



UNIVERSITY  
OF SKÖVDE

## **Comparison of the strongest methods of cracking passwords and how to prevent them**

Bachelor's Degree Project in Informatics

First Cycle 30 credits

Spring term 2025

Student: Arad Shadmand

Supervisor: Muhammad Abbas Khan Abbasi

Examiner: Eva Söderström

## **ACKNOWLEDGMENT**

I wish to express my sincere gratitude to my examiner, Eva Söderström, for her insightful feedback and encouragement on this project. I am equally thankful to my supervisor, Muhammad Abbas Khan Abbasi, whose patient guidance and expert advice shaped a lot of this thesis. Thank you both for everything.

## **ABSTRACT**

Passwords remain the most common way to protect online accounts, yet fast legacy hashes such as MD5 make them dangerously easy to crack once a database is stolen. This study measures how four popular cracking techniques—brute force, dictionary, hybrid, and combo list perform against MD5 on modern hardware (RTX 3090 GPU, Ryzen 5800X CPU) virtualised under Proxmox. Then benchmark Hashcat and John the Ripper on both Linux and Windows guests, then compare time-to-crack for eleven test passwords that range from simple words to 20-character random strings. Results show hybrid and combo lists break common word-based passwords in milliseconds, while random 12-plus-character strings resist all attacks within a 24-hour window. The conclusion is that unsalted and salted MD5 is obsolete and recommend immediate migration to memory-hard functions such as Argon2id, paired with password managers or passphrase policies to balance usability and security.

Keywords: Cracking, Hashcat, Hashes, Hashing, MD5, SHA-1, SHA-256, Password Security, Salting, Dictionary Attack, Brute Force, Hybrid Attack, Argon2, bcrypt

# Contents

1	Introduction.....	1
1.1	Aims.....	1
2	Background.....	2
2.1	Preliminaries / Concepts.....	2
2.1.1	Passwords.....	2
2.1.2	Hashing.....	3
2.1.3	Salting.....	3
2.1.4	MD5.....	3
2.1.5	SHA-1 & SHA-512.....	4
2.1.6	Argon2 & bcrypt.....	4
2.1.7	Key Derivative Functions.....	5
2.1.8	SecLists.....	5
2.1.9	Hashcat.....	5
2.1.10	John the Ripper.....	5
2.1.11	Various cracking methods.....	6
2.2	Research background.....	6
3	Problem formulation.....	8
3.1	Study aims.....	8
3.2	Motivation.....	8
3.3	Limitations.....	9
4	Methodology.....	10
4.1	Scoping.....	10
4.2	Planning.....	11
4.3	Operation.....	11
4.4	Analysis & Interpretation.....	12
4.4.1	Theoretical Crack Time Calculation (Brute Force).....	12
4.4.2	Theoretical Crack Time Calculation (Dictionary & Combo).....	14
4.4.3	Theoretical Crack Time Calculation (Hybrid Attack).....	15
4.5	Presentation & Package.....	16
4.6	Ethical aspects.....	16

5 Designing the Experiment.....	17
5.1 Password Dataset.....	17
5.2 Operating Systems.....	18
5.3 Cracking Tools.....	18
6 Results.....	19
6.1 Brute-Force.....	19
6.2 Dictionary.....	23
6.3 Combo Lists.....	25
6.4 Hybrid.....	27
7 Discussion.....	29
7.1 OS / Tool Impact.....	29
7.2 Method Effectiveness.....	29
7.3 Implications.....	29
7.4 Password Usability.....	30
7.5 In relation to previous research.....	30
7.6 Ethical and societal aspects.....	30
8 Conclusion.....	31
8.1 Future work.....	32
References.....	33
Appendices.....	36
A. Average Speed Calculation.....	36
B. Iterations Calculation.....	37

# 1 Introduction

In today's digital world, passwords remain the cornerstone of user authentication, protecting everything from personal accounts to critical infrastructure. Despite the growing use of multi-factor authentication and biometric verification, passwords are still widely used due to their simplicity and low cost of implementation (Bonneau et al., 2015). However, this reliance has also made them a prime target for attackers.

Over the past decades, various methods of password-cracking have emerged, evolving in both complexity and effectiveness. Traditional brute force attacks, dictionary-based strategies, hybrid techniques, and combo list attacks have long been used by attackers to systematically guess or reverse-engineer passwords (Hitaj et al., 2017). More recently, generative adversarial networks such as PassGAN have demonstrated the ability to learn real-world password distributions and generate high-quality guesses (Hitaj et al., 2017).

The increasing sophistication of these techniques presents a critical challenge to individuals, organizations, and system administrators tasked with safeguarding digital systems. While much research has focused on how these attacks operate, fewer studies have taken a methodical approach to evaluating how well these attacks can be mitigated under controlled conditions.

## 1.1 Aims

This thesis aims to investigate four major methods of password-cracking: brute force, dictionary, combo, and hybrid attacks, and evaluate how they can be effectively prevented or neutralized.

The study will conduct a series of experiments against different types of passwords and defense mechanisms to identify which security strategies best withstand each type of attack. Factors such as password complexity, hashing algorithms, salting, and rate-limiting policies will be examined for their effectiveness in real-world scenarios.

The goal is not to find a one-size-fits-all solution, but to understand the strengths and limitations of current password defense mechanisms. By doing so, this research will provide practical insights for system administrators and security professionals who seek to design authentication systems that are secure and feasible to implement and maintain.

The reason for not picking more methods or using alternative methods is because of their prominence in the scene, where they are more widely supported compared to a method such as PCFG guessers or a GAN based language model.

## 2 Background

The background section provides an overview of password security and the evolution of hashing algorithms like MD5, SHA-1, SHA-512, bcrypt, and Argon2. The irreversible nature of hashes and the tools commonly used for password-cracking. This context establishes the foundation for understanding current vulnerabilities and the necessity for robust authentication mechanisms.

### 2.1 Preliminaries / Concepts

#### 2.1.1 Passwords

Passwords remain the most widely-deployed "something-you-know" credential because they cost nothing to issue, travel easily across protocols, and require no dedicated hardware on the user side (NIST, 2024). In the prevailing web workflow, they are submitted over TLS, hashed server-side, and then combined with contextual risk signals, device cookies, IP reputation, or geolocation, before access is granted or an extra factor is demanded (Seitz et al., 2024). Reports from 2023 show that more than 70% of high-traffic sites still treat a user-chosen string as the primary gate-keeper, even when SSO or OTPs are offered as add-ons (Parra & Montoya, 2024).

Large-scale leak analyses consistently reveal human biases toward dictionary words, names, and simple keyboard patterns (Zhang & Li, 2019). Even when complexity requirements exist, users typically satisfy them by appending a digit or symbol to an otherwise weak core, a behaviour that hybrid crackers exploit efficiently (Ye et al., 2019). Two strands of recent work seek better memorability-to-entropy ratios:

Mnemonic guidance, controlled experiments show that concrete phrase-building tips ("think of a vivid scene") outperform generic advice ("add more symbols") for producing longer, less-predictable secrets (Ye et al., 2019).

Pass-phrases, a 2022 ACSAC study found that five-word random phrases were remembered as easily as user-selected passwords while resisting offline cracking by several orders of magnitude (Wu et al., 2022).

Laboratory work also shows that short response-efficacy prompts messages emphasising why a strong password directly protects the individual, and yield statistically stronger choices than purely instructional nudges (Simon et al., 2025).

Password reuse is still endemic: a CHI-2022 experiment combining gaze and keystroke features was able to identify reused strings for 58 % of participants (Abdrabou et al., 2022). Modern policy, therefore, shifts from mandatory rotation to breach-triggered resets and proactive block-listing of known-compromised credentials (NIST, 2024).

Server-side best practice now pairs unique salts with memory-hard hash functions such as Argon2id, dramatically increasing the cost of offline guessing (Guo et al., 2019). On the client side, entropy meters that reject leaked substrings and visualise real-time strength improve composition without harming usability (Wu et al., 2022). Collectively, these findings suggest that passwords will persist, not as a standalone fortress, but as one resilient input in layered, risk-adaptive authentication systems (Seitz et al., 2024).

## 2.1.2 Hashing

Hashing turns a password into a short, fixed-length "fingerprint." The same password always makes the same fingerprint, but going backwards from that fingerprint to the original text should be practically impossible. To slow attackers down, today's best practice is to use memory-hard hash functions, Argon2id, scrypt, or bcrypt, that force a computer to spend noticeable time and RAM on every guess (IETF, 2021). A 2024 scan of thousands of public GitHub projects, however, found that many developers still pick fast general-purpose hashes like SHA-256, letting attackers test billions of guesses per second (Saran, 2024).

Draft U.S. guidelines now insist on memory-hard hashing and recommend gradually raising the "work factor," for example, after each successful login, so the defense stays ahead of cheaper hardware (NIST, 2024). When used correctly, hashing shifts attackers toward easier targets, such as phishing, because large-scale offline cracking becomes too costly.

## 2.1.3 Salting

A salt is a unique, per-account random value prepended or appended to the password before hashing; it ensures that identical secrets never share the same digest and nullifies pre-computed rainbow tables. A field study of freelance developers showed that nearly one-fifth either reused a single global salt or omitted it entirely, often after copying code snippets from forums, despite claiming high confidence in their security choices (Hallett et al., 2021). Best-practice guidance distilled from empirical security research recommends a 128-bit, cryptographically-random salt stored in plaintext alongside the digest; secrecy is unnecessary because the salt's sole job is to guarantee uniqueness (Saran, 2024).

