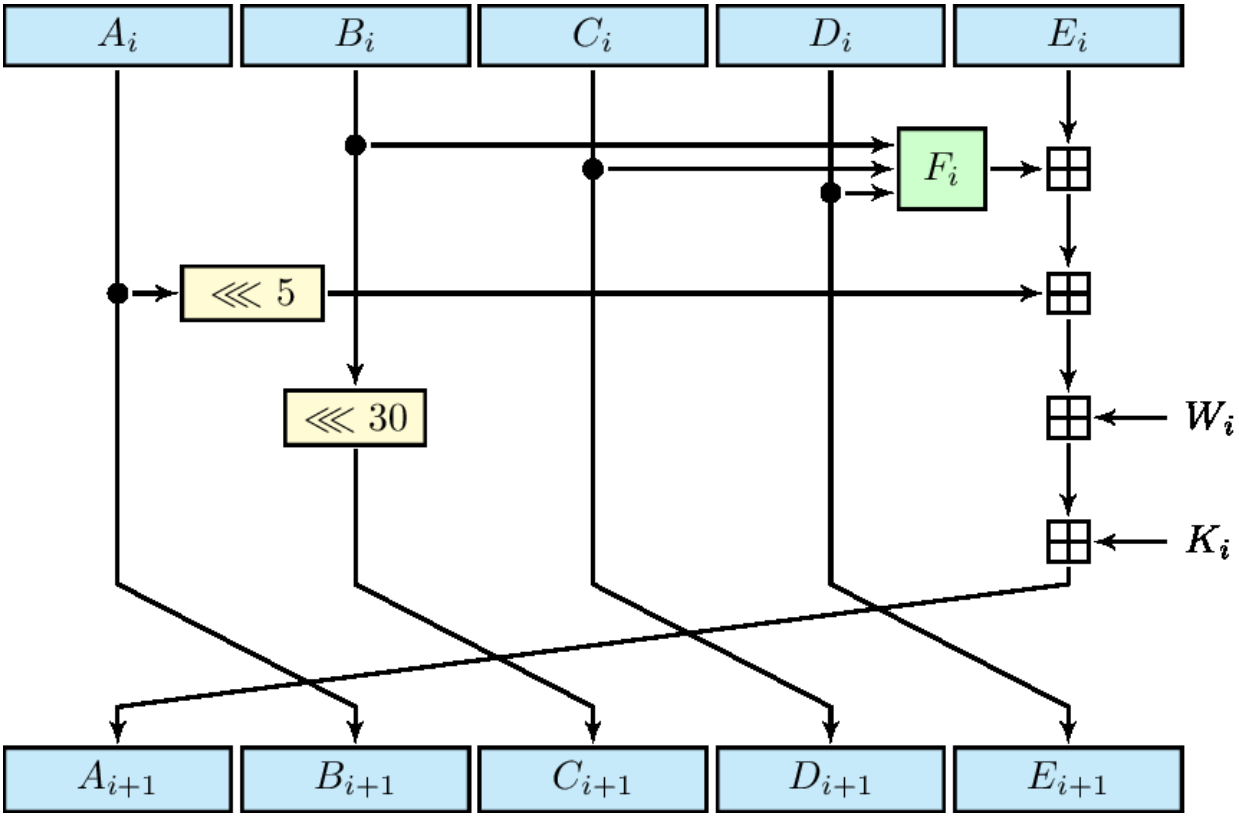

Figure 8. One iteration of the SHA-1 round function.

[16] https://en.wikipedia.org/wiki/SHA-1

- A, B, C, D, and E represent 32-bit intermediate variables that hold portions of the hash state.
- F is a nonlinear function that changes depending on the round.
- $\lll_n$ indicates a circular bitwise left rotation by *n* positions.
- *n* denotes the number of positions by which bits are rotated and varies with each round.
- W_i refers to the message schedule word used in round *i*, derived from the original message block.
- K_i is the constant value used during round *i*.

Despite its widespread use, SHA-1's collision resistance has been significantly compromised. In 2005, a collision attack with a computational complexity of less than 2^{69} operations, substantially lower than the ideal 2^{80} theoretical bound for a 160-bit hash function demonstrated [29]. This vulnerability was further exploited in 2017 when researchers from CWI Amsterdam and Google successfully generated a practical collision, producing two distinct PDF files with the same SHA-1 hash a demonstration known as the “SHAttered” attack [30]. Due to these cryptographic weaknesses, major browser vendors including Microsoft, Google, and Mozilla ceased support for SHA-1-based SSL certificates by 2017, reinforcing its deprecation in practice for digital certificates.

Although SHA-1’s preimage resistance has not yet been practically broken, studies on reduced-round versions [26] that indicate potential future vulnerabilities. This result raises long term security concerns for SHA-1 in password-related use.

Another critical weakness of SHA-1, much like MD5, is its computational efficiency. It is designed with prioritizing speed, which benefits adversary in an attack scenario. Tools like Hashcat or custom-built GPU clusters can calculate vast numbers of SHA-1 hashes per second, making brute-force and dictionary attacks highly effective if proper additional protective measures (like salting or iteration) are not enforced [18].

Given its practical vulnerability to collision attacks, its high-speed design, and its lack of tunable defense parameters, SHA-1 is no longer suitable for password hashing. Similar to the MD5, modern alternatives like scyrpt and Argon2 should be used instead.

3.3.3 Secure Hash Algorithm 2 (SHA-2)

SHA-2 (Secure Hash Algorithm 2) is a family of cryptographic hash functions developed by the U.S. National Security Agency (NSA) and standardized by the National Institute of Standards and Technology (NIST) in 2001. It was designed to address the vulnerabilities found in its predecessor, SHA-1, and to provide enhanced security[17]. The SHA-2 family includes several variants with different output sizes: SHA-224, SHA-256, SHA-384, SHA-512, SHA-512/224, and SHA-512/256 these variants accommodate different application requirements in terms of both performance and security. All SHA-2 variants are based on the Merkle–Damgård construction and operate on 512-bit (for SHA-256 and below) or 1024-bit (for SHA-512 and its derivatives) message blocks. The compression function applies to multiple rounds of logical operations, modular additions, and bitwise rotations [31]. For example, SHA-256 uses 64 rounds, while SHA-512 uses 80. Each round involves unique constants and message schedule words derived from the original message block, promoting randomness and resistance [19]. Single round of SHA-2 hash function is shown in the Figure 9.

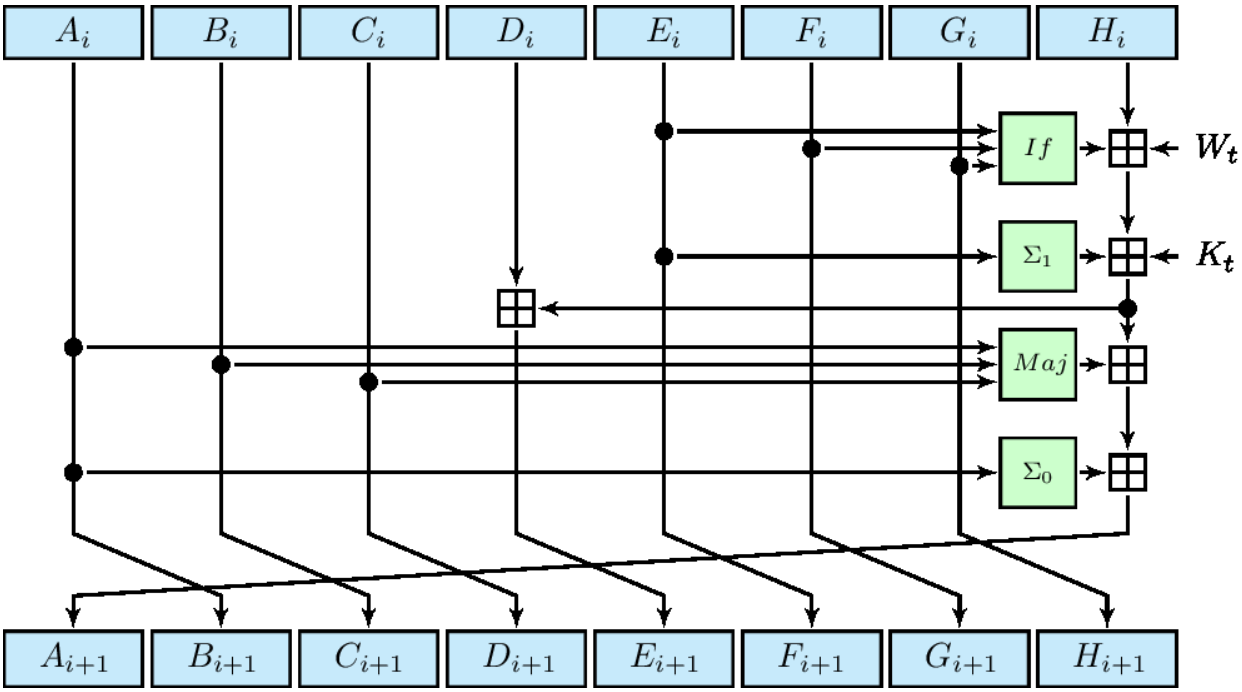

Figure 9. Sha 2 round function

[17] https://en.wikipedia.org/wiki/SHA-2

A brief comparison of SHA variants is shown in Table 2.

Table 2. Secure Hash Algorithm properties

Algorithm	Message Size (bits)	Block Size (bits)	Word Size (bits)	Message Digest Size (bits)
SHA-1	**$< 2^{64}$**	**512**	**32**	**160**
SHA-224	**$< 2^{64}$**	**512**	**32**	**224**
SHA-256	**$< 2^{64}$**	**512**	**32**	**256**
SHA 384	**$< 2^{128}$**	**1024**	**64**	**384**
SHA-512	**$< 2^{128}$**	**1024**	**64**	**512**
SHA-512/224	**$< 2^{128}$**	**1024**	**64**	**224**
SHA-512/256	**$< 2^{128}$**	**1024**	**64**	**256**

Note: Word size refers to the bit-length of the internal variables (e.g., A, B, C, ..., H) and registers used during the hash function's compression operations. It determines the size of the data units the algorithm processes per round and aligns with the underlying architecture used in CPUs or GPUs.

