BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF INFORMATION TECHNOLOGY
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF INFORMATION SYSTEMS
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

ANALYSIS OF HASHCAT PERFORMANCE UNDER DIFFERENT COMBINATIONS OF ATTACKS AND THEIR PARAMETERS

NAVRHNĚTE METODU MĚŘENÍ VÝKONNOSTI HASHCATU PRO RŮZNÉ KOMBINACE PARAMETRŮ JEDNOTLIVÝCH ÚTOKŮ

BACHELOR'S THESIS
BAKALÁŘSKÁ PRÁCE

AUTHOR **SOTIRIOS PUPAKIS**
AUTOR PRÁCE

SUPERVISOR **Ing. DANIEL DOLEJŠKA,**
VEDOUCÍ PRÁCE

BRNO 2023

BRNO UNIVERSITY OF TECHNOLOGY
FACULTY OF INFORMATION TECHNOLOGY

Bachelor's Thesis Assignment

149882

Institut:	Department of Information Systems (UIFS)
Student:	**Pupakis Sotirios**
Programme:	Information Technology
Specialization:	Information Technology
Title:	**Analysis of Hashcat Performance under Different Combinations of Attacks and Their Parameters**
Category:	Software analysis and testing
Academic year:	2022/23

Assignment:

1. Study the hashcat, its currently supported attacks and hash types. Analyze their implementation from available source code. Consider the latest available released version and respect the recommendations of the supervisor.
2. Propose a method to measure the performance of hashcat under different combinations of parameters of each attack as recommended by the supervisor. Propose a method for collecting metrics relevant to hashcat performance analysis. Propose a method for analyzing the dependencies of each parameter on the performance of each attack.
3. Perform the measurement and data collection as suggested in Task 2 and as recommended by the supervisor.
4. Analyze the measured data and iterate the measurement design as necessary to gain a better understanding of the dependencies of each parameter.
5. Discuss the results obtained and draw appropriate conclusions.

Literature:

- NETMUX (ed.), 2017. *Hash crack: password cracking manual*. V2.0. Herndon, Virginia: Netmux. ISBN 978-1-975924-58-4.
- HRANICKÝ, Radek, ZOBAL, Lukáš, RYŠAVÝ, Ondřej and KOLÁŘ, Dušan, 2019. Distributed password cracking with BOINC and hashcat. *Digital Investigation*. 1 September 2019. Vol. 30, p. 161–172. DOI 10.1016/j.diin.2019.08.001. V2.0. Herndon, Virginia: Netmux. ISBN 978-1-975924-58-4.
- hashcat wiki, [no date]. Online. [Accessed 27 October 2022]. Available from: https://hashcat.net/wiki/

Requirements for the semestral defence:
Points 1, 2 and partial development of point 3.

Detailed formal requirements can be found at https://www.fit.vut.cz/study/theses/

Supervisor:	**Dolejška Daniel, Ing.**
Head of Department:	Kolář Dušan, doc. Dr. Ing.
Beginning of work:	1.11.2022
Submission deadline:	31.7.2023
Approval date:	31.10.2022

Abstract

Hashcat is a program that is used to crack passwords. It allows the user to choose from different attack types, or to combine them in a combined attack. Combination of some of the attack vectors can result in degraded craking performance. The aim of this thesis is to meassure the performance with different attack combinations, find combinations that result in degraded performance and find out how much performance is not utilised in the worst case scenario against the best case scenario. This will allow us to better understand, how different combinations effect each other and allow us to avoid such cases in the future.

Abstrakt

Hashcat je program, který je použit pro lámání hesel. Lámání probíhá za použití různých způsobů útoku, které se také můžou kombinovat. Kombinace některých vektorů útoků může vyústit v zhoršenou výkonnost lámání. Cílem této práce je změřit, jak moc kombinace různých útoků ovlivňuje výkonnost lámání hesel, najít nejhorší možné kombinace a porovnat jejich výkon vůči nejlepším kombinacím. Toto nám do budoucna umožní lépe pochopit, jak různé útoky ovlivňují výkonnost lámání a také kterým kombinacím se vyhnout.