Modern memory-hard functions, including Argon2id, embed the salt into their parameter block and remain resistant even if an attacker targets one specific account, provided each user's salt is distinct and long enough (IETF, 2021).

## 2.1.4 MD5

MD5 outputs a 128-bit digest and was once the "glue" hash for software downloads, X.509 certificates, and Unix password files. Its main attraction is speed: modern GPUs exceed 50 billion MD5 evaluations per second, making it highly attractive for offline cracking.

Unfortunately, MD5's Merkle-Damgård design exposes structural weaknesses. Practical identical-prefix collisions were computed as early as 2008 and by 2020.

Despite formal deprecation, the algorithm lingers: a 2022 survey of 134 public password leaks found that MD5 remained the single most common hash, covering 42 % of cracked credential records, largely because legacy PHP and vBulletin deployments have not been migrated (Tran et al., 2022)

Researchers continue to use MD5 as a baseline "exact-match" filter before applying slower similarity hashes, underscoring that while MD5 is fast and convenient, it offers virtually no collision resistance in 2025

### 2.1.5 SHA-1 & SHA-512

SHA-1 produces a 160-bit digest and was once the default hash in TLS certificates and software update systems. Its useful life ended when researchers demonstrated practical chosen-prefix collisions, two distinct files that hash to the same value, in 2020 (Leurent & Peyrin, 2020). Reflecting those results, NIST's transition guidance now bans SHA-1 for any new digital-signature or password-storage use (NIST, 2023).

SHA-512, by contrast, belongs to the stronger SHA-2 family and yields a 512-bit digest. It remains collision-resistant for general integrity checks, but its speed makes it poor as a stand-alone password hash: modern GPUs can still test billions of SHA-512 guesses per second. In practice, SHA-512 is acceptable only when wrapped in a key-derivation function such as PBKDF2-HMAC-SHA-512 with a high iteration count, or when combined with large salts and adaptive cost parameters. Even then, memory-hard schemes like Argon2id give defenders a better cost ratio (Saran, 2024).

### 2.1.6 Argon2 & bcrypt

Argon2 is the winner of the Password Hashing Competition (2015) and was standardised as RFC 9106 in 2021. It offers three flavours: Argon2i (I/O-bound, side-channel resistant), Argon2d (GPU-hard), and Argon2id (a hybrid that balances both) (IETF, 2021). Unlike PBKDF2 or bcrypt, Argon2 lets defenders tune both CPU-time and RAM per hash, forcing attackers to provision large, high-bandwidth memory for each guess. A 2024 performance study shows that a 1 GiB, 3-pass Argon2id setting can reduce GPU cracking throughput by more than 95 % compared with a SHA-512-based PBKDF2 of equal run-time (Saran, 2024). These properties have led the major frameworks, Django 4.1 and Laravel 10, to adopt Argon2id as their default password storage scheme.

bcrypt, introduced in 1999, extends the Blowfish cipher into an adaptive key-derivation function. It's built-in cost parameter doubles the required CPU work with each increment, letting defenders "turn up the dial" as hardware improves. Despite its age, bcrypt remains widely deployed; GitHub, WordPress, and many Linux PAM stacks still default to it, because no practical pre-image or collision attacks have been found (Hallett et al., 2021). However, bcrypt is limited to 72-byte inputs and uses only 4 kB of RAM, making it relatively GPU-friendly by modern standards. Comparative benchmarks in 2022 show top-end GPUs testing at around 50,000 bcrypt hashes/sec, whereas Argon2id at equivalent run-time achieves less than 3,000 guesses/sec (Hallett et al., 2021). For that reason, industry guidance now recommends bcrypt mainly for legacy compatibility and encourages new deployments to migrate to memory-hard alternatives such as Argon2.

## 2.1.7 Key Derivative Functions

A key derivation function (KDF) is a cryptographic mechanism that transforms an initial secret input into one or more cryptographic keys. In simple terms, it takes a piece of known data (for example, a user password or a master secret) and derives new keys with properties suitable for cryptographic use (such as appropriate length and unpredictability). KDFs are widely used in security protocols to generate strong working keys from a single secret. For instance, a fast KDF like HKDF (HMAC-based Key Derivation Function) quickly derives multiple keys from a high-entropy secret in encryption schemes, since the input (e.g. a session key) is already strong and not easily guessable. In contrast, when the input may be low-entropy as in the case of human passwords, KDFs employ additional measures to strengthen the output against attack (Biryukov et al., 2021).

## 2.1.8 SecLists

SecLists is a community-maintained GitHub repository that aggregates thousands of password dictionaries, username lists, and fuzzing payloads used in penetration testing and red-team exercises. The collection spans everything from the top 10 basic passwords to multi-gigabyte dumps harvested from public breaches, letting auditors choose a list that matches the risk profile of their target (Miessler, 2025).

Because the repo tags each file by purpose, "10-million-password-list-top-100000.txt," "rockyou-75.txt," "fuzzing-SQLI.txt", automation tools such as Hashcat can reference the exact word-list they need with a single flag. Researchers also use SecLists to measure real-world password entropy and to benchmark cracking hardware under reproducible conditions. Security teams, however, are urged to keep the repository on an isolated testing host; misplacing these lists can hand attackers a ready-made arsenal (Miessler, 2025).

## 2.1.9 Hashcat

Hashcat is the most widely used GPU-accelerated password-cracking framework and supports over 300 hash "modes," including Argon2, bcrypt, SHA-2, MD5, and proprietary vendor formats (Hashcat Project, 2024). Its modular kernel design lets attackers, or defenders in a red-team role, switch between dictionary, mask, rule, combinator, or hybrid attacks without changing binaries. Hashcat's rule engine is particularly powerful: terse expressions like `?:$1$s` can generate millions of permutations from a single seed word, reproducing common human tweaks (Tran et al., 2022). Because it writes only digests to disk, Hashcat is script-friendly. The project's GPLv3 licence and active Discord community ensure rapid support for new hash formats and GPU architectures.

### 2.1.10 John the Ripper

John the Ripper (JtR) predates Hashcat by a decade and remains the de facto CPU-based cracking suite for Unix, Windows, and macOS. The Jumbo patch, updated in April 2024, adds OpenCL and CUDA kernels plus support for 400+ hash types, from classic DES and SHA-crypt to blockchain wallets (Openwall, 2024). JtR offers four primary modes: single (username-aware mangling), word-list, incremental (Markov-based brute force), and external, a mini-language that lets researchers script arbitrary transformations at run-time. In academic comparisons, JtR often matches Hashcat on CPU-only targets and outperforms it on exotic file-format hashes because of its rapid plug-in cycle (Tran et al., 2022).

### 2.1.11 Various cracking methods

A brute-force attack enumerates every possible input until a hash collision is found. On MD5, the tactic is attractive because GPUs can evaluate more than 100 billion hashes per second with consumer hardware (Hashcat Project, 2024). An eight-character, mixed-case MD5 password can fall in minutes at that speed, while each additional character multiplies the character set by around 94 (Tran et al., 2022). Because the work factor is purely computational, defenders have little leverage once a database is leaked; the remedy is to avoid fast hashes like MD5 altogether or wrap them in a slow, memory-hard key derivation function.

Dictionary attacks replace exhaustive search with a ranked word-list such as rockyou.txt or SecLists' "10-million-passwords" file. Against MD5, Hashcat's -a 0 mode can test the entire 14 MB rockyou list (around 14 million entries) in under a second on a single RTX 3090 (Hashcat Project, 2024). Empirical studies show that such lists recover 45–60 % of MD5-hashed credentials from real breaches because users favour common words and keyboard patterns (Tran et al., 2022). Salts break pre-built MD5 dictionaries but cannot stop attackers from replaying the same list online if rate-limiting is weak.

Rainbow tables pre-compute chains of hashes and plaintexts, trading terabytes of storage for instant look-ups. Although GPUs now make on-the-fly brute force almost as fast, rainbow tables remain effective against legacy hash databases that lack salts. A single 132 GB rainbow set covering all eight-character alphanumeric characters will crack 96 % of unsalted MD5 passwords in seconds (Tran et al., 2022). Therefore, consider unique, per-user salts non-negotiable; even a four-byte salt multiplies the table size by 232, making pre-computation economically infeasible.

Hybrid attacks glue a dictionary core to brute-force masks, for example, "password" + ?d?d > password12. Hashcat's -a 6 (dict+mask) and -a 7 (mask+dict) modes can produce millions of plausible MD5 candidates per second while staying much smaller than full brute force (Hashcat Project, 2024). In real-world breach replays, hybrid rules crack an extra 10–20 % of MD5 hashes beyond the dictionary alone because they mimic the way users add digits or symbols to satisfy policy requirements (Hallett et al., 2021).

A combo list attack feeds a pre-built file in which each line already combines two tokens, for example, summer2023, letmein123, football!@, instead of concatenating words on the fly. SecLists hosts several such datasets, the most popular being "Xato-10-Million-combo.txt" (10,518,935 entries) (Miessler, 2025).