To date, full round SHA-2 has withstood practical cryptanalysis and remains secure against practical collision, preimage, and second preimage attacks. However, reduced-round versions have been analyzed, and theoretical attacks proposed, especially against SHA-256 variants [32] though none are yet practically exploitable. Given these characteristics, SHA-2 continues to be widely used in digital certificates, blockchain technologies, file integrity checks, and general-purpose cryptographic applications.

However, SHA-2, like its predecessors, is not designed for password hashing. It was designed to be computationally efficient and secure for general-purpose applications, such as digital signatures and data integrity verification.

According to performance analyses, SHA-256 is approximately 15.5% slower than SHA-1 for small strings and up to 23.4% slower for longer strings, due to its more complex structure and longer output size [33]. Despite this slight reduction in speed, SHA-256 remains a fast hash function making it inadequate for password storage where intentional computational cost is essential to slow down offline attacks. Without added defenses like salting, attackers can exploit SHA-2's efficiency to execute rapid dictionary or rainbow table attacks, particularly against weak or reused passwords. Moreover, just like MD5 or SHA1, SHA-2 lacks tunable parameters such as iteration count or memory hardness, further limiting its resistance to modern cracking techniques. These shortcomings have led to the development of dedicated password hashing algorithms like bcrypt, scrypt, and Argon2.

3.4 Dedicated Designs

While traditional hash functions like MD5, SHA-1, and SHA-2 have historically been used for password storage, their design prioritizes speed and efficiency qualities which makes it easier for attackers to make an exhaustive search when the database is compromised, and they get access to hashed databases. As a result, dedicated password hashing functions have been developed which are also known as key derivation functions (KDFs). They intentionally introduce features like key stretching and memory hardness to resist modern password cracking techniques. Among these, bcrypt, scrypt, and Argon2 have emerged as the most widely adopted and recommended options

for secure password storage. These algorithms offer tunable parameters such as iteration count, memory usage, and parallelism, enabling developers to adjust their resistance against evolving hardware capabilities. The following sections explore each of these algorithms in detail, evaluating their design principles, strengths, and limitations in practical use.

3.4.1 Bcrypt

Bcrypt is a password hashing algorithm introduced by Niels Provos and David Mazières in 1999, designed to improve the resilience of password storage systems beyond what standard hash functions could offer. Unlike general-purpose hash functions like MD5 or SHA-1, which are optimized for speed, bcrypt is intentionally slow and incorporates a work factor to make brute-force attacks computationally expensive [34].

Bcrypt extends the Blowfish Cypher [35] which is a Feistel type block cypher, through its EksBlowfish variant (Expensive Key Schedule Blowfish), which increases the computational demands by iteratively processing the password and salt through a configurable number of key derivation rounds and uses it in ECB mode. Single round of Blowfish Cypher and the full architecture of Blowfish Cypher are shown in the Figure 10. and Figure 11.

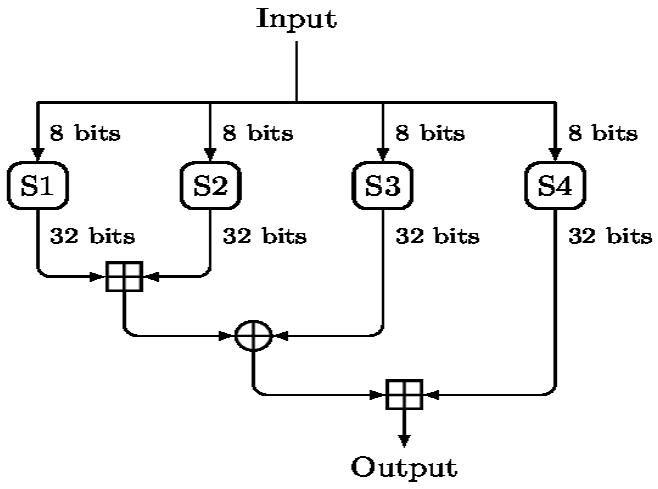

Figure 10. Blowfish round function

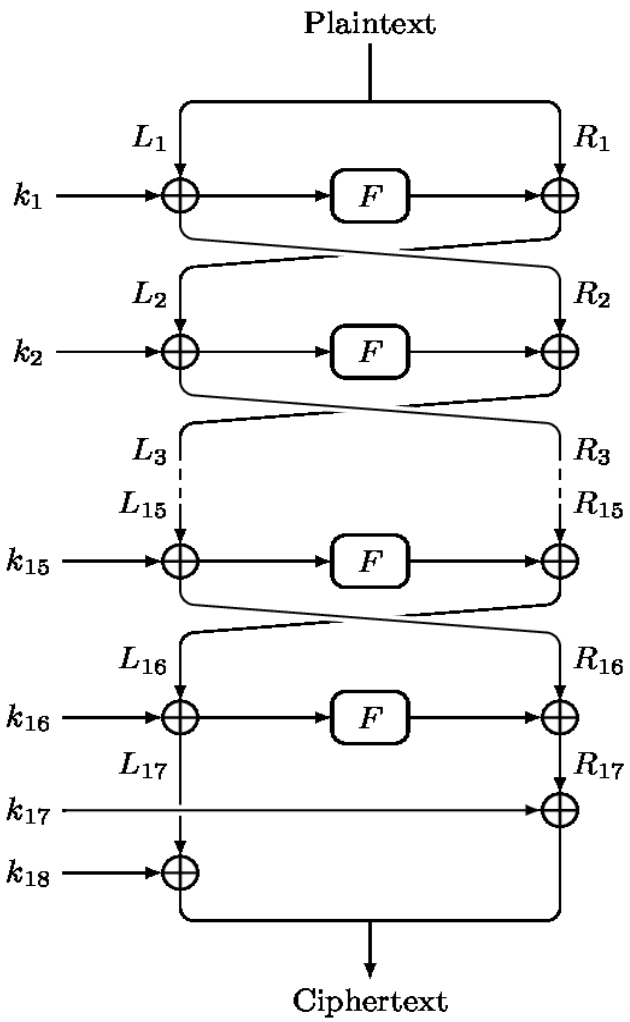

Figure 11. Blowfish Cypher

Unlike standard Blowfish which optimized for efficient encryption EksBlowfish performs 2^{cost} iterations during key setup, where each increment to the cost parameter (typically 10–14) doubles the required computation[18]. For example:

- At cost=10, hashing takes ~65ms on modern hardware.
- At cost=12 (4,096 iterations), it requires ~250ms.
- At cost=14 (16,384 iterations), it exceeds ~1,015ms

[18] https://auth0.com/blog/hashing-in-action-understanding-bcrypt/

When a user creates or updates a password, bcrypt executes the following workflow:

1) Input Preparation:
- Accepts the user's plaintext password (e.g., "Password123")
- Generates a 128-bit random salt (e.g., "9b13ppX8")

2) Key Initialization
- Initializes the EksBlowfish cypher using hexadecimal digits of π as starting point.

3) Expensive Key Derivation
- Concatenates the salt into password and creates a single input: "Password1239b13ppX8 "
- Processes this input through 2^{cost} iterations of the EksBlowfish key schedule:
- At cost=12, this requires 4,096 sequential iterations.
- Each iteration modifies the cipher's internal state.
- Outputs a secret state *K* (4,168 bytes)

4) Fixed Text Encryption
- Encrypts the 24-byte string "OrpheanBeholderScryDoubt" 64 times consecutively using *K*.
- The final ciphertext becomes the password "hash".

5) Storage
- Stores the result as: $2a$12$9b13ppX8$F5sH...
- where: 2a = algorithm version 12 = cost factor 9b13ppX8 = salt F5sH... = derived hash

The whole bcrypt algorithm is shown in Figure 12.

Figure 12. Visual representation of bcrypt algorithm.

Bcrypt mitigates key attack vectors with different interdependent security mechanisms: For instance, its adaptive work factor (2^{cost} rounds) which makes each password hash more time-consuming to compute. This significantly slows down brute-force attempts, making large-scale attacks too costly. Also, it generates a unique 128-bit salt for each password. This guarantees that even if different users have the same password, their stored hashes will look completely different, due to that property precomputed attacks like rainbow tables become ineffective by requiring attackers to search through 2^{128} possibilities [5].

Bcrypt remains one of the most widely used password hashing algorithm even after 27 years of its introduction. Niels Provos, one of the creators of bcrypt, stated this situation in his recent publishing as: bcrypt's continued relevance is supported not only by its cryptographic design but also by its practical advantages. Its inclusion in numerous open-source libraries has enabled broad integration across platforms and programming languages. As noted by Wikipedia, bcrypt is available in C, C++, C#, Delphi, Elixir, Go, Java, JavaScript, Perl, PHP, Python, and Ruby. This cross-language support has contributed to its widespread use. Additionally, bcrypt's emphasis on adjustable computational cost makes it appealing for large-scale web services, particularly when compared to newer algorithms like Argon2[19].

Despite its proven resilience, the evolution of computational capabilities and attack vectors over the past two decades has led to updated security requirements for using bcrypt safely. Modern guidelines emphasize specific configurations to ensure its continued effectiveness:

- The OWASP Password Storage Cheat Sheet [5] recommends:
- Enforce 72-byte maximum length at input validation, or Pre-hash longer passwords with a secure hash algorithm like SHA-512
- Depending on the performance abilities of the authentication mechanism. "Use bcrypt with work factor (cost parameter) ideally >12 or least 10.
- The NIST Guidance [13] advises using password hashing algorithms that are slow and resistant to brute-force attacks. Although bcrypt is still considered acceptable, memory-

[19] https://www.usenix.org/publications/loginonline/bcrypt-25-retrospective-password-security?trk=public_post_comment-text

hard functions like Argon2id are now preferred for new implementations due to their improved resistance to GPU and ASIC-based cracking attempts.