Keywords

hashcat, password cracking

Klíčová slova

hashcat, lámání hesel, měření výkonnosti

Reference

PUPAKIS, Sotirios. *Analysis of Hashcat Performance under Different Combinations of Attacks and Their Parameters.* Brno, 2023. Bachelor's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor Ing. Daniel Dolejška,

Rozšířený abstrakt

V dnešní době se veliká část našich životů odehrává na internetu, jsou tam uloženy naše vzpomínky, plány, ale i finance. Jelikož se na internetu může každý vydávat za kohokoliv a nelze vizuálně vždy ověřit, kdo je na druhé straně připojení, například při interakci s obline bankovnictvím, je potřeba zajistit autentifikaci jiným, než vizuálním způsobem.

Nejčastější způsob této autentifikace je znalost, či neznalost hesla, které slouží jako jediný klíč bránící naše data před lidmi, kteří by k nim neměli mít přístup. Hesla jsou nedílnou součástí našich životů a neobejdeme se bez nich jediný den. Tento systém avšak není perfektní, jelikož aby bylo heslo použitelné, musí ho znát obě strany, které se pomocí něj autentifikují. Internetové aplikace si ukládají data, mezi která patří i hesla, do databází, které slouží jako paměť pro aplikace.

V případě, že potřebuje aplikace autentifikovat uživatele, podívá se do své databáze, najde si heslo uživatele který se snaží autentifikovat a porovná je. Pokud se shodují, uživatel je tím, za koho se vydává. Pokud se neshodují, tak k autentifikaci nemůže dojít. Jelikož servery na kterých běží aplikace nejsou nehacknutelné, může dojít k jejich úniku. Servery také mají fyzické lokace, takže pokud útočník získá fyzický přístup k serveru, může si databázi stáhnout také bez problému.

Z tohoto důvodu se hesla neukládají v takové podobě, v jaké je zadává uživatel, ale dochází k jejich hashování. Hashování je proces, při kterém se text, v tomto případě heslo, změní k nepoznání. Hashovací funkce, neboli funkce které provádí hashování, mají vlastnost, že při stejném vstupu vždy dojdou k stejnému závěru. Další jejich vlastnost je, že je nelze jednoduše obrátit a z hashovaného hesla získat heslo samotné.

Jedna z možností, jak získat původní heslo, je zkoušet všechny možné kombinace textů, pusit je přes stejnou hashovací funkci a poté porovnou, zda-li se výsledné hashe rovnají. Pokud ano, útočník našel původní heslo a může se nyní autentifikovat jako uživatel, kterým není. Jedním z programů, který lze použít k tomuto závěru, je hashcat. Hashcat je otevřený, volně šířitelný program, který je vytvořen právě za účelem zjišťování původních hesel.

Tato bakalářská práce se zaměřuje na výkonovou analýzu programu hashcat, za použití různých hashovacích algoritmů a různých parametrů. Získávání originálních hesel z hashů je dlouho trvající operace, jelikož se musí zkusit veliké množství všech možných kombinací písmen, z kterých se heslo může skládat. S vývojem nového a rychlejšího hardwaru, se doba potřebná k prolomení hesel zkracuje z let na dny

Z tohoto důvodu vznikají nové, moderjnější hashovací algortimy, které jsou navrženy taky, aby trvaly dlouho vypočítat a byly výpočetně náročné. Hashcat tyto nové hashovací funknce podporuje také. Tato bakalářská práce nám umožní do budoucna lépe pochopit, jak různé spouštěcí parametry ovlivní výkon při získávání původních hesel.

Závěr této práce lze shrnout tak, že výkon hashcatu závisí na mnoha parametrech. Tyto parametry se liší jak hashovací algortimus od hashovacího algoritmu, tak typ útoku od typu útoku. Největší ztráty výkonu nastávájí na rychle běžících hashovacích funkcích, jako MD5, při použití slovníkového útoku. Rozdíly výkonosti také záleží na základní rychlosti hashovacího algoritmu. Rozdíly jsou zjevnější u algortimu jako MD5, který běží v rámci gigahashů za sekundu, zamtíco u algortimu bscrypt, bežící ve stovkách hashů za sekundu, jsou rozdíly téměř zanedbatelné.

Analysis of Hashcat Performance under Different Combinations of Attacks and Their Parameters

Declaration

I hereby declare that this Bachelor's thesis was prepared as an original work by the author under the supervision of Mr. Ing. Daniel Dolejška I have listed all the literary sources, publications and other sources, which were used during the preparation of this thesis.