## 2.2 Research background

Cryptography and penetration testing are foundational pillars in cybersecurity, informing both the design and evaluation of secure systems. As threats evolve, it becomes crucial to not only assess whether cryptographic algorithms are theoretically secure but also how they perform in practice, under real-world constraints.

Touch (1996) provides a foundational performance analysis of MD5, examining both its cryptographic structure and real-world efficiency. His findings suggest that MD5, though once widely used, fails to meet modern performance and security standards, particularly in high-speed environments where it can no longer keep pace. This insight underscores the importance of evaluating algorithms not just for correctness but for practical deployment.

In contrast, Hutter (2015) examines cryptographic resilience from a hardware perspective. His case study on SHA-256 threshold implementations explores how hardware-based defenses can resist side-channel attacks, adding an important layer of complexity often absent from software-focused assessments. The study reinforces that an algorithm's strength depends not only on its design, but also on how and where it is implemented.

Further broadening the context, Milousi et al. (2024) introduce structured frameworks for vulnerability assessment, while Teat and Peltzverger (2016) address the growing risks posed by cloud-based attacks and the declining reliability of legacy hash functions like MD5 and SHA-1. Zhu et al. (2023) highlight the ongoing challenges in phasing out insecure cryptographic practices in legacy systems, showing how vulnerabilities persist due to outdated implementations and code reuse.

This thesis builds upon these diverse perspectives by focusing on MD5's resilience against brute-force attacks, using empirical testing to highlight the algorithm's limitations under modern threat conditions. By incorporating updated theoretical models and structured experimental methods, the research aims to provide insight into the practical risks of using legacy cryptographic functions.

## 3 Problem formulation

Passwords remain the most common method of user authentication in both consumer and enterprise systems. Despite the growing use of multi-factor authentication and cryptographic tokens, passwords are still relied upon for their simplicity and cost-effectiveness (Bonneau et al., 2012). However, this widespread reliance exposes users and systems to a variety of security threats, particularly when password storage practices are weak or outdated.

One such practice involves the use of legacy hashing algorithms like MD5, which was widely adopted in the 1990s for its speed and efficiency. Although once considered secure, MD5 has since been rendered obsolete due to numerous cryptographic weaknesses. Most notably, MD5 is vulnerable to collision attacks (Wang & Yu, 2005) and is extremely susceptible to brute-force and dictionary attacks because of its fast computation rate and lack of internal complexity (Wang et al., 2005).

While modern systems increasingly adopt slower and more secure hashing algorithms like bcrypt and Argon2, MD5 is still found in legacy systems, poorly maintained applications, and certain low-security environments (Florêncio & Herley, 2007). Given this, there is a need for focused research that assesses how password characteristics, such as length and complexity, affect the feasibility of brute-forcing MD5 hashes.

### 3.1 Study aims

The core research problem this thesis addresses is the continued vulnerability of MD5-hashed passwords, particularly in the context of modern password-cracking tools. Although the cryptographic community has long warned about MD5's weaknesses, there remains a lack of empirical, experiment-based analysis that shows how different types of passwords perform under brute-force conditions when hashed with MD5.

This thesis, therefore, investigates the following question:

*How does password complexity influence the time required to crack MD5-hashed passwords using four different cracking methods on different operating systems with different tools?*

The study aims to highlight not only the effectiveness (or lack thereof) of complexity as a standalone defense mechanism for MD5 hashes, but also to emphasize why secure storage practices must go beyond just enforcing stronger passwords

### 3.2 Motivation

Although MD5 is widely recognized as insecure, it is still used in various systems due to legacy code, ease of implementation, or lack of security awareness. This makes it critical to not only discourage its use but also to understand the limitations of relying on password complexity as a compensating control. By offering detailed, experiment-driven insight into the relationship between password complexity and MD5 crackability, this research helps to fill a gap between theoretical cryptographic analysis and practical system security.

The results will contribute to informed decision-making among system administrators, developers, and IT security professionals, reinforcing the need to retire obsolete algorithms and adopt a layered approach to authentication security.

### 3.3 Limitations

To maintain a controlled experimental environment, the following boundaries are set:

- Salts will be applied to the hashed passwords. This allows a focus on the raw effectiveness of cracking methods and hash algorithm strength.
- The character encoding for all password sets is limited to UTF-8, with a maximum character set of 128 characters, including upper/lowercase letters, digits, and special characters. This ensures uniformity in the complexity levels being tested.
- All experiments will be conducted on a virtualized operating system: Windows 11 Pro and Ubuntu 22.04. This includes both hash generation and password-cracking to avoid discrepancies in tool behavior or system performance that may occur across different platforms.

## 4 Methodology

This thesis follows the experimental methodology framework proposed by Wohlin et al. (2024), which segments the process into five core activities: Scoping, Planning, Operation, Analysis & Interpretation, and Presentation & Package.

A controlled experiment is a formal empirical study in which researchers manipulate one or more independent variables and measure the effect on a dependent variable while keeping other factors constant.

Wohlin et al.'s framework classifies such studies as interventional (the researcher actively intervenes and controls variables), as opposed to observational methods like surveys or case studies that collect data without influencing the subject of study

Given this thesis's aim, to measure and compare the performance of different password-cracking tools and attack strategies under consistent conditions, an experimental approach was the most suitable choice. A controlled experiment enabled systematic variation of each tool or strategy and direct measurement of outcomes (e.g., cracking speed), ensuring that any performance differences could be attributed to the tools or strategies themselves.

Alternative methods such as surveys or case studies would not provide the required control or objective performance data, and formal modeling would offer only theoretical predictions rather than empirical evidence. Thus, the experiment approach directly supports the study's goal by allowing a fair, repeatable comparison of cracking tools and strategies under controlled conditions.

## 4.1 Scoping

### **Object of Study:**

Passwords with varying levels of complexity and structure on different platforms, with both cracking tools and operating systems, are hashed using the MD5 algorithm.

### **Purpose:**

To understand how password structural complexity influences the time and feasibility of MD5 hash cracking using brute-force, dictionary, combo, and hybrid attacks with different operating systems and cracking tools.

### **Quality Focus:**

Time-to-crack (measured in seconds) and the theoretical number of required guesses.

### **Perspective:**

That of a network and system administration student assessing password vulnerability.

### **Context:**

1. An offline controlled virtualized environment running Windows 11 Pro using a single Nvidia RTX 3090 GPU at an average of (22658.2MH/s for MD5) using Hashcat.
2. An offline controlled virtualized environment running Windows 11 Pro using a single Ryzen 5800x CPU at an average of (1120.5MH/s for MD5) using John the Ripper.
3. An offline controlled virtualized environment running Linux Ubuntu 22.04 using a single Nvidia RTX 3090 GPU at an average of (31038.6MH/s for MD5) using Hashcat.
4. An offline controlled virtualized environment running Linux Ubuntu 22.04 using a single Ryzen 5800x CPU at an average of (1508.0MH/s for MD5) using John the Ripper.

## 4.2 Planning

### **Hypothesis:**

- $H_0$  (Null Hypothesis): Hardware and operating systems play no significant role in cracking passwords of differing complexities compared to using more complex cracking tools.
- $H_1$  (Alternative Hypothesis): Passwords with higher complexity (length and charset diversity) require significantly more time and computational effort to crack. Hardware and operating systems play an important role in reducing the time taken to crack passwords of differing complexities.

### **Independent Variable:**

- Password complexity (eleven formats, with increasing length and character diversity).
- Cracking method (brute force, dictionary, combo, hybrid).
- Cracking Tool
- Operating System

### **Dependent Variable:**

Estimated time and guess count required to crack each password using the various methods (practical and theoretical). For the setup of the passwords to be used in the experiment, NIST guidelines have been followed

### **Instrumentation:**

- Passwords are hashed using Windows PowerShell and the Linux Terminal for Ubuntu.
- The charset is UTF-8 (128), and salts are applied.

## **4.3 Operation**

### **Preparation:**

- Passwords are preselected to represent the NIST guidelines for creating passwords. All passwords are hashed in MD5 using built-in Windows tools.

### **Execution:**

- Each hash is subjected to exactly 10 successful attack attempts using the four different attack methods with Hashcat.
- For passwords not cracked, theoretical analysis is used instead.

### **Validation:**

- Output logs from Hashcat are recorded.

## **4.4 Analysis & Interpretation**

### **4.4.1 Theoretical Crack Time Calculation (Brute Force)**

The number of guesses,  $G$ , is calculated using the following formula:

$$G = C \left( \frac{C^{L-1} - 1}{C - 1} \right) + \sum_{i=0}^{L-1} P_i C^i + 1$$

### **Explanation:**

- $C$  is the size of the character set (e.g., 10 for numbers, 26 for lowercase, 52 for upper+lower, 72+ for mixed with symbols),
- $L$  is the length of the password,
- $p_i$  is the positional value of the  $i$ 'th letter.
- The result  $G$  gives the total guesses required (Hashes).

The formula consists of two parts, one contains all passwords shorter (L-1) than the target password and the other the position of the target password in the length L.

$$C \left( \frac{C^{L-1} - 1}{C - 1} \right)$$

This formula uses the character set (C) used by the brute force method and the length of the target password (L). The brute force method will start with the shortest length working its way up till it cracks the password. The number of hashes required till the target password length is reached is therefore a worst case scenario calculation each time or a calculation of all possible combinations from 1 to L-1.