In summary, bcrypt continues to serve as a viable and secure password hashing function if it is implemented with an adequate cost factor (≥12), enforced input validation, and awareness of its limitations compared to more modern memory-hard algorithms.

3.4.2 Scrypt

Scrypt is a password-based key derivation function (PBKDF) developed by Colin Percival in 2009 to counter the weaknesses of traditional password hashing algorithms when exposed to parallel computation hardware like GPUs and ASICs. Its defining characteristic is memory hardness, the requirement for significant memory usage during computation making large-scale brute-force or dictionary attacks economically unfeasible [36] Standardized in RFC 7914 [37] by IETF (Internet Engineering Task Force). Scrypt is widely used in systems requiring elevated resistance to hardware-accelerated password cracking.

It took password, salt, N, r, p, and dkLen from user as input parameters. Input parameters and the purpose of these parameters are shown in Table 3.

Table 3. Input parameters and their purpose in scrypt.

Parameter	Purpose
Password	User Secret
Salt	Uniqueness
N	CPU/Memory cost
R	Block size
P	Parallelization
dkLen	Output length

1) Initial Setup - Constructing HMAC with SHA-256

Scrypt begins with the construction of an HMAC object using SHA-256 but crucially, no initial key or data is inserted yet.

The HMAC object gets created, without a key or data → HMAC (null, null)

A key is hashed from the input (user password) and the HMAC object initialized with this key → HMAC (h(password), null)

The data is still undefined but will be used in PBKDF2 steps of the algorithm.

2) First PBKDF2 (with HMAC-SHA256)

PBKDF2 is applied using the password (P), salt (S), and iteration count to derive a large array B, typically p × 128 × r bytes in size.

This is done using PBKDF2-HMAC-SHA256, meaning the HMAC primitive inside PBKDF2 uses SHA-256.

Each parallel instance (p total) creates a separate block B_i *(B_i refers to each intermediate block of key material generated during the first PBKDF2-HMAC-SHA256 step.*)

3) The SMix Function

Each B_i block is passed independently through the memory-intensive SMix function:

These are XORed with the current block, passed through BlockMix, which itself calls Salsa20/8. The quarter round function of Salsa20/8 is shown in Figure 13.

After all the B_i blocks are processed by SMix, they are concatenated together to form a new large buffer called B′ (B prime):

** This is the true "memory-hard" step designed to make parallelization very costly on GPU/ASIC due to the large memory footprint and data-dependent reads.*

**For SMix or BlockMix Salsa20/8 used as the only cryptographic primitive.*

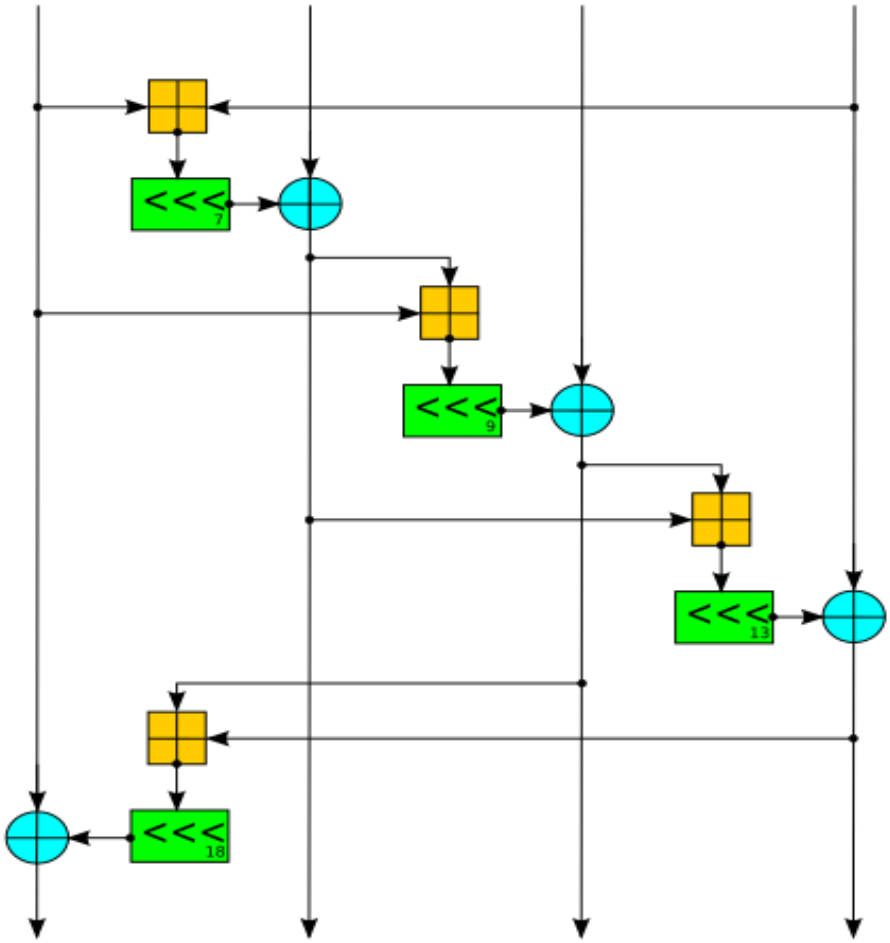

Figure 13. Salsa quarter round function four parallel copies make a round [20].

[20] By Sissssou - Own work, CC BY-SA 3.0, https://commons.wikimedia.org/w/index.php?curid=5805693

4) Final PBKDF2 (with HMAC-SHA256)

After processing all p blocks through SMix, the concatenated result B' undergoes a second application of PBKDF2-HMAC-SHA256 to derive the final output key of length dkLen which also is the stored final hash of the password.

Simple graphical representation of the whole scrypt construction is shown in Figure 14.

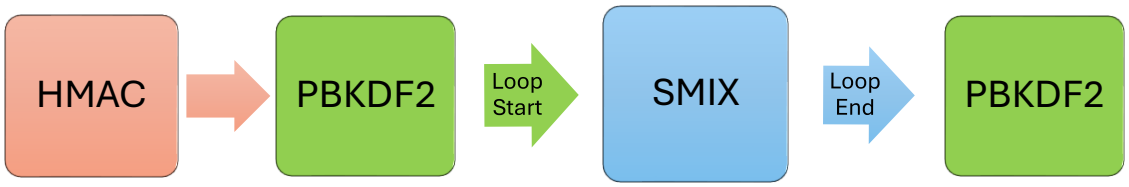

Figure 14 Simplified visual representation of scrypt

The entire process can be summarized as:

$$scrypt(P, S, N, r, p, dkLen) = MFcrypt_{HMAC\_SHA256, SMixr(P,S,N,r,p,dkLen)}$$

The notation MFcrypt (short for "Memory-Hard Function Crypt") denotes the core construction of the scrypt algorithm, combining two primary components: HMAC-SHA256 and $Smix^r$.

Scrypt's fundamental strength lies in its resistance to brute-force attacks leveraging GPU and ASIC hardware due to its high memory requirements and sequential access patterns. Compared to traditional KDFs like bcrypt or PBKDF2, scrypt notably:

- Memory Hardness: Requires significant RAM, thereby making custom ASIC or GPU attacks costlier.
- Configurable Parameters: Enables adjustment of the CPU cost (N), memory cost (r), and parallelization factor (p) to maintain resilience against evolving threats.

Its core innovation can be summarized as *"An algorithm that asymptotically uses almost as many memory locations as operations"* [36].

To emphasize the effectiveness of scrypt comparison of password cracking costs in different algorithms is shown in Table 4.

Table 4. Comparison of approximate password cracking cost from 2009 [36]

KDF	6 letters	8 letters	8 chars	10 chars	40-char text	80-char text
DES CRYPT	< $1	< $1	< $1	< $1	< $1	< $1
MD5	< $1	< $1	< $1	$1.1k	$1	$1.5T
MD5 CRYPT	< $1	< $1	$130	$1.1M	$1.4 k	$1.5 * 10^{15}
PBKDF(100 ms)	< $1	<$1	$18k	$160M	$200k	$2.7 * 10^{17}
bcrypt (95 ms)	< $1	$4	$130k	$1.2B	$1.5M	$48B
scrypt (64 ms)	< $1	$150	$4.8M	$43B	$52M	$6 * 10^{19}
PBKDF2 (5.0 s)	< $1	$29	$920k	$8.3B	$10M	$11 x 10^{18}
bcrypt (3.0 s)	< $1	$130	$4.3M	$39B	$47M	$1.5T
scrypt (3.8 s)	$900	$610k	$19B	$175T	$210B	$2.3 * 10^{23}

In summary, the scrypt algorithm represents a powerful advancement in password hashing and key derivation methods due to its strong resistance against hardware-accelerated brute-force attacks. Its memory-hard design effectively addresses vulnerabilities found in algorithms like MD5, SHA-1, and even bcrypt. Even though researchers[21] emphasizes that since it is not designed as a password solution storage it should not be the first choice, there are no fatal vulnerabilities known today and suggested by industry standard guidelines.

- The OWASP Password Storage Cheat Sheet [5] recommends scrypt as a viable option for password hashing.

"While Argon2id should be the best choice for password hashing, scrypt should be used when the former is not available."

- While NIST has not officially included scrypt in its standards, there have been public comments advocating for its inclusion. In the Public Comments on SP 800-132, [38] experts have suggested:

[21] https://blog.ircmaxell.com/2014/03/why-i-dont-recommend-scrypt

"We strongly suggest the addition of memory hard password hash functions to NIST SP 800-132 with mentioning Argon2 and potentially scrypt.