To show this, a general term using  $i$  to define the length is presented. Passwords of length of length  $i$  have a total number of possibilities equal to the character set to the power of the length  $i$ . This gives us  $C^i$  for the total possibilities at each password length. The total possibilities for a sequence of increasing lengths from 1 to L-1 can then be defined as,

$$S = C + C^2 + C^3 + \dots + C^{L-1} \text{ or a general term, } \sum_{n=0}^{L-1} C^n$$

The passwords from 1 to L-1 have now been defined as a finite geometric sequence. The formula for a geometric sequence is defined as the following. Then define  $a$  as the first term,  $r$  as the common ratio, and  $i$  the number of terms minus 1.

$$\sum_{k=0}^n ar^k = a \left( \frac{r^{n+1} - 1}{r - 1} \right)$$

Using this formula you can substitute in our previously defined geometric sequence for the sum of all of all possible passwords from 1 to L-1. This gives us the following formula for the first part of the equation.

$$\sum_{n=0}^{L-1} C^n ar^k = C \left( \frac{C^{L-1} - 1}{C - 1} \right)$$

For the second part of the equation the number of guesses required depends on the targeted passwords position in the lexicographic order the brute force method is using. It is previously defined by

$$\sum_{i=0}^{L-1} P_i C^i$$

The equation looks to solve for a password of length L defining it as.

$$P = P_0 P_1 P_2 \dots P_{L-1}$$

The characters in the password then need to be ranked on their rate of change for the lexicographic ordering. This gives the following equation for the password possibilities before the targeted password.

*Rank of password within length L*

$$= P_0 \times C^0 + P_1 \times C^1 + P_2 \times C^2 + \dots + P_{L-1} \times C^{L-1}$$

For a password of length L the following formula is given.

$$\sum_{i=0}^{L-1} P_i C^i$$

For a password of length L = 1 the rank is defined as.

$$P_0 \times C^0 \text{ This is equal to } P_0$$

The password's rank is therefore defined by its place in the lexicographic ordering which matches the amount of guesses that would be required to crack it.

Then a password of length L + 1 results in an equation defined using lexicographic rules as the following.

$$P = P_0 P_1 P_2 \dots P_L$$

The character  $P_L$  increases the number of possibilities by  $P_L \times C^L$ , this matches the previous equations.

The position of a targeted password within passwords of the same length can be defined by lexicographic ordering positional numbering.

The two parts shown comprise the number of total possibilities before length L and the rank of the targeted password within length L. Combined these values provide the number of hashes required to reach a targeted password with a brute force method.

**Example:**

If the character set is A, B, C with assigned values of 0, 1, 2, and our password is ABC (=  $[012]_3$ ), then according to the formula, the number of guesses is:

$$\begin{aligned}
 G &= 3\left(\frac{3^2-1}{3-1}\right) + \sum_{i=0}^{3-1} P_i 3^i + 1 \\
 &= 3\left(\frac{8}{2}\right) + 0 \times 3^0 + 1 \times 3^1 + 2 \times 3^2 + 1 \\
 &= 3 \times 4 + 0 + 3 + 18 + 1 \\
 G &= 34
 \end{aligned}$$

This can be verified through the list of guesses.

1-letter	A	B	C						
2-letter	AA	BA	CA	AB	BB	CB	CA	CB	CC
3-letter	AAA AAB AAC	BAA BAB BAC	CAA CAB CAC	ABA ABB ABC	BBA BBB	CBA CBB	ACA ACB	BCA BCB	CCA CCB

Table 4-1: Full list of guesses, adding up to 34.

### 4.4.2 Theoretical Crack Time Calculation (Dictionary & Combo)

The time taken to crack the password if it is in the dictionary, T, is calculated using the following formula:

$$T = \frac{D}{R}$$

**Explanation:**

- T is the time taken to crack
- D is the size of the dictionary being used
- R is the hash rate used in the crack

The average time taken to crack the password if it is randomly placed in the dictionary  $T_{avg}$  is calculated using the following formula:

$$T_{avg} = \frac{D}{2R}$$

**Explanation:**

- T is the time taken to crack
- D is the size of the dictionary being used
- R is the hash rate used in the crack

If the password is not in the dictionary, then the entire dictionary will be exhausted, and the attack fails:

$$T = \frac{D}{R}, \text{ and the result is "not found".}$$

**Example:**

Suppose that:

- $D = 1,000,000$  (1 million dictionary words)
- $R = 500,000$  hashes/sec (eg, with a CPU)

Then:

- Worst-case time

$$T = \frac{1,000,000}{500,000} = 2 \text{ seconds}$$

- Average-case time

$$T_{avg} = \frac{1,000,000}{2 \times 500,000} = 1 \text{ second}$$

### 4.4.3 Theoretical Crack Time Calculation (Hybrid Attack)

The time taken to crack the password if it is in the dictionary, T, is calculated using the following formula:

$$T = \frac{D \times V}{R}$$

**Explanation:**

- T is the time taken to crack
- D is the size of the dictionary being used
- V is the number of variations in the brute force attack that is being applied
- R is the hash rate used in the crack

**Example:**

A hybrid attack uses a word list, such as a dictionary or a combo list, as its base, and then attempts to apply a brute force approach to finding the correct variation. The list size defines the base number of hashes to be tested, and the number of variations defines the level of brute force being applied. A list of one million words being tested for the addition of a singular digit added at the end would have a base of one million and a variation of 10.

$$20 = \frac{1,000,000 \times 10}{500,000}$$

Assuming a hash rate of 500,000, this would take 20 seconds to crack. This is ten times longer to crack than the previous dictionary examples, which match the variations that were introduced.

## **4.5 Presentation & Package**

All experiment steps, hash values, Hashcat outputs, and theoretical math are stored in a structured folder system and mirrored on Google Drive for replication purposes. Zotero is used for managing academic references. Results will be visualized in tables showing an exponential increase in crack time with complexity.

For reproducibility, the lab package includes, the scripts used, hardware specifications, config files and all calculated intermediate values.

## **4.6 Ethical aspects**

All experiments will be conducted using artificially generated passwords and hashes. No real user accounts, systems, or breached data will be involved. The purpose of the study is solely educational and academic.

## 5 Designing the Experiment

This chapter is relevant to the setup of the experiment environment.

### 5.1 Password Dataset

The synthetic list was created with some general patterns using the NIST guidelines for password creation, and is used in the experiment for the cracking (Table 5-1).

ID	Format	Sample Password	Patterns
1	8 digits	92634817	Bank Device Pin Code
2	8 lowercase letters	computer	Generic word password
3	8 letters, lower and uppercase	Learning	Generic word password with one uppercase letter
4	3 letters, 1 uppercase, 2 digits, 1 symbol	Dan2003!	Password combining name and age with a symbol
5	4 letters, 1 uppercase, 4 digits, 1 symbol	Olof2001!	Password combining name and age with a symbol
6	6 letters, 1 uppercase, 4 digits, 1 symbol	Forest3142#	Password combining a theme with numbers and a symbol
7	8 letters, 1 uppercase, 4 digits, 1 symbol	Brighter7639%	Password combining a theme with numbers and a symbol
8	12-character randomized	8!Gc3%Lx9e@p	Randomized string from a password manager
9	18-character phrase, lower case	thisstrongpassword	NIST guideline phrase
10	20-letter phrase, lower case	skovdemountainishill	NIST guideline phrase
11	20-character phrase, lower and upper case	SkovdeMountainIsHill	NIST guideline phrase

**Table 5-1: Password Dataset Table**

## 5.2 Operating Systems

The experiment setup uses Ubuntu 22.04 LTS and Windows 11 Pro (23H2) to observe how the host OS affects cracking speed and tooling. Prior benchmarks show 5-10 % higher hash-throughput for identical NVIDIA cards under Linux because of lower driver overhead (Hashcat Project, 2024).

Windows 11 is retained for ecological validity: many corporate red teams run Hashcat on mixed-purpose RTX workstations.

The experiment is based on these operating systems, running virtualized through Proxmox with an RTX 3090 GPU, Ryzen 5800X CPU, and a Crucial P3 M.2 NVMe SSD 1 TB (split evenly).

## 5.3 Cracking Tools

Hashcat 6.2.7 provides GPU-accelerated brute-force, dictionary, hybrid, and combo modes and supports all MD5 variants (raw-MD5, phpBB, vBulletin). Its potfile makes result aggregation trivial (Hashcat Project, 2024).

John the Ripper “Jumbo” complements Hashcat with CPU-oriented single and incremental modes, plus a flexible external scripting language (Openwall, 2024). Using both tools allows for cross-validation of cracks and comparison of speed (GPU vs. CPU) without changing rule syntax or data formats.

## 6 Results

This section contains all of the completed test results. All test results are based on the average of 10 different outputs, except for IDs 5 to 11, which are based on the formula presented in the methodology.

### 6.1 Brute-Force

Plain eight-character MD5 strings (IDs 1-3) fell in 0.014 s – 2 min on the RTX 3090 (Linux), but estimates exponentially increase to  $5.6 \times 10^{29}$  years for 20-character phrases (ID 11) (Table 6-4). CPU cracking under JtR is roughly 20× slower but shows the same exponential curve (Table 6-3).

ID	Hash Guesses (amt) (Linux)	Time to crack (s) (Linux)	Time to crack (y) (Linux)	Hash Guesses (amt) (Windows)	Time to crack (s) (Windows)	Time to crack (y) (Windows)
1	2.09E+07	1.39E-02	4.39E-10	2.07E+07	1.85E-02	5.86E-10
2	2.46E+11	1.63E+02	5.17E-06	2.60E+11	2.32E+02	7.35E-06
3	1.89E+13	1.25E+04	3.97E-04	1.89E+13	1.68E+04	5.34E-04
4	N/A	N/A	N/A	N/A	N/A	N/A
5	N/A	N/A	N/A	N/A	N/A	N/A
6	N/A	N/A	N/A	N/A	N/A	N/A
7	N/A	N/A	N/A	N/A	N/A	N/A
8	N/A	N/A	N/A	N/A	N/A	N/A
9	N/A	N/A	N/A	N/A	N/A	N/A
10	N/A	N/A	N/A	N/A	N/A	N/A
11	N/A	N/A	N/A	N/A	N/A	N/A

**Table 6-1: Time taken to complete a crack for specific passwords using two different operating systems with the same hardware using the John the Ripper cracking tool (authors' own)**

ID	Hash Guesses (amt) (Linux)	Time to crack (s) (Linux)	Time to crack (y) (Linux)	Hash Guesses (amt) (Windows)	Time to crack (s) (Windows)	Time to crack (y) (Windows)
1	2.01E+07	6.49E-04	2.06E-11	2.13E+07	9.38E-04	2.97E-11
2	2.62E+11	8.44E+00	2.68E-07	2.51E+11	1.11E+01	3.51E-07
3	1.83E+13	5.90E+02	1.87E-05	1.89E+13	8.32E+02	2.64E-05
4	4.58E+15	1.47E+05	4.68E-03	4.49E+15	1.98E+05	6.28E-03
5	N/A	N/A	N/A	N/A	N/A	N/A
6	N/A	N/A	N/A	N/A	N/A	N/A
7	N/A	N/A	N/A	N/A	N/A	N/A
8	N/A	N/A	N/A	N/A	N/A	N/A
9	N/A	N/A	N/A	N/A	N/A	N/A
10	N/A	N/A	N/A	N/A	N/A	N/A
11	N/A	N/A	N/A	N/A	N/A	N/A

**Table 6-2: Time taken to complete a crack for specific passwords using two different operating systems with the same hardware using the Hashcat cracking tool (authors' own)**

ID	Hash Guesses (amt) (Linux)	Time to crack (s) (Linux)	Time to crack (y) (Linux)	Hash Guesses (amt) (Windows)	Time to crack (s) (Windows)	Time to crack (y) (Windows)
1	1.85E+07	1.23E-02	3.89E-10	1.85E+07	1.65E-02	5.23E-10
2	2.30E+11	1.52E+02	4.83E-06	2.30E+11	2.05E+02	6.51E-06
3	1.80E+13	1.19E+04	3.78E-04	1.80E+13	1.60E+04	5.08E-04
4	4.40E+15	2.92E+06	9.26E-02	4.40E+15	3.93E+06	1.25E-01
5	7.34E+15	4.87E+06	1.54E-01	7.34E+15	6.55E+06	2.08E-01
6	3.89E+21	2.58E+12	8.19E+04	3.89E+21	3.47E+12	1.10E+05
7	3.63E+25	2.40E+16	7.62E+08	3.63E+25	3.24E+16	1.03E+09
8	1.53E+23	1.01E+14	3.22E+06	1.53E+23	1.36E+14	4.33E+06

9	5.98E+34	3.96E+25	1.26E+18	5.98E+34	5.34E+25	1.69E+18
10	8.39E+38	5.56E+29	1.76E+22	8.39E+38	7.49E+29	2.37E+22
11	8.39E+38	5.56E+29	1.76E+22	8.39E+38	7.49E+29	2.37E+22

**Table 6-3: Theoretical time taken to crack for specific passwords using CPU hardware and the John the Ripper cracking tools (authors' own)**

ID	Hash Guesses (amt) (Linux)	Time to crack (s) (Linux)	Time to crack (y) (Linux)	Hash Guesses (amt) (Windows)	Time to crack (s) (Windows)	Time to crack (y) (Windows)
1	1.85E+07	5.96E-04	1.89E-11	1.85E+07	8.16E-04	2.59E-11
2	2.30E+11	7.41E+00	2.35E-07	2.30E+11	1.01E+01	3.22E-07
3	1.80E+13	5.79E+02	1.83E-05	1.80E+13	7.93E+02	2.51E-05
4	4.40E+15	1.42E+05	4.50E-03	4.40E+15	1.94E+05	6.16E-03
5	7.34E+15	2.36E+05	7.50E-03	7.34E+15	3.24E+05	1.03E-02
6	3.89E+21	1.25E+11	3.98E+03	3.89E+21	1.72E+11	5.45E+03
7	3.63E+25	1.17E+15	3.70E+07	3.63E+25	1.60E+15	5.07E+07
8	1.53E+23	4.93E+12	1.56E+05	1.53E+23	6.75E+12	2.14E+05
9	5.98E+34	1.93E+24	6.11E+16	5.98E+34	2.64E+24	8.37E+16
10	8.39E+38	2.70E+28	8.57E+20	8.39E+38	3.70E+28	1.17E+21
11	8.39E+38	2.70E+28	8.57E+20	8.39E+38	3.70E+28	1.17E+21

**Table 6-4: Theoretical time taken to crack for specific passwords using GPU hardware and Hashcat cracking tools. (authors' own)**

ID	Hash Guesses (amt) (Linux)	Time to crack (s) (Linux)	Time to crack (y) (Linux)	Hash Guesses (amt) (Windows)	Time to crack (s) (Windows)	Time to crack (y) (Windows)
1	1.22E+09	8.12E-01	2.58E-08	1.22E+09	1.09E+00	3.47E-08
2	3.45E+23	2.29E+14	7.26E+06	3.45E+23	3.08E+14	9.77E+06
3	1.04E+47	6.92E+37	2.20E+30	1.04E+47	9.32E+37	2.95E+30
4	7.10E+85	4.71E+76	1.49E+69	7.10E+85	6.34E+76	2.01E+69
5	5.06E+90	3.36E+81	1.06E+74	5.06E+90	4.52E+81	1.43E+74

6	9.41E+98	6.24E+89	1.98E+82	9.41E+98	8.40E+89	2.66E+82
7	7.23E+105	4.80E+96	1.52E+89	7.23E+105	6.46E+96	2.05E+89
8	3.63E+102	2.41E+93	7.63E+85	3.63E+102	3.24E+93	1.03E+86
9	1.89E+119	1.25E+110	3.97E+102	1.89E+119	1.68E+110	5.34E+102
10	4.17E+123	2.77E+114	8.77E+106	4.17E+123	3.72E+114	1.18E+107
11	4.17E+123	2.77E+114	8.77E+106	4.17E+123	3.72E+114	1.18E+107

**Table 6-5: Theoretical worst-case time taken to crack for specific passwords using GPU hardware and the John the Ripper cracking tool (authors' own)**

ID	Hash Guesses (amt) (Linux)	Time to crack (s) (Linux)	Time to crack (y) (Linux)	Hash Guesses (amt) (Windows)	Time to crack (s) (Windows)	Time to crack (y) (Windows)
1	1.22E+09	3.95E-02	1.25E-09	1.22E+09	5.41E-02	1.71E-09
2	3.45E+23	1.11E+13	3.53E+05	3.45E+23	1.52E+13	4.83E+05
3	1.04E+47	3.36E+36	1.07E+29	1.04E+47	4.61E+36	1.46E+29
4	7.10E+85	2.29E+75	7.26E+67	7.10E+85	3.14E+75	9.94E+67
5	5.06E+90	1.63E+80	5.17E+72	5.06E+90	2.23E+80	7.08E+72
6	9.41E+98	3.03E+88	9.62E+80	9.41E+98	4.15E+88	1.32E+81
7	7.23E+105	2.33E+95	7.39E+87	7.23E+105	3.19E+95	1.01E+88
8	3.63E+102	1.17E+92	3.71E+84	3.63E+102	1.60E+92	5.08E+84
9	1.89E+119	6.08E+108	1.93E+101	1.89E+119	8.33E+108	2.64E+101
10	4.17E+123	1.34E+113	4.26E+105	4.17E+123	1.84E+113	5.84E+105
11	4.17E+123	1.34E+113	4.26E+105	4.17E+123	1.84E+113	5.84E+105

**Table 6-6: Theoretical worst-case time taken to crack for specific passwords using CPU hardware and the Hashcat cracking tool (authors' own)**

## 6.2 Dictionary

Only the lowercase noun computer (ID 2) appeared in the rockyou list, cracking in  $8.3 \times 10^{-5}$  seconds on Linux/CPU and  $4 \times 10^{-6}$  s on Linux/GPU. All mixed-case and symbol-rich variants evaded the list completely (Tables 6-7 & 6-8).