This indicates a recognition of scrypt's strengths in the cryptographic community, although it awaits formal adoption by NIST. But they also emphasized that since it includes various cryptographic primitives it is an overly complex construction, and this complexity increases the chances of vulnerable deployment or implementation.

Nevertheless, the complexity inherent to its design necessitates careful implementation and parameter selection. When applied thoughtfully (minimum CPU/memory cost parameter of (2^{17}), a minimum block size of 8 (1024 bytes), and a parallelization parameter of 1) [5], scrypt significantly improves the security of password storage systems, especially used with used with other password storage best practices and standards.

3.4.3 Argon2

Argon2 is a state-of-the-art password hashing algorithm designed to provide security against various attack vectors, including brute-force and side-channel attacks [39]. Developed by Biryukov, Dinu, and Khovratovich. Argon2 won the Password Hashing Competition in 2015 [22], highlighting its superior security features and performance [40].

Argon2 is a memory-intensive key derivation function designed with streamlined architecture. Its primary objectives include maximizing memory utilization efficiency and supporting parallel computation across multiple cores. The algorithm is specifically optimized for x86-based systems, leveraging the cache hierarchy and memory structure of modern Intel and AMD processors. While Argon2 includes three variants, the main recommended version is Argon2id, with Argon2d and Argon2i serving as complementary alternatives tailored for specific threat models [41].

[22] https://www.password-hashing.net/

Variants of Argon2:

Argon2 offers three distinct variants, each tailored to address specific security concerns:

- **Argon2id**: Combines the features of both Argon2d and Argon2i, offering a balanced defense against both side-channel attacks and GPU-based attacks. It is the recommended variant for most applications [41].
- **Argon2d**: Utilizes data-dependent memory access, making it highly resistant to GPU-based attacks (time memory trade off attacks). However, this variant is more susceptible to side-channel attacks, such as cache-timing attacks [41].
- **Argon2i:** Uses memory access patterns that are data-independent, providing enhanced protection against side-channel attacks. This variant is particularly suitable for environments where such attacks are a significant concern [41].

Argon2 employs a modified version of the BLAKE2 BLAKE2b [42] cryptographic hash function as its underlying primitive for both initialization and internal compression. BLAKE2b was chosen for its speed (which is faster than SHA-1 SHA-2 and MD5[23]), security, and efficient implementation on 64-bit architectures.

Additionally, the compression function G used to generate and update memory blocks throughout Argon2 is derived from a reduced-round variant of BLAKE2b's internal permutation. This design ensures high diffusion and non-linearity between memory blocks, which is essential for achieving memory hardness and resistance to cryptanalytic attacks [41].

Argon2 has the following inputs:

- P: Password (string)
- S: Salt (recommended: 128 bits)
- p: Degree of parallelism (number of lanes)

[23] https://en.wikipedia.org/wiki/BLAKE_(hash_function)

- t: Number of passes (iterations over memory)
- m: Memory size (in kibibytes)
- T: Desired output tag length
- v: Version number
- y: Type (0 = Argon2d, 1 = Argon2i, 2 = Argon2id)

Optional inputs include a secret key (K) and associated data (X), both of which must have a length not greater than (2^{32} -1) bytes.

The Argon2 function follows these major steps (RFC 9106, §3.2):

1. Initial Hash Generation (H_0)
   An initial 64-byte hash value is generated using BLAKE2b function, incorporating all input parameters. This serves as the root of all further computations.

2. Memory Allocation
   Memory is divided into m′ blocks, where:

$$m' = 4 \times p \times \left\lfloor \frac{m}{4p} \right\rfloor$$

   Each block is 1024 bytes. The memory is organized in a 2D matrix of p rows (lanes) and $q = m'/p$ columns.

3. First Block Initialization
   Each lane is initialized using H′, the extended hash function based on H, as follows:

$$B[i][0] = H'^{1024}\big(H_0 \mid \mathrm{LE32}(0) \mid \mathrm{LE32}(i)\big)$$

4. Second Block in Each Lane
   The second block for each lane is generated similarly, using index 1 instead of 0.

5. Remaining Block Generation

   All remaining blocks $B[i][j]$ are computed using the compression function G, where the inputs are the previous block and another pseudorandomly selected block:

$$B[i][j] = G(B[i][j-1],\ B[i][z])$$

- *Role of the Compression Function G: The compression function G is central to Argon2's design. It is a permutation-based function derived from BLAKE2b and is responsible for mixing memory blocks securely. This function ensures diffusion and non-linearity between block inputs.*
- *The selection of z depends on whether Argon2d, Argon2i, or Argon2id is being used.*

6. Multiple Passes (t > 1)

   For more than one pass, all blocks are recomputed, and the output is XORed with the previous value.

7. Final Block Computation

   After t passes, the final block C is computed by XORing the last block of each lane:

$$C = B[0][q-1] \oplus B[1][q-1] \oplus \ldots \oplus B[p-1][q-1]$$

8. Tag Generation

   This XORed value C is passed into a final variable-length hash function (based on BLAKE2b again). The function produces the output hash (also called the “tag”) of desired length T (e.g., 256 bits).

While Argon2 is widely regarded as a secure memory-hard password hashing algorithm, recent research has highlighted important considerations regarding its real-world effectiveness. One of the primary concerns is the inconsistent adoption of secure parameter configurations, which can significantly reduce its theoretical advantages. A very recent study demonstrated through economic modeling that although Argon2 with 2048 MiB memory (as recommended in [41]) can reduce compromise rates by up to 46.99% compared to SHA-256, many software projects continue to implement weaker configurations, such as the 46 MiB setting suggested by OWASP, or even lower. Their large-scale analysis of GitHub repositories revealed that 46.6% of real-world Argon2

implementations use weaker-than-recommended settings, including sensitive applications such as password managers [43].

Importantly, this paper reaffirmed Argon2's strength against GPU and ASIC-accelerated attacks, especially when using Argon2id with RFC-recommended parameters. However, it also cautioned that default settings provided by libraries are often blindly adopted by developers, highlighting the need for better developer guidance, parameter selection tools, and integration of password strength estimators into secure systems [43].

Furthermore, Argon2's security guarantees rely heavily on password strength. Even under high-memory configurations, simulations using the RockYou dataset showed compromise rates exceeding 96.8% for weak passwords, regardless of the algorithm. This emphasizes that Argon2 cannot compensate for poor user behavior, such as choosing easily guessable passwords. The study also found diminishing returns in security as memory allocation increases suggesting that beyond a certain point, additional memory does not yield proportional improvements in resistance to cracking attacks [43].

Another class of attacks, known as time–memory trade-off (TMTO) attacks, aims to reduce memory usage by increasing computational time [40]. explored such trade-offs and showed that, while possible in theory, Argon2's design significantly limits the effectiveness of TMTO strategies when conservative parameter settings (e.g., high memory cost) are used. Importantly, no practical preimage or collision attacks have been demonstrated against Argon2's core compression function, derived from BLAKE2b. As RTF standardization document stated, Argon2id with sufficient memory (≥19 MiB), two iterations, and parallelism of at least one is considered resistant to all known practical attacks [41].

With its well-designed memory hard architecture, resilience to both classical and hardware-accelerated attacks, and flexibility through tunable parameters, Argon2 particularly Argon2id remains the state-of-the-art solution for secure password storage. Compared to legacy algorithms like bcrypt and scrypt, Argon2 provides superior trade-offs in terms of security, efficiency, and implementability. Whereas bcrypt is limited by its fixed memory usage and scrypt relies on multiple primitives and lacks data-independent operation modes, Argon2 leverages a streamlined design based on iterations of BLAKE2 and permits independent control over memory, time, and

parallelism parameters. This allows Argon2 to maintain high resistance against time–memory trade-off attacks and side-channel vulnerabilities when properly configured.

Moreover, as mentioned before Argon2 has received formal standardization through [41] and its adoption is increasingly promoted by security organizations such as [5]. Although challenges remain particularly in the secure selection of parameters and widespread awareness of developers, Argon2id with conservative settings (e.g., ≥19 MiB memory, ≥2 iterations) currently offers the most strong, adaptable, and future-proof defense for password hashing in modern applications.

A comparison for mentioned hash functions is shown in the Table 5.

Table 5. Comparison table for mentioned hash functions

Algorithm	Introduction Date	Output Length	Adjustable Parameters	Memory Hardness	Recommendation
MD-5	1991	128-bit	No	No	No
SHA-1	1993(1995)	160-bit	No	No	No
SHA-256	2001	256-bit	No	No	With added protections
bcrypt	1999	192-bit	Yes	No	With specific parameter choices
scrypt	2009	Adjustable	Yes	Yes	Yes
Argon2	2015	Adjustable	Yes	Yes	Yes

4. Additional Measures

Storing passwords securely means more than selecting a strong hashing algorithm. In practice, additional layers of defense are necessary to mitigate large-scale threats such as database leaks, targeted offline attacks, and credential reuse. Among these measures, two widely recognized techniques salting and peppering play a critical role in improving the uniqueness and secrecy of stored password hashes. While salting ensures that each password produces a distinct hash value, even if the same password is used by multiple users, peppering introduces a hidden element that complicates an attacker's ability to perform brute-force operations. Another complementary approach is the use of honeywords which are decoy passwords stored alongside real ones to detect unauthorized use of leaked credentials by triggering silent alerts during login attempts.

Together, these methods serve to reinforce the integrity of password storage systems and are recommended in both academic literature and practical security guidelines.

4.1 Salting

Salt is a randomly generated, user-specific value added to each password prior to hashing and stored with the resulted hash. An example of salt storage is shown in the Figure 15.

Due to uniqueness of each salt, attackers cannot use a single precomputed hash to check against multiple stored hashes; instead, they must compute hashes individually for each password-salt pair. This approach significantly increases the computational cost of cracking a large number of hashes, as the effort scales with the number of entries [44].