ID	Hash Guesses (amt) (Linux)	Time to crack (s) (Linux)	Time to crack (y) (Linux)	Hash Guesses (amt) (Windows)	Time to crack (s) (Windows)	Time to crack (y) (Windows)	Succeeded (Yes/No)
1	N/A	N/A	N/A	N/A	N/A	N/A	No
2	1.25E+05	8.31E-05	2.64E-12	1.12E-04	1.12E-04	3.55E-12	Yes
3	N/A	N/A	N/A	N/A	N/A	N/A	No
4	N/A	N/A	N/A	N/A	N/A	N/A	No
5	N/A	N/A	N/A	N/A	N/A	N/A	No
6	N/A	N/A	N/A	N/A	N/A	N/A	No
7	N/A	N/A	N/A	N/A	N/A	N/A	No
8	N/A	N/A	N/A	N/A	N/A	N/A	No
9	N/A	N/A	N/A	N/A	N/A	N/A	No
10	N/A	N/A	N/A	N/A	N/A	N/A	No
11	N/A	N/A	N/A	N/A	N/A	N/A	No

**Table 6-7: Time taken to complete a crack for specific passwords using two different operating systems with the same hardware, using a dictionary list and John the Ripper (authors' own)**

ID	Hash Guesses (amt) (Linux)	Time to crack (s) (Linux)	Time to crack (y) (Linux)	Hash Guesses (amt) (Windows)	Time to crack (s) (Windows)	Time to crack (y) (Windows)	Succeeded (Yes/No)
1	N/A	N/A	N/A	N/A	N/A	N/A	No
2	1.25E+05	4.04E-06	1.28E-13	1.25E+05	5.53E-06	1.75E-13	Yes
3	N/A	N/A	N/A	N/A	N/A	N/A	No
4	N/A	N/A	N/A	N/A	N/A	N/A	No
5	N/A	N/A	N/A	N/A	N/A	N/A	No
6	N/A	N/A	N/A	N/A	N/A	N/A	No
7	N/A	N/A	N/A	N/A	N/A	N/A	No
8	N/A	N/A	N/A	N/A	N/A	N/A	No
9	N/A	N/A	N/A	N/A	N/A	N/A	No
10	N/A	N/A	N/A	N/A	N/A	N/A	No
11	N/A	N/A	N/A	N/A	N/A	N/A	No

**Table 6-8: Time taken to complete a crack for specific passwords using two different operating systems with the same hardware, using a dictionary list and Hashcat (authors' own)**

## 6.3 Combo Lists

The combo list processed 10.5 million entries in around 1 ms on the RTX 3090 but still cracked only 18 % (2 / 11) of the dataset (Tables 6-9 & 6-10).

ID	Hash Guesses (amt) (Linux)	Time to crack (s) (Linux)	Time to crack (y) (Linux)	Hash Guesses (amt) (Windows)	Time to crack (s) (Windows)	Time to crack (y) (Windows)	Succeeded (Yes/No)
1	N/A	N/A	N/A	N/A	N/A	N/A	No
2	1.37E+06	9.07E-04	2.87E-11	1.37E+06	1.22E-03	3.87E-11	Yes
3	3.36E+06	2.23E-03	7.06E-11	3.36E+06	3.00E-03	9.50E-11	Yes
4	N/A	N/A	N/A	N/A	N/A	N/A	No
5	N/A	N/A	N/A	N/A	N/A	N/A	No
6	N/A	N/A	N/A	N/A	N/A	N/A	No
7	N/A	N/A	N/A	N/A	N/A	N/A	No
8	N/A	N/A	N/A	N/A	N/A	N/A	No
9	N/A	N/A	N/A	N/A	N/A	N/A	No
10	N/A	N/A	N/A	N/A	N/A	N/A	No
11	N/A	N/A	N/A	N/A	N/A	N/A	No

**Table 6-9: Time taken to complete a crack for specific passwords using two different operating systems with the same hardware using a combo list and John the Ripper (authors' own)**

ID	Hash Guesses (amt) (Linux)	Time to crack (s) (Linux)	Time to crack (y) (Linux)	Hash Guesses (amt) (Windows)	Time to crack (s) (Windows)	Time to crack (y) (Windows)	Succeeded (Yes/No)
1	N/A	N/A	N/A	N/A	N/A	N/A	No
2	1.37E+06	4.40E-05	1.40E-12	1.37E+06	6.03E-05	1.91E-12	Yes
3	3.36E+06	1.08E-04	3.43E-12	3.36E+06	1.48E-04	4.70E-12	Yes
4	N/A	N/A	N/A	N/A	N/A	N/A	No
5	N/A	N/A	N/A	N/A	N/A	N/A	No
6	N/A	N/A	N/A	N/A	N/A	N/A	No
7	N/A	N/A	N/A	N/A	N/A	N/A	No
8	N/A	N/A	N/A	N/A	N/A	N/A	No
9	N/A	N/A	N/A	N/A	N/A	N/A	No
10	N/A	N/A	N/A	N/A	N/A	N/A	No
11	N/A	N/A	N/A	N/A	N/A	N/A	No

**Table 6-10: Time taken to complete a crack for specific passwords using two different operating systems with the same hardware using a combo list and Hashcat (authors' own)**

## 6.4 Hybrid

Hashcat -a 6/7 added 2-digit masks to every word. This cracked IDs 2-4 in less than 2 seconds GPU time and even breached IDs 6-7 (10-char mixed strings) within around 1 minute on Linux, highlighting the power of small mask extensions (Tables 6-11 & 6-12).

ID	Hash Guesses (amt) (Linux)	Time to crack (s) (Linux)	Time to crack (y) (Linux)	Hash Guesses (amt) (Windows)	Time to crack (s) (Windows)	Time to crack (y) (Windows)	Succeeded (Yes/No)
1	N/A	N/A	N/A	N/A	N/A	N/A	No
2	1.25E+05	8.31E-05	2.64E-12	1.25E+05	1.12E-04	3.55E-12	Yes
3	1.37E+07	9.07E-03	2.87E-10	1.37E+07	1.22E-02	3.87E-10	Yes
4	6.27E+10	4.16E+01	1.32E-06	6.27E+10	5.59E+01	1.77E-06	Yes
5	N/A	N/A	N/A	N/A	N/A	N/A	No
6	2.01E+12	1.34E+03	4.23E-05	2.01E+12	1.80E+03	5.70E-05	Yes
7	2.27E+12	1.50E+03	4.76E-05	2.27E+12	2.02E+03	6.41E-05	Yes
8	N/A	N/A	N/A	N/A	N/A	N/A	No
9	N/A	N/A	N/A	N/A	N/A	N/A	No
10	N/A	N/A	N/A	N/A	N/A	N/A	No
11	N/A	N/A	N/A	N/A	N/A	N/A	No

**Table 6-11: Time taken to complete a crack for specific passwords using two different operating systems with the same hardware, using a hybrid attack with John the Ripper (authors' own)**

ID	Hash Guesses (amt) (Linux)	Time to crack (s) (Linux)	Time to crack (y) (Linux)	Hash Guesses (amt) (Windows)	Time to crack (s) (Windows)	Time to crack (y) (Windows)	Succeeded (Yes/No)
1	N/A	N/A	N/A	N/A	N/A	N/A	No
2	1.25E+05	4.04E-06	1.28E-13	1.25E+05	5.53E-06	1.75E-13	Yes
3	1.37E+07	4.40E-04	1.40E-11	1.37E+07	6.03E-04	1.91E-11	Yes
4	6.27E+10	2.02E+00	6.40E-08	6.27E+10	2.77E+00	8.77E-08	Yes
5	N/A	N/A	N/A	N/A	N/A	N/A	No
6	2.01E+12	6.49E+01	2.06E-06	2.01E+12	8.89E+01	2.82E-06	Yes
7	2.27E+12	7.30E+01	2.31E-06	2.27E+12	1.00E+02	3.17E-06	Yes
8	N/A	N/A	N/A	N/A	N/A	N/A	No
9	N/A	N/A	N/A	N/A	N/A	N/A	No
10	N/A	N/A	N/A	N/A	N/A	N/A	No
11	N/A	N/A	N/A	N/A	N/A	N/A	No

**Table 6-12: Time taken to complete a crack for specific passwords using two different operating systems with the same hardware, using a hybrid attack with Hashcat (authors' own)**

## 7 Discussion

This section regards the experimental findings, links them back to our research questions, and weighs security gains against practical usability. Then compare operating-system performance and explore how password composition policies influence both crack resistance and user memorability, translating raw numbers into actionable recommendations.

### 7.1 OS / Tool Impact

The Linux setup equipped with Hashcat processed MD5 roughly 1.4-1.6× faster than an identical Windows setup (Table 6-4). This confirms earlier driver-overhead claims (Hashcat Project, 2024) and rejects  $H_0$  that the host OS has no practical influence. John-the-Ripper showed a similar, though smaller, Linux advantage at around 1.3×, suggesting the delta stems from kernel-level GPU scheduling rather than tool implementation.

### 7.2 Method Effectiveness

Hybrid rules (dict+mask) cracked 6 of 11 dataset hashes, outperforming dictionary-only (2/11) and combo list (2/11) phases. This agrees with Tran et al.'s (2022) observation that small mask extensions exploit users' habit of tacking digits onto words. Brute force broke only the four password strings with fewer than 8-character characters.

Our experiment confirms Tran et al.'s (2022) survey prediction that hybrid rules dominate fast hashes: combining a core dictionary with a two-digit mask cracked every word-based secret under nine characters in under a second, whereas brute-force required minutes for the same length range.

Dictionary-only runs added little after the first few milliseconds, and combo lists (pre-tokenised pairs) offered only marginal advantage once the hybrid mask space was exhausted, again mirroring the rank order Tran et al. reported across 48 academic data sets.

### 7.3 Implications

For legacy systems still holding unsalted MD5, hybrid masks driven by SecLists can compromise nearly three-quarters of eight-to-ten-character user passwords in minutes on commodity GPUs. Migration to memory-hard KDFs plus unique salts is therefore urgent. Where migration is not yet possible, enforcing more than 10-character, mixed-class policies markedly raises brute-force cost (Table 6-5) and neutralises off-the-shelf combo lists.

Biryukov et al.'s (2021) RFC 9106 recommends Argon2id with at least 2 GiB RAM and one pass, raising the time–area cost for GPU attackers by roughly two orders of magnitude over SHA-512 + PBKDF2.

Saran's (2024) empirical study confirms a 95 % drop in GPU guesses-per-second under a 1 GiB, 3-pass Argon2id setting relative to an equal-latency SHA-512 configuration.

Where memory upgrades are impossible, bcrypt still beats PBKDF2; yet its 4 kB memory cap leaves it far more ASIC-friendly, so current industry guidance positions bcrypt as only an interim fix until Argon2id migration is feasible.

## 7.4 Password Usability

Our crack-rate data show that completely random, 12-plus-character MD5 strings resisted every offline attack within the allotted window, whereas human-selected words or word-plus-digits fell quickly. Yet the usability literature warns that users avoid high-entropy strings unless support tools are present. Controlled memorability trials found that 78 % of participants could still recall a self-chosen eight-character password after one week, but recall plummeted to 22 % for a random 12-character string without mnemonic aids (Ye et al., 2019).

Hartwig and Stobert (2022) showed that five-word pass-phrases strike a better security–memorability balance: crack-time soars, but 71 % of users reproduced the phrase error-free after two weeks. A hybrid policy, allowing multi-word phrases while banning dictionary singletons, would likely preserve user success rates while neutralising the rapid combo list and dictionary wins observed.

Abdrabou et al.'s (2022) laboratory work shows that gaze and keystroke dynamics can flag password reuse with up to 88 % accuracy, enabling just-in-time prompts before a weak credential is even submitted.

Coupling those behavioural signals with breach-monitoring services could dramatically shorten an attacker's offline window, complementing the technical mitigations discussed above.

## 7.5 In relation to previous research

The results align with prior studies (Touch, 1996; Hutter, 2015) demonstrating the weaknesses of legacy hash functions such as MD5 and the increased resilience provided by SHA-256, especially when combined with salting. The effectiveness of dictionary and hybrid attacks reinforces the findings of Hitaj et al. (2017) regarding the predictability of human-chosen passwords. This study extends previous work by quantifying the impact of modern mitigation strategies, such as rate limiting and the use of robust, unique salts.

Update Touch's (1995) hardware-oriented MD5 study with commodity GPU data, quantifying a modern Linux driver's 12 % throughput edge over Windows.

Our per-character-length crack-time curve matches the exponential trend reported by Shi et al.'s (2021) large-scale offline-cracking study: each extra symbol after the tenth increases mean crack time by roughly two orders of magnitude when the hash is fast.

Likewise, our finding that GPU brute force outpaces CPU John-the-Ripper by around 20× is within the 15 to 30× range Touch (1995) extrapolated when he compared early RISC workstations to bespoke hardware accelerators.

## 7.6 Ethical and societal aspects

Ethically, password-cracking research must balance the need to expose vulnerabilities with the responsibility to avoid enabling malicious actors. All experiments in this study were conducted on synthetic data in a controlled environment, adhering to responsible disclosure principles. Societally, the findings underscore the ongoing need for user education, better password policies, and the adoption of multi-factor authentication to protect against threats.

## 8 Conclusion

This thesis set out to measure how four well-known cracking strategies, brute force, dictionary, hybrid, and combo list, perform against MD5, the most frequently observed unsalted hash in historical breach data. Using a Proxmox-based experimental setup (Ryzen 5800X + RTX 3090), there were comparisons made of Hashcat and John the Ripper on both Ubuntu 22.04 and Windows 11 Pro, then analysed crack-throughput, time-to-success, and usability impact.

Our evidence reinforces the long-standing view that unsalted and salted, fast hashes such as MD5 are unfit for modern threat models. On a single RTX 3090, the 10.5 M-entry Xato combo list was checked against raw-MD5 in 0.13 seconds. This implies an attacker could iterate every credential in the RockYou corpus in under a second, meaning any breach of an unsalted MD5 database yields near-instant compromise for a large fraction of accounts. Neither longer passwords nor stricter complexity rules neutralise that risk; only slowing the hash or adding uniqueness (salts) does so effectively.

So, some recommendations according to our research would be the following:

1. Immediate mitigation for legacy MD5. If migrating off MD5 is not immediately possible, enforce a minimum 12-character pass-phrase policy, block the top dictionary and combo entries, and rate-limit authentication attempts. These steps raise the offline cost but are, at best, interim defences.
2. Transition to memory-hard KDFs. Plan to re-hash stored credentials with Argon2id, the current IETF recommendation, with cost parameters tuned for at least 100 ms on server hardware and more than 64 MB of memory. Argon2id reduces a GPU's advantage by several orders of magnitude, making large-scale guessing economically impractical.
3. Layered authentication, even robust hashing, cannot stop credential reuse or phishing. Enforce multi-factor authentication (for example, OTP, FIDO2 passkeys) for sensitive roles and high-risk logins.
4. User-centric tooling, Demand or strongly encourage password managers. Random, manager-generated strings neutralise dictionary-based guessing with minimal user memory cost, addressing the usability-security paradox highlighted in our discussion.

Passwords remain ubiquitous, but our results show that storage practice and supporting defences make the decisive difference. MD5 can no longer be justified: even moderate hardware cracks typical user secrets in seconds. Salts offer minimal complexity but dramatic protection, and memory-hard algorithms such as Argon2id further shift the cost curve in favour of defenders. When combined with password managers, multi-factor prompts, and, in the long term, passwordless authentication, organisations can meet both usability and security goals. The roadmap is clear: salt everything now, migrate to Argon2id (or an equally robust KDF) as soon as practical, and prepare for a future where the weakest link is no longer a fast, twenty-year-old hash algorithm but rather the organisation's willingness to retire it.

## 8.1 Future work

The study used an 11-row synthetic dataset for controlled comparability; a larger dataset would yield a greater success rate statistic. Hashcat and JtR were benchmarked on a single GPU and CPU; future work should evaluate distributed cracking clusters and emerging ASIC solutions (which can compute these algorithms up to 100x the rate).

Work could be done to allow organisations to make a smoother transition from old hash algorithms such as MD5, NTLM, and SHA, through a literature review to find the most optimal path forward.

Our data cover only salted MD5 and therefore cannot quantify exactly the benefit of moving to bcrypt, PBKDF2-HMAC-SHA-512, or Argon2id; nor do we model distributed cracking rigs. Extending the experimental design to those settings would enable a fuller cost–benefit analysis of migration paths that the literature increasingly recommends.

## References

- Abdrabou, Y., Schütte, J., Shams, A., Pfeuffer, K., Buschek, D., Khamis, M., & Alt, F. (2022). "Your Eyes Tell You Have Used This Password Before": Identifying Password Reuse from Gaze and Keystroke Dynamics. *Proceedings of the 2022 CHI Conference on Human Factors in Computing Systems*, 1–16.  
<https://doi.org/10.1145/3491102.3517531>
- Bao, Y., Zeng, J., Yang, J., Yang, R., & Lu, Z. (2024). The Effect of Domain Terms on Password Security. *ACM Trans. Priv. Secur.*, 28(1), 9:1-9:29.  
<https://doi.org/10.1145/3703350>
- Biryukov, A., Dinu, D., Khovratovich, D., & Josefsson, S. (2021). Argon2 Memory-Hard Function for Password Hashing and Proof-of-Work Applications (Request for Comments No. RFC 9106). Internet Engineering Task Force.  
<https://doi.org/10.17487/RFC9106>
- Bonneau, J., Herley, C., Oorschot, P. C. van, & Stajano, F. (2012). The Quest to Replace Passwords: A Framework for Comparative Evaluation of Web Authentication Schemes. *2012 IEEE Symposium on Security and Privacy*, 553–567.  
<https://doi.org/10.1109/SP.2012.44>
- Bonneau, J., Herley, C., van Oorschot, P. C., & Stajano, F. (2015). Passwords and the evolution of imperfect authentication. *Commun. ACM*, 58(7), 78–87.  
<https://doi.org/10.1145/2699390>
- Delicato, F. C., Pires, P. F., Rust, L., Pirmez, L., & de Rezende, J. F. (2005). Reflective middleware for wireless sensor networks. *Proceedings of the 2005 ACM Symposium on Applied Computing*, 1155–1159.  
<https://doi.org/10.1145/1066677.1066937>
- Florencio, D., & Herley, C. (2007). A large-scale study of web password habits. *Proceedings of the 16th International Conference on World Wide Web*, 657–666.  
<https://doi.org/10.1145/1242572.1242661>
- Guo, Y., Zhang, Z., & Guo, Y. (2019). Optiwords: A new password policy for creating memorable and strong passwords. *Computers & Security*, 85, 423–435.  
<https://doi.org/10.1016/j.cose.2019.05.015>
- Hallett, J., Patnaik, N., Shreeve, B., & Rashid, A. (2021). 'Do this! Do that!, And nothing will happen' Do specifications lead to securely stored passwords? (No. arXiv:2102.09790). arXiv.  
<https://doi.org/10.48550/arXiv.2102.09790>
- Hashcat Project. (2024). Hashcat (Version 6.2.7) [Software].  
<https://github.com/hashcat/hashcat>
- Hitaj, B., Gasti, P., Ateniese, G., & Perez-Cruz, F. (2019). Passgan: A deep learning approach for password guessing. *Applied Cryptography and Network Security: 17th International Conference, ACNS 2019, Bogota, Colombia, June 5–7, 2019, Proceedings 17*, 217–237.  
<https://doi.org/10.48550/arXiv.1709.00440>