Salting also defends against rainbow table attacks and lookup-based methods by ensuring that the attacker cannot rely on precomputed hash databases. Additionally, it conceals whether two users have chosen the same password, since different salts will produce entirely different hash outputs even if the original passwords are identical. However, since salts are not secret values, they do not prevent security against dictionary attacks [44].

Modern password hashing algorithms such as Argon2id, bcrypt, and scrypt incorporate salting internally, due to that no additional salting required while using them [5].

The primary role of a salt is to enable the creation of a large number of unique keys derived from the same password when using a fixed number of iterations. Specifically, for any given password, the total number of possible derived keys is approximately 2^{Slen}, where *Slen* represents the length of the salt in bits. This substantial key space makes it highly impractical for an attacker to precompute a lookup table without any knowledge, even for a limited set of commonly used passwords. Moreover, to prevent a precomputed table each salt or at least a part of salt should be generated by an approved RNG (Random Bit Generator) and the length of this randomly generated portion or the whole salt of the salt should be at least 128 bits. This ensures not only security against precomputed tables but also prevents collisions across different databases. Beyond these recommendations it is also allowed to use an optional "purpose" string as a prefix to randomly generated salts, the final value should look like this:

$$S = \text{purpose} \,||\, \text{randomValue}$$

This approach aims to guarantee preventing any collision between two distinct databases and could be useful if system requires passwords hashed in different contexts [45]

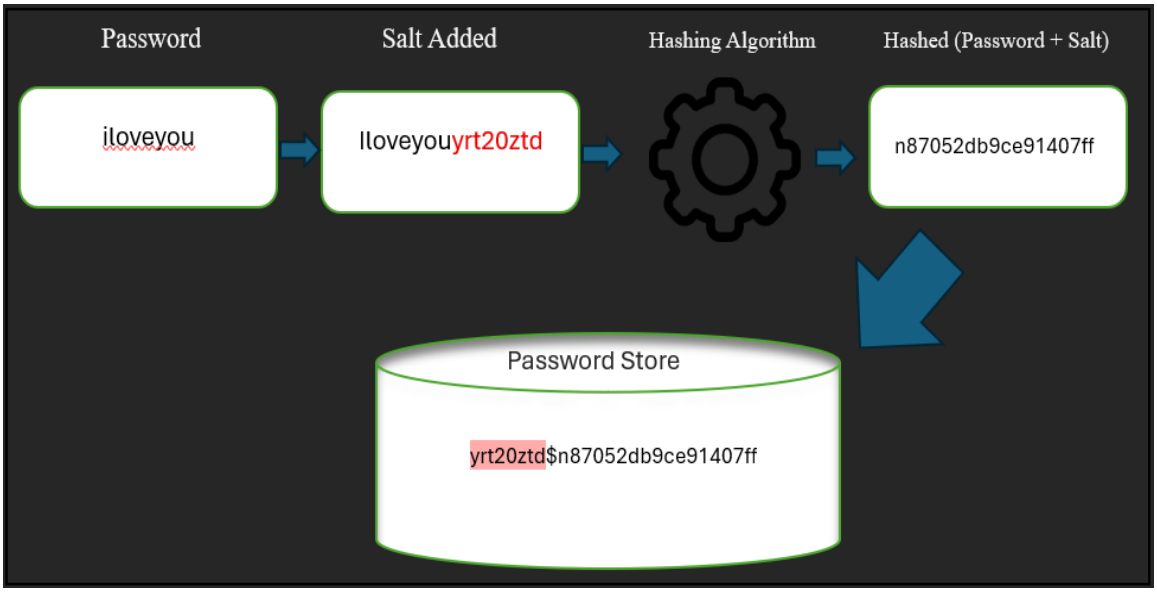

Figure 15 Example of salt storage.

Beyond the standard practice of using a single random salt per password, more advanced salting strategies have been proposed to further enhance security. One such approach is dynamic salting, where the salt can change or be re-generated over time or per usage. For example, recent research introduced the concept of generating a *safe one-time salt* for each authentication session [46] In this scheme, often called *salt-per-session*, a fresh salt value is generated every time a user initiates a login, and the authentication process incorporates this one-time salt. The aim of this approach (referred to as RGSCS, Random Generator of a Safe Cryptographic Salt per Session) is to evolve traditional password hashing into an enhanced security system. By using a new salt for each session (in addition to the static stored salt), an attacker who manages to steal a database of password hashes would find that those hashes are tied to specific session salts or are short-lived, thereby offering an additional layer of protection. Dynamic salting can limit the usefulness of any single hash value to an attacker, at the cost of additional complexity in the authentication protocol (often involving the server sending a fresh salt or challenge to the client for each login).

In summary, salting remains a foundational technique in secure password storage, providing strong protection against precomputed and bulk attacks when implemented correctly. While modern algorithms already integrate internal salting mechanisms, understanding its principles and the benefits of advanced methods helps build more resilient systems.

4.2 Peppering

Peppering, also known as secret salt, is a complementary security mechanism to salting that introduces an additional layer of defense in password storage architectures. While salt is a unique and non-secret value added to each password before hashing, a pepper is a secret, application-wide value that is also added to the password, typically before or after salting and hashing. The key distinction lies in its secrecy: unlike salts, which are stored in the same location with the hash in the database, the pepper is kept separate from that database, often in a secure configuration file, hardware security module (HSM), or environment variable[24].

Peppering strengthens resistance against offline dictionary attacks and rainbow table attacks by introducing an unknown variable attackers cannot access even if they fully compromise

[24] https://en.wikipedia.org/wiki/Pepper_(cryptography)

the user database. This means even if attackers obtain password hashes and salts, they still cannot verify guessed passwords without the pepper[25].

In practical implementations, the pepper can be:

- A static string appended to all passwords before hashing (e.g., H (password + pepper).
- A key used in an HMAC function, such as HMAC(pepper, password || salt) [5].
- Incorporated into more complex password hashing schemes like bcrypt or Argon2, depending on the API design.
- For example, while using SHA-256, a developer might hash a password like:

SHA-256(password || salt || pepper)

It is important to recognize that peppers serve as supplementary security measures and should not be used as a substitute for core practices such as employing a strong hash function and properly implemented salts. Since a pepper is typically a single, shared secret across the entire system, it introduces a single point of failure. If compromised, the pepper becomes ineffective; however, its exposure does not affect the overall security of the system. Therefore, peppers should be considered as an optional but beneficial layer of defense in a multi-tiered security architecture.

4.3 Honeywords

In traditional password storage models, the successful cracking of a hashed password remains silent and mostly undetectable until the attacker uses the credentials. Honeywords, introduced by Juels and Rivest in 2013 [47], offers a simple yet powerful solution to these models. The core idea involves generating and storing multiple plausible-looking passwords for each user, with only one being the legitimate password (the *sugarword*) and the rest serving as decoys (honeywords). When a honeyword is submitted during authentication, the system detects the unauthorized use, triggers an alarm and transforms password cracking from a silent event to a detectable event [47].

[25] https://en.wikipedia.org/wiki/Pepper_(cryptography)

Effective honeyword generation is rooted in several design criteria:

- Believability: Honeywords must closely resemble real passwords to deceive attackers.
- Diversity: They should include both weak and strong variants to reflect real-world password distributions.
- Separation from the true password: The true password must be hidden among honeywords in a way that an attacker cannot identify it with high probability.

To achieve this, one can start with a dictionary of common passwords, apply some set of rules such as character substitution, case variation, and number/symbol appending, and then randomly place the user's actual password among the generated decoys. For instance:

Real password: Password91

Honeywords: Passw0rd91, Passw0rd92, Password21, Password911, etc.

To apply honeywords the system split into two components [47];

- Main Authentication Server: Stores all *sweetword* hashes and receives user input.
- Honey checker: A hardened, isolated server that stores only the index of the correct password among the sweetwords for each user.

During login, if the submitted password hash matches any sweetword, its index is sent to the honey checker. If the index does not match the known correct position, an intrusion alarm is triggered. This distributed architecture ensures that even if the password database is breached, the attacker cannot distinguish the correct password without also compromising the honey checker.

Honeywords offer several notable advantages:

- Intrusion detection: Any use of a decoy reveals that the password file has likely been breached.
- Attack deterrence: Attackers risk exposure even if they crack the hashes.
- Detecting time of breaches: Periodic rotation of honeywords to detect the timing of breaches.

However, it should also be considered that honeywords can be weaponized as an attack vector for DoS (Denial of Service) attacks. An attacker who knows the true password might intentionally use a honeyword to lock accounts.

In summary honeywords represent a low-cost, backward-compatible enhancement to password security, especially suitable for detecting breaches in offline attack scenarios. But similar to peppers they are not a replacement for core measures like strong hash functions, as an additional layer of defence they only offer visibility into attack attempts that would otherwise remain undetected.

5. Other Issues and Future Directions

Effective password protection today depends on a broad set of design choices that extend beyond the core authentication mechanism. As digital systems grow more complex and cyber threats become more advanced consideration must also be given to the adoption of industry standards, the secure storage of credentials, and the implementation of additional defense mechanisms such as multi-factor authentication. In this section, key supporting practices and emerging trends will be examined to demonstrate how overall password security can be further improved.

5.1 Standards and Best Practices

Standardization provides the foundation for secure, interoperable, and long-term resilient password storage systems. It enables developers to follow well-established principles, avoid design inconsistencies, and reduce the risks associated with fragmented security practices. As Ross Anderson emphasizes in his book "Security Engineering", "*many secure distributed systems have incurred large costs, or developed serious vulnerabilities, because their designers ignored the basics of how to build (and how not to build) distributed systems*" [48].These mentioned "basics" often refer to shared technical standards that formalize consensus on secure design and implementation. By relying on such standards, systems can achieve strong protection mechanisms, remain resilient against evolving threats, and support verifiable compliance. Without them, security often becomes reactive, inconsistent, and vulnerable to misconfigurations.

By establishing consensus-driven requirements, standards: Reduces reliance on temporary solutions that introduce vulnerabilities and design flaws. Defines best practices by turning proven cryptographic research into actionable procedures. Helps systems to align with regulatory frameworks (e.g., GDPR, PCI-DSS, KVKK) and provides measurable security benchmarks.

Key industry standards can be listed as:

1) National Institute of Standards and Technology SP800-63B / FIPS 180-4

- SP 800-63B Digital Identity Guidelines - Authentication and Lifecycle Management [13]

This standard provides authoritative guidelines for federal agencies and private-sector systems managing digital identities, focusing on the secure handling of credentials. It defines best practices for secure authentication, including how credentials (like passwords) must be managed, stored, and protected.

Key recommendations from this guideline include:

- Use one-way key derivation functions (KDFs): Argon2id (preferred), PBKDF2, bcrypt, or scrypt.
- Apply unique, random salt (minimum 32 bits).
- Avoid deprecated hash functions like MD5 and SHA-1.
- Do not require periodic password changes unless there's evidence of compromise.

- Federal Information Processing Standards FIPS PUB 180-4: Secure Hash Standard (SHS) [49]

The Federal Information Processing Standards (FIPS) represent a collection of publicly available technical standards developed by the U.S. National Institute of Standards and Technology (NIST). FIPS publications establish critical benchmarks for cybersecurity and system interoperability, often adapting existing industry standards from organizations like the International Organization for Standardization (ISO[26]), American National Standards Institute (ANSI[27]) and Institute of Electrical and Electronics Engineers (IEEE[28]). FIPS 180-4 specifies approved cryptographic hash functions for federal information systems requiring secure data integrity verification. As a mandatory standard for all U.S. government civilian agencies and contractors, it establishes the foundational algorithms for digital signatures, message authentication, and other cryptographic applications.

[26] https://www.iso.org/home.html
[27] https://www.ansi.org/
[28] https://www.ieee.org/

2) Open Worldwide Application Security Project (OWASP) Cheat Sheet [5]

OWASP Authentication Cheat Sheet provides developers with practical, up-to-date guidance for implementing secure authentication in web applications. Its recommendations include:

- Use memory-hard functions like Argon2id (recommended), or bcrypt, PBKDF2, scrypt.
- Use pepper only with strong protection and never store it with hashes.
- Enforce password complexity and minimum length (typically ≥ 12 characters).
- Avoid storing password hints and never use plaintext storage.
- Implement rate-limiting and account lockout to prevent brute-force attacks.

3) International Standards Organization (ISO)/ International Electrotechnical Commission (IEC) 27001, 27002, and 29115

- ISO/IEC 27001: Information Security Management Systems – Requirements [50]
- ISO/IEC 27002: Code of Practice for Information Security Controls [51]
- ISO/IEC 29115: Entity Authentication Assurance [52]

ISO/IEC 27001, 27002, and 29115 establish internationally recognized frameworks for information security management, ensuring systematic protection of data confidentiality, integrity, and availability including secure credential storage and authentication. Key recommendations include:

- Enforce the least privilege and access controls on stored authentication data.
- Securely generate, store, and rotate salts, keys, and other cryptographic material
- Monitor and audit access to credential storage systems.
- Support multifactor authentication when feasible and identity assurance levels (especially in ISO 29115).

4) Internet Engineering Task Force RFCs - RFC 8018 / *RFC 9106*

- RFC 8018: PKCS #5 Password-Based Cryptography Specification Version 2.1 [53]
- RFC 9106: Argon2 memory-hard functions for password hashing and other applications [41]

The Internet Engineering Task Force publishes its technical documentation as RFCs, which is an acronym for their historical title Requests for Comments. They define the core technical infrastructure of the Internet, including mechanisms for addressing, routing, and data transport. In context of password storage management, they formally define Password-Based Key Derivation Functions (PBKDFs) and memory-hard functions like Argon2, which are designed to protect against brute-force and GPU-based attacks.

Two critical RFCs for password storage are:

- RFC 8018: The formal specification for PBKDF2, a widely used Password-Based Key Derivation Function, with guidelines for secure implementation (e.g., minimum iteration counts).
- RFC 9106: Standardizes Argon2, the memory-hard password hashing algorithm, including recommended parameters (time cost, memory size, parallelism) to resist GPU/ASIC attacks.

These RFCs provide algorithm definitions, security considerations, and implementation best practices to protect against brute-force and precomputation attacks.

Following well-known standards and best practices is not merely a recommendation, it is a necessity for building secure, scalable, and auditable authentication infrastructures. With guidelines from trusted organizations like NIST, ISO, OWASP, and the IETF are based on years of research and real-world experience developers can ensure that their systems are not only resilient to evolving threats but also compliant with legal and industry expectations These standards transforms decades of cryptographic research and operational expertise into actionable guidance, offering a reliable foundation upon which modern security architectures should be built.

5.2 Where to Store Passwords

While secure password storage practices such as using strong hash functions and salting are essential, the security of the environment where password hashes are stored is equally critical. Insecure storage environments such as misconfigured databases, exposed configuration files, or unencrypted backups can make even the strongest storage architectures ineffective.

Storing credentials in insecure locations remains one of the most common and critical mistakes in application development. Research shows that sensitive data such as hashed passwords and API keys is frequently stored in local configuration files or directly embedded in application source code. These secrets are sometimes unintentionally committed to version control systems like git, where they become accessible to collaborators or even the public if the repository is not properly secured. The risk is increased in shared development environments and automated deployment pipelines, where insecure files can easily spread across systems [54]

Misconfigurations in cloud storage systems present another major threat. In a forensic study of credential leaks, it was found that poorly configured storage environments such as publicly accessible cloud systems or overly permissive access logs enabled attackers to retrieve hashed passwords and other authentication data. These mistakes often happen due to missing or weak settings for access control, encryption, or monitoring. The study highlights that protecting password data is not just about using the right cryptographic measures, but it also depends on securing the entire environment where that data is kept [55].

To prevent credential exposure, organizations must combine proper cryptographic techniques with strong environmental security measures. This includes not only technical safeguards, but also policy level controls and secure development workflows. Key principles for securing password storage environments include:

1.Implement Encryption at Rest

Encrypting data at rest is a critical security control that protects password storage systems from unauthorized access when physical or logical security fails. According to the National Institute of Standards and Technology (NIST), data at rest includes all digital information that

resides on persistent storage devices such as hard drives, SSDs, or cloud volumes, and it must be protected using cryptographic methods to ensure confidentiality and integrity.

NIST emphasizes that encryption at rest does not replace strong authentication or access control mechanisms but serves as an additional safeguard particularly useful in cases of stolen backups, improper access to storage devices, or insider threats [56].

2. **Secure Configuration Files and Application Code**

Storing sensitive information, such as credentials, in configuration files or embedding them directly into application source code poses significant security risks. A recent study conducted in 2022 highlights that secrets like API keys and database credentials are frequently exposed due to improper storage practices. They recommend using environment variables and external secret management services to securely store secrets and prevent accidental exposure through version control systems [57].

3.**Enforce the Principle of Least Privilege**

The principal of least privilege enforces that users and processes should operate with the possible minimal level of access needed to perform their roles. In their study Saltzer and Schroeder emphasize that adhering to this principle reduces the risk of intentionally or unintentionally misuse of privileges, thereby enhancing system security [58].

4. Regularly Audit and Monitor Access Logs

Regular monitoring and auditing of access logs are critical for detecting unauthorized access attempts and unusual activities. The National Institute of Standards and Technology (NIST) emphasizes that audit logs are vital for identifying security violations and ensuring individual accountability. They recommend that organizations establish comprehensive log management policies and procedures to effectively monitor and analyze system activities [59].

5. Educate Developers on Secure Coding Practices

Educating developers on secure coding is critical for minimizing possible vulnerabilities in software. A study that conducted a large-scale survey revealing that many developers lack

awareness of secure coding practices. They advocate for integrating security education into the software development lifecycle to improve adherence to secure coding standards [60].

To sum up, keeping passwords safe is not just about how we store them, it also requires protecting the places where these passwords are stored. Using encryption, limiting who can access password data, storing secrets securely, and training developers are all important steps. By combining these measures, we can reduce the risk of potential data breaches and leak of user credentials.

5.3 Multi Factor Authentication

In an era where password breaches remain one of the most critical vectors for unauthorized access, Multi-Factor Authentication (MFA) has emerged as an additional defense mechanism in digital security. MFA is designed to "fragment" the authentication process, meaning it transforms a single point of attack into two or more independent challenges. This fragmentation principle fundamentally alters the threat landscape: even if a user's password is compromised, the attacker must still overcome separate, unrelated factors such as a time-sensitive one-time password (OTP), a biometric signature, or a device-bound verification step [61].

Recent large-scale empirical research using Azure Active Directory users showed how effective this fragmentation really is. Among accounts protected by MFA (including those with previously compromised credentials) the overall risk of unauthorized access was reduced by 99.22%, and by 98.56% specifically for accounts known to have leaked passwords [62].

MFA mechanisms are typically classified into three categories:

1. Something You Know: e.g., a PIN or password.
2. Something You Have: e.g., a smartphone, smartcard or hardware token.
3. Something You Are: e.g., biometrics like retina scans, fingerprints or facial recognition.

Each method comes with trade-offs. OTPs sent via SMS, for example, are vulnerable to SIM swapping[29] and interception attacks, while biometrics raise privacy concerns and cannot be changed if compromised. Hardware tokens like YubiKeys[30] offer strong resistance to phishing and replay attacks but require physical possession, making deployment costlier and less scalable in some environments.

NIST guidelines recommend combining factors that are fundamentally different (e.g., password + hardware token) rather than relying on two similar forms (e.g., password + OTP via SMS), since this minimizes the risk of a single failure mode compromising the system [13]

Despite its crucial role in reducing unauthorized access, MFA can still be compromised through a variety of attack vectors that exploit implementation weaknesses, user behavior, or protocol design. Common attack types include:

- **Man-in-the-Middle (MitM) Attacks**

  Tools like Modlishka and Evilginx2 operate as transparent reverse proxies, sitting between the user and the target service. These tools intercept credentials and session tokens, allowing attackers to bypass MFA by replaying the session [63]. These "Adversary-in-the-Middle" attacks are currently among the most dangerous MFA bypass techniques, especially when combined with real-time phishing pages and TLS termination control [63].