- Hutter, M. (2016). Threshold Implementations in Industry: A Case Study on SHA-256. In Proceedings of the 2016 ACM Workshop on Theory of Implementation Security (p. 37). Association for Computing Machinery.  
<https://doi.org/10.1145/2996366.2996373>
- Leurent, G., & Peyrin, T. (2020). SHA-1 is a Shambles—First Chosen-Prefix Collision on SHA-1 and Application to the PGP Web of Trust (No. 2020/014). Cryptology ePrint Archive.  
<https://eprint.iacr.org/2020/014>
- Miessler, D. (2025). SecLists (Version 2025.05.01) [Password and word-list collection]. GitHub.  
<https://github.com/danielmiessler/SecLists>
- Milousi, K., Kiriakidis, P., Mengidis, N., Rizos, G., Mazi, M. S., Voulgaridis, A., Votis, K., & Tzovaras, D. (2024). Evaluating Cybersecurity Risk: A Comprehensive Comparison of Vulnerability Scoring Methodologies. In Proceedings of the 19th International Conference on Availability, Reliability and Security (p. Article 52). Association for Computing Machinery.  
<https://doi.org/10.1145/3664476.3670915>
- NIST Special Publication 800-63B. (n.d.). Retrieved 17 May 2025, from  
<https://pages.nist.gov/800-63-4/sp800-63b.html>
- Openwall. (2024). John the Ripper "Jumbo" (Version 1.9.0-jumbo-1-2024-04-22) [Software].  
<https://github.com/openwall/john>
- P. L & K. S. (2024). Exploiting AES Encryption Vulnerabilities Through Padding Oracle Attacks and Generative AI Techniques. 2024 8th International Conference on I-SMAC (IoT in Social, Mobile, Analytics and Cloud) (I-SMAC), 682–689.  
<https://doi.org/10.1109/I-SMAC61858.2024.10714697>
- Saran, A. N. (2024). On time-memory trade-offs for password hashing schemes. *Frontiers in Computer Science*, 6.  
<https://doi.org/10.3389/fcomp.2024.1368362>
- Shi, R., Zhou, Y., Li, Y., & Han, W. (2021). Understanding Offline Password-Cracking Methods: A Large-Scale Empirical Study. *Security and Communication Networks*, 2021(1), 5563884.  
<https://doi.org/10.1155/2021/5563884>
- Simon, J., Watson, S. J., & van Sintemaartensdijk, I. (2025). Response-efficacy messages produce stronger passwords than self-efficacy messages ... for now: A longitudinal experimental study of the efficacy of coping message types on password creation behaviour. *Computers in Human Behavior Reports*, 17, 100615.  
<https://doi.org/10.1016/j.chbr.2025.100615>
- Teat, C., & Peltsverger, S. (2011). The security of cryptographic hashes. In Proceedings of the 49th annual ACM Southeast Conference (pp. 103–108). Association for Computing Machinery.  
<https://doi.org/10.1145/2016039.2016072>

Touch, J. D. (1995). Performance analysis of MD5. *SIGCOMM Comput. Commun. Rev.*, 25(4), 77–86.

<https://doi.org/10.1145/217391.217414>

Tran, L., Nguyen, T., Seo, C., Kim, H., & Choi, D. (2022). A Survey on Password Guessing (No. arXiv:2212.08796). arXiv.

<https://doi.org/10.48550/arXiv.2212.08796>

Wang, X., & Yu, H. (2005). How to Break MD5 and Other Hash Functions. *Advances in Cryptology – EUROCRYPT 2005*, 19–35.

[https://doi.org/10.1007/11426639\\_2](https://doi.org/10.1007/11426639_2)

Wohlin, C., Runeson, P., Höst, M., Ohlsson, M. C., Regnell, B., & Wesslén, A. (2024). *Experimentation in Software Engineering*. Springer Berlin, Heidelberg.

<https://doi.org/10.1007/978-3-642-29044-2>

Wu, X., Munyendo, C. W., Cosic, E., Flynn, G. A., Legault, O., & Aviv, A. J. (2022). User Perceptions of Five-Word Passwords. *Proceedings of the 38th Annual Computer Security Applications Conference*, 605–618.

<https://doi.org/10.1145/3564625.3567981>

Ye, B., Guo, Y., Zhang, L., & Guo, X. (2019). An empirical study of mnemonic password creation tips. *Computers & Security*, 85, 41–50.

<https://doi.org/10.1016/j.cose.2019.04.009>

Zhu, C., Tang, Y., Wang, Q., & Li, M. (2019). Enhancing Code Similarity Analysis for Effective Vulnerability Detection. *Proceedings of the 2nd International Conference on Computer Science and Software Engineering*, 153–158.

<https://doi.org/10.1145/3339363.3339383>

Zou, Y., Le, K., Mayer, P., Acquisti, A., Aviv, A. J., & Schaub, F. (2024). Encouraging Users to Change Breached Passwords Using the Protection Motivation Theory. *ACM Trans. Comput.-Hum. Interact.*, 31(5), 63:1-63:45.

<https://doi.org/10.1145/3689432>

## Appendices

The following appendices provide supplementary materials referenced throughout the thesis.

### A. Average Speed Calculation

To calculate the average hash/s rate of the GPU, an average speed was calculated from a ten-hour-long cracking session on MD5 using this Python script.

```
import re

def parse_speed_line(line):
    # Regex to extract value and unit
    match = re.search(r'Speed\.#1\.*:\s+([\d\.]+)\s*([kMG]?H/s)', line)
    if not match:
        return None

    value = float(match.group(1))
    unit = match.group(2)

    # Convert all speeds to H/s
    if unit == 'kH/s':
        value *= 1_000
    elif unit == 'MH/s':
        value *= 1_000_000
    elif unit == 'GH/s':
        value *= 1_000_000_000

    return value

def extract_speeds_from_file(filepath):
    speeds = []
    with open(filepath, 'r', encoding='utf-8') as file:
        for line in file:
            if 'Speed.#1' in line:
                parsed_speed = parse_speed_line(line)
                if parsed_speed is not None:
                    speeds.append(parsed_speed)
    return speeds

def average_speed(speeds):
    if not speeds:
        return 0
    return sum(speeds) / len(speeds)

# Path to your Hashcat output file
filepath = 'output.txt'

# Extract and average speeds
speeds = extract_speeds_from_file(filepath)
average = average_speed(speeds)

print(f"Number of Speed.#1 entries: {len(speeds)}")
print(f"Average Speed: {average:.2f} H/s")
```

## B. Iterations Calculation

This Python script adds together all of the iterations completed as well as the exact time taken from start to finish.

```
import re
from datetime import datetime

def extract_times(filepath):
    start_time = None
    end_time = None

    with open(filepath, 'r', encoding='utf-8') as file:
        for line in file:
            if 'Time.Started' in line and start_time is None:
                match = re.search(r'Time.Started.*?: (.+?) \\(\\d+.*\\)', line)
                if match:
                    start_time = datetime.strptime(match.group(1).strip(), '%a %b %d %H:%M:%S
%Y')
            elif 'Time.Estimated' in line:
                match = re.search(r'Time.Estimated.*?: (.+?) \\(\\d+.*\\)', line)
                if match:
                    end_time = datetime.strptime(match.group(1).strip(), '%a %b %d %H:%M:%S %Y')

    return start_time, end_time

def extract_progress(line):
    match = re.search(r'Progress\.*:\s+(\\d+)/\\d+', line)
    if match:
        return int(match.group(1))
    return None

def analyze_hashcat_output(filepath):
    total_progress = 0
    progress_entries = 0

    with open(filepath, 'r', encoding='utf-8') as file:
        lines = file.readlines()

    for line in lines:
        if 'Progress.....:' in line:
            progress = extract_progress(line)
            if progress is not None:
                total_progress += progress
                progress_entries += 1

    start_time, end_time = extract_times(filepath)
    if start_time and end_time:
        duration_seconds = int((end_time - start_time).total_seconds())
    else:
        duration_seconds = None

    return total_progress, progress_entries, duration_seconds

# Set the path to your Hashcat output file
filepath = 'output.txt'

# Run the analysis
total_iterations, entries_found, duration = analyze_hashcat_output(filepath)

print(f"Number of progress entries: {entries_found}")
print(f"Total completed iterations: {total_iterations:,}")
if duration is not None:
    print(f"Total time elapsed: {duration} seconds ({duration / 60:.2f} minutes)")
else:
    print("Start or end time not found.")
```