- **Token Theft and Replay Attacks**

  In some scenarios, attackers use malware to extract session tokens from a victim's device. Once the token is obtained, it can be reused to authenticate without re-entering MFA. This includes access tokens, refresh tokens, or device-specific secrets used in OAuth-based logins [63].

[29] https://www.corbado.com/blog/sms-cost-reduction-passkeys/sim-swapping-sms-authentication-risk
[30] https://www.yubico.com

- **Push Bombing (MFA Fatigue Attacks)**

MFA systems that rely on user approval via mobile prompts (e.g., Microsoft Authenticator, Duo Push) are susceptible to push bombing, where an attacker repeatedly sends MFA requests hoping the victim eventually accepts one out of fatigue[31].

- **SIM-Swapping and SMS OTP Interception**

SMS based MFA is particularly susceptible to SIM-swapping attacks. They are type of attacks where adversaries manipulate telecom services into transferring a victim's phone number to a SIM card under their control. This enables them to intercept one-time passcodes (OTPs) and compromise accounts[32].

In summary, Multi-Factor Authentication (MFA) remains one of the most effective measures against unauthorized access by significantly lowering the risk posed by password only systems. However, its implementation must be carefully planned, with attention to usability, hardware requirements, and emerging attack vectors. When combined with secure password storage practices, MFA acts as a critical component in modern authentication systems.

5.4 Rise of Passkeys and Passwordless Authentication

When implemented according to industry standards, password-based authentication remains a secure and reliable method. However, as demonstrated throughout this study, many developers are unaware of the best practices and unable to follow standards about secure password storage. In addition, passwords can be difficult for users to manage or remember, which leads to insecure habits like reusing the same password or choosing weak ones. These challenges have encouraged the shift toward authentication without passwords. Passkeys emerging as a leading solution in this field. Based on public key cryptography, passkeys eliminate shared secrets and prevent many of the risks explored in this paper. Their growing use reflects a practical move to improve both security and usability, especially in environments where correct password handling cannot be guaranteed.

[31] https://www.beyondtrust.com/resources/glossary/mfa-fatigue-attack
[32] https://www.corbado.com/blog/sms-cost-reduction-passkeys/sim-swapping-sms-authentication-risk

Passkeys operate on a fundamentally different principle than traditional passwords. Rather than using a shared secret known to both the user and the server, passkeys rely on public key cryptography, where a key pair is generated: the private key remains stored on the user's device, and the public key shared to the server and stored. During authentication, the server uses this public key to verify a signed challenge sent by the user's device. This approach removes the need to transmit or store sensitive secrets. Passkeys are implemented through the FIDO2 standard, which is based on open specifications and includes two main components: WebAuthn [64] which facilitates communication between the browser and the server, and CTAP2 (Client to Authenticator Protocol[33]) which manages the interaction between the user's device and the authenticator [65]

Despite their strong security architecture and growing support across platforms, passkeys also have limitations, especially in terms of usability, interoperability, device management and single point of failure nature.

A major concern is device dependency. Since passkeys are designed to never leave the device where they are generated. Losing access to a device means potentially losing access to all associated passkeys. Although some password managers (e.g., Google Password Manager, iCloud Keychain) allow syncing or transferring passkeys across devices, this currently contradicts with the original WebAuthn constraint and creates inconsistencies across implementations. To address this, the FIDO Alliance is working on a formal solution called the Credential Exchange Protocol[34] (CXP), which would allow secure passkey transfer between trusted devices. However, as of now, CXP remains a draft and is not part of the FIDO2 standard [65].

Another challenge lies in account recovery. While traditional password systems often offer straightforward reset mechanisms via email or SMS, the process for recovering access to a lost passkey is more complex. The recovery mechanisms such as using another device with the same passkey or relying on synced cloud backups can be confusing or unavailable to some users. Moreover, if account recovery depends on email, then the email account becomes a single point of failure which undermines the security benefits of passkeys [65].

[33] https://fidoalliance.org/specs/fido-v2.0-ps-20190130/fido-client-to-authenticator-protocol-v2.0-ps-20190130.html
[34] https://fidoalliance.org/specs/cx/cxp-v1.0-wd-20240522.html

Lastly, secure implementation of authenticators is crucial. Some hardware-based authenticators rely on Trusted Platform Modules[35] (TPMs) or Trusted Execution Environments[36] (TEEs) for storing private keys. However, these components themselves can have vulnerabilities. For instance, a 2022 attack targeted TEEs [66] on certain Android devices, enabling bypass of FIDO2 WebAuthn security. Such attacks highlight the need for careful design and certification of authenticators. To address this, the FIDO Alliance maintains a certification program called FIDO Certification Program to classify authenticators based on their security levels.

In summary, while passkeys represent a promising advancement in digital authentication, they are prone to become a single point of failure and their success depends on ongoing challenges like device portability, account recovery, cross-platform support, and secure implementation of authentication.

[35] https://learn.microsoft.com/en-us/windows/security/hardware-security/tpm/trusted-platform-module-overview
[36] https://learn.microsoft.com/en-us/azure/confidential-computing/trusted-execution-environment

6. Conclusion

From the first spoken passwords to modern times, the challenge of authentication without relying on obscurity remained a fundamental problem throughout human history. With the development of cryptographic tools such as hash functions and key derivation algorithms we finally developed technical methods for securely storing passwords. However, we are still witnessing major data breaches that affect millions of people in the 21st century. This situation raises an obvious question: If we already know how to store passwords securely, why do so many systems still fail to do so?

This study aims to answer that question by showing that the core issue is not the absence of secure methods, but the failure to apply them often due to oversight, lack of awareness, or underestimating potential consequences in implementation. To help with that problem this project explored the best practices in password storage, with providing an in-depth look into how modern cryptographic mechanisms work, where they can fail, and how they should be correctly applied. It also expanded the discussion to include complementary measures, approved industry standards as well as evaluating secure storage practices. Lastly it focuses on emerging trends such as multi-factor authentication and passkeys to assess their potential benefits and limitations.

Key points from this study can be listed as:

- Use a memory hard key derivation function such as scrypt or Argon2 whenever possible.
- If it is not possible to use these functions, ensure the chosen hash function is properly salted and iterated based on the system's security needs.
- Implement multi-factor authentication and rate-limiting to mitigate online attacks.
- When appropriate implement additional security measures such as peppering or honeywords to strengthen the overall authentication mechanisms.
- Store hashed credentials in securely managed databases and conduct regular audits.
- Follow established industry standards and official guidelines while designing and implementing a password storage mechanism.

Ultimately, today the challenge of password storage is not about technical problems. It is about awareness, usability, and most importantly responsibility. Whether systems use passwords, passkeys, or something entirely new every system will have weaknesses that waits for exploiting, but one principle will remain constant: Deeply understanding what is best and applying it carefully.

References

[1] S. Boonkrong and C. Somboonpattanakit, "Dynamic salt generation and placement for secure password storing," *IAENG International Journal of Computer Science,* vol. 43, no. 1, 2016.

[2] J. Bonneau, C. Herley , P. C. Van Oorschot and F. Stajano, "The quest to replace passwords: A framework for comparative evaluation of web authentication schemes.," in *IEEE Symposium on Security and Privacy, 2012*, (2012).

[3] A. Naiakshina, A. Danilova, C. Tiefenau, M. Herzog, S. Dechand and M. Smith, "Why do developers get password storage wrong? A qualitative usability study.," in *2017 ACM SIGSAC Conference on Computer and Communications Security (CCS'17)*, 2017.

[4] M. Kaminska and R. Franke, "Utility of hashing and salting algorithms in quality improvement studies," *Canadian Family Physician,* vol. 69(3), p. 215–216, 2023.

[5] Foundation, OWASP, "Password Storage Cheat Sheet," OWASP, 2023. [Online]. Available: https://cheatsheetseries.owasp.org/cheatsheets/Password_Storage_Cheat_Sheet. [Accessed 2025].

[6] R. E. Smith, Authentication: from passwords to public keys., Addison-Wesley Longman Publishing Co., Inc.., 2001.

[7] R. Morris and K. Thompson, "Password security: A case history.," *Communications of the ACM,,* vol. 22, no. 11, pp. 594-597, 1979.

[8] Ntantogian, Christoforos, M. Stefanos and X. Christos, "Evaluation of password hashing schemes in open source web platforms.," *Computers & Security,* vol. 84, pp. 206-224, 2019.

[9] P. Software, "Brief analysis of RockYou leaked passwords," Passcape Software, 2012 https://www.passcape.com/text/articles/rockyou_leaked_passwords.pdf.

[10] N. Petru-Cristian, "A Comprehensive Analysis of High-Impact Cybersecurity Incidents: Case Studies and Implications," ResearchGate, October 2023.

[11] E. Bauman, Y. Lu and Z. Lin, "Half a century of practice: Who is still storing plaintext passwords?," in *11th International Conference on Information Security Practice and Experience (ISPEC 2015)*, Beijing, China, 2015.

[12] Verizon, "2024 Data Breach Investigations Report," Verizon, 2024 https://www.verizon.com/business/resources/reports/2024-dbir-data-breach-investigations-report.pdf.

[13] (NIST), National Institute of Standards and Technology, "Digital Identity Guidelines – Authentication and Lifecycle Management SP 800-63B," NIST, 2017 https://nvlpubs.nist.gov/nistpubs/specialpublications/nist.sp.800-63b.pdf.

[14] C. Wang, S. T. Jan, H. Hu, D. Bossart and G. Wang, "The next domino to fall: Empirical analysis of user passwords across online services," in *Proceedings of the Eighth ACM Conference on Data and Application Security and Privacy*, 2018.

[15] D. Wang, Z. Zhang, P. Wang, J. Yan and X. Huang, "Targeted Online Password Guessing: An Underestimated Threat," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security (CCS '16)*, 2016.

[16] J. Sushil, S. Pierangela and M. Yung, Encyclopedia of Cryptography, Security and Privacy, Third ed., Springer, 2024.

[17] Y. Zang, "Rainbow Tables," University of Minnesota, Morris, 2019 Zang, Y. (2019). Rainbow Tables. In URL https://umm-csci. github. io/seni or-seminar/seminars/fall2019/zang. pdf..

[18] I. Alkhwaja, M. Albugami, A. Alkhwaja and M. Alghamdi, "Password cracking with brute force algorithm and dictionary attack using parallel programming," *Applied Sciences,* vol. 13, 2023.

[19] H. Yang, . L. Peiyue and S. Yongxin, "The parallel computation in one-way hash function designing," in *2010 International Conference on Computer, Mechatronics, Control and Electronic Engineering*, Changchun, China, 2010.

[20] S. S. Thomsen, "Cryptographic Hash Functions," Technical University of Denmark PhD Thesis, Lyngby, Denmark, 2009.

[21] Zhong, Jinmin and X. Lai, "Improved preimage attack on one-block MD4," *Journal of Systems and Software,* vol. 85, no. 5, pp. 981-994, 2012.

[22] J. Kelsey, S. Chang, M. Sönmez Turan and . L. Chen, "Recommendation for password-based key derivation: Part 1: Storage applications," National Institute of Standards and Technology, Gaithersburg, 2010.

[23] R. L. Rivest, "The MD5 Message-Digest Algorithm," RFC Editor, 1992.

[24] Yu, X. Wang and H. , "How to break MD5 and other hash functions," in *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, Berlin, Heidelberg, 2005.

[25] M. Stevens, A. Sotirov, J. Appelbaum, A. Lenstra, D. Molnar, D. A. Osvik and B. De Weger, "Short chosen-prefix collisions for MD5 and the creation of a rogue CA certificate," in *Annual International Cryptology Conference*, Berlin, Heidelberg, 2009.

[26] K. Aoki and Y. Sasaki, "New Preimage Attacks Against Reduced SHA-1," in *Fast Software Encryption*, 2012.

[27] G. E. Moore, "Cramming more components onto integrated circuits," *Electronics,* vol. 38, no. 8, p. 114–117, 1965.

[28] F. Wang, C. Yang, Q. Wu and Z. Shi, "Constant Memory Optimizations in MD5 Crypt Cracking Algorithm on GPU-Accelerated Supercomputer Using CUDA," in *2012 7th International Conference on Computer Science & Education*, 2012.

[29] X. Wang, Y. L. Yin and H. Yu, "Finding Collisions in the Full SHA-1," in *Advances in Cryptology – CRYPTO 2005*, Santa Barbara, California, 2005.

[30] M. Stevens, E. Bursztein, P. Karpman, A. Albertini and Y. Markov, "The First Collision for Full SHA-1," Cryptology ePrint Archive, 2017.

[31] National Institute of Standards and Technology, "Secure Hash Standard (SHS)," NIST, 2015 https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.180-4.pdf.

[32] P. Sarkar and S. K. Sanadhya, "Attacking reduced round SHA-256," in *Applied Cryptography and Network Security: 6th International Conference, ACNS 2008*, New York, NY, USA, 2008.

[33] A. Lawrence and A. Lawrence Selvakumar, "Time Complexity Analysis and Comparison of SHA Algorithms," *Journal of Advanced Research in Dynamical and Control Systems,* 2019.

[34] N. Provos and D. Mazieres, "A future-adaptable password scheme," in *USENIX Annual Technical Conference, FREENIX Track*, 1999.

[35] B. Schneier, "Description of a new variable-length key, 64-bit block cipher (Blowfish)," in *International Workshop on Fast Software Encryption*, 1993.

[36] C. Percival, "Stronger key derivation via sequential memory-hard functions," in *BSDCan*, Ottawa, 2009.

[37] C. Percival and S. Josefsson, "The scrypt Password-Based Key Derivation Function RFC 7914," Internet Engineering Task Force (IETF), 2016 https://datatracker.ietf.org/doc/html/rfc7914.html.

[38] (NIST), National Institute of Standards and Technology, "SP 800-132 Initial Public Comments – Cryptographic Key Derivation and Password-Based Mechanisms," NIST, 2010 https://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-132.pdf.

[39] I. M. Verbauwhede, Secure Integrated Circuits and Systems, Introduction to Side-Channel Attacks, New York: Springer, 2009, p. 27–42.

[40] A. Biryukov, D. Dinu and D. Khovratovich, "Argon2: New generation of memory-hard functions for password hashing and other applications," in *2016 IEEE European Symposium on Security and Privacy (EuroS&P)*, 2016.

[41] A. Biryukov, D. Dinu, D. Khovratovich and S. Josefsson, "The memory-hard Argon2 password hash and proof-of-work function," Internet Research Task Force (IRTF), 2021.

[42] J. P. Aumasson, S. Neves, Z. Wilcox-O'Hearn and C. Winnerlein, "BLAKE2: Simpler, smaller, fast as MD5," in *International Conference on Applied Cryptography and Network Security*, Berlin, 2013.

[43] P. Tippe and M. P. Berner, "Evaluating Argon2 adoption and effectiveness in real-world software," arXiv, 2025.

[44] M. McGiffen, Pro Encryption in SQL Server 2022: Provide the Highest Level of Protection for Your Data, Berkeley, CA: Apress, 2022, pp. 269-275.

[45] J. Kelsey and S. Chang, "Recommendation for password-based key derivation: Part 1: Storage applications," National Institute of Standards and Technology, Gaithersburg, MD, 2010.

[46] Y. Asimi, A. Amghar, A. Asimi and Y. Sadqi, "New random generator of a safe cryptographic salt per session," *International Journal of Network Security,* vol. 18, no. 3, p. 445–453, 2016.

[47] R. L. Rivest and A. Juels, "Honeywords: Making password-cracking detectable," in *2013 ACM SIGSAC conference on Computer & communications security*, 2013.

[48] R. Anderson, Security engineering: A guide to building dependable distributed systems, 3rd ed., Wiley , 2020.

[49] Technology, National Institute of Standards and, "FIPS PUB 180-4: Secure Hash Standard (SHS)," U.S. Department of Commerce, 2015.

[50] International Organization for Standardization, "SO/IEC 27001:2022 — Information technology – Security techniques – Information security management systems – Requirements," ISO, 2022 https://www.iso.org/standard/27001.

[51] International Organization for Standardization, "ISO/IEC 27002:2022 — Information security, cybersecurity and privacy protection — Information security controls," ISO, 2022 https://www.iso.org/standard/75652.html.

[52] International Organization for Standardization, "ISO/IEC 29115:2013 —Information technology — Security techniques — Entity authentication assurance framework," ISO, 2013 https://www.iso.org/standard/45138.html.

[53] K. Moriarty, B. Kaliski and A. Rusch, "PKCS #5: Password-Based Cryptography Specification Version 2.1 (RFC 8018)," Internet Engineering Task Force, 2017.

[54] A. Krause, J. H. Klemme, N. Huaman, D. Wermke, Y. Acar and S. Fahl, "Committed by accident: Studying prevention and remediation strategies against secret leakage in source code repositories," 2022.

[55] C. S. Alliance, "The Common Cloud Misconfigurations That Lead to Cloud Data Breaches," 2023. [Online]. Available: https://cloudsecurityalliance.org/blog/2023/10/11/the-common-cloud-misconfigurations-that-lead-to-cloud-data-breaches.

[56] K. Scarfone, M. Souppaya and M. Sexton, "NIST SP 800-111 Guide to storage encryption technologies for end user devices," National Institute of Standards and Technology, 2007.

[57] S. K. Basak, L. Neil, B. Reaves and L. Williams, "What are the practices for secret management in software artifacts?," in *2022 IEEE Secure Development Conference (SecDev)*, 2022.

[58] J. H. Saltzer and M. D. Schroeder, "The protection of information in computer systems," *Proceedings of the IEEE,* vol. 63, no. 9, p. 1278–1308, 1975.

[59] K. A. Scarfone and M. P. Souppaya, "Cybersecurity Log Management Planning Guide," National Institute of Standards and Technology (NIST), 2023.

[60] T. E. Gasiba, U. Lechner, . M. Pinto Albuquerque and D. Mendez, "Is Secure Coding Education in the Industry Needed? An Investigation Through a Large Scale Survey," in

2021 IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering Education and Training (ICSE-SEET), 2021.

[61] R. K. Banyal, P. Jain and V. K. Jain, "Multi-factor authentication framework for cloud computing," in *2013 Fifth International Conference on Computational Intelligence, Modelling and Simulation*, 2013.

[62] L. Meyer, S. Romero, G. Bertoli, T. Burt, A. Weinert and J. Lavista Ferres, "How effective is multifactor authentication at deterring cyberattacks?," arXiv, 2023.

[63] H. Mehtälä, "The cybercriminal arsenal: MFA attacks," JAMK University of Applied Sciences, Jyväskylä, Finland, 2024 Bachelor´s thesis.

[64] (W3C), World Wide Web Consortium, "Web Authentication: An API for accessing Public Key Credentials," 2021. [Online]. Available: https://www.w3.org/TR/webauthn/.

[65] S. G. Doğan, "A survey on password-free authentication method: Passkey," Middle East Technical University, Ankara, 2025 Term Project.

[66] A. Shakevsky, E. Ronen and A. Wool, "Trust dies in darkness: Shedding light on Samsung's TrustZone keymaster design," in *Proceedings of the 31st USENIX Security Symposium (USENIX Security '22)*, 2022.