.....
Sotirios Pupakis
July 31, 2023

Acknowledgements

I would like to express my thanks to my family and friends, who have been supporting me and cheering me on.

Contents

List of Figures

Chapter 1

Introduction

The increasing prevalence of password-based security measures necessitates a robust understanding of their vulnerabilities and effectiveness. In this digital age, where data breaches and unauthorised access pose significant threats, it is crucial to evaluate the strength of passwords and the efficiency of password-cracking tools. Hashcat, a widely utilised password-cracking tool, has gained prominence for its ability to leverage the computational power of GPUs to expedite the password recovery process.

This thesis explores the intricate relationship between Hashcat's performance, specifically its hash speed, and the combination of various parameters. By analyzing the impact of parameter selection on the speed at which Hashcat can crack passwords, we aim to shed light on optimizing its performance for more effective password security assessments and penetration testing.

The objective of this research is twofold: firstly, to investigate how different parameter combinations influence the hash speed of Hashcat, and secondly, to provide insights into optimizing Hashcat's configuration to enhance its performance in password recovery tasks. Understanding the impact of parameters such as hash type, attack mode, hashcat mode, workload profile, and hardware configuration is essential for practitioners and researchers involved in password security analysis.

To achieve these goals, we employ a systematic approach involving a series of experiments using diverse benchmark datasets and password hashes. Through statistical analysis and performance metrics, we measure the impact of parameter combinations on cracking speed.

The findings of this research will contribute to the existing knowledge base surrounding Hashcat's performance measurement. By identifying optimal parameter combinations and exploring the trade-offs between hash speed and other factors, such as accuracy and energy consumption, this thesis aims to provide practical recommendations for configuring Hashcat in different scenarios.

Furthermore, the outcomes of this research will have implications for enhancing password security practices and strengthening the resilience of password-based authentication systems. By uncovering the factors that impact Hashcat's hash speed, security professionals can make informed decisions when selecting parameters and configuring the tool for password-cracking operations.

Chapter 2

Hashes

In this chapter, the basics of hashes, hash functions, and their real-world uses will be explained. A salt and why is it important in the cybersecurity sphere will be explained. The basics of password cracking and the program hashcat will be explained.

2.1 Hash functions

In the context of computer science and cryptography, a hash refers to a mathematical function that takes an input (or „message“) and produces a fixed-size string of characters, which is typically a sequence of alphanumeric characters [11]. This output is commonly referred to as the hash value, hash code, or simply a hash.

Hash functions have many different use cases, and not all hash functions are appropriate for all of them. Some of the examples where hash functions are used are data integrity verification, digital signatures and data structures. There are a number of hash functions, but not all of them are suitable for all applications. With the advancements in technology and computing power, message digest algorithm 5 (MD5) is no longer considered secure. For this reason Secured Hashing Algorithm (SHA) family has been introduced. Reversing hashes created by these functions takes unreasonably long and is not suitable for brute-force attacks.

Hash functions have these characteristics:

- deterministic

  for a given input, the hash function will always produce the same hash value
- irreversible

  it is computationally infeasible to retrieve the original input from the hash
- small change in the input produces significant changes in the resulting hash value
- different inputs are highly likely to produce different hash values

2.2 Introduction to hashes

Hashes are outputs of hash functions. These are unique fixed-size strings of characters that represent a large set of data. The uniqueness and reproducibility of hashes make them ideal for a variety of purposes, such as data integrity checks and cryptographic applications.

For data integrity checks, a slight change in the original data will produce a significantly different hash. This makes hashes excellent for detecting modifications in data. For example, if you download a file from the internet, it may come with a hash value. By hashing the downloaded file and comparing it with the provided hash, you can check whether the file has been tampered with during transmission.

Lastly, in cryptographic applications, hashes are fundamental components. They are used in the creation of digital signatures [5], password security, and more. When you create a password for an online account, rather than storing the actual password, the system will often store the hash of the password. This way, even if the database is compromised, the attackers can't learn your password from the hash due to the „one-way" nature of the hash functions.

2.3 Passwords

When you create an account on a website or an application, you typically choose a password that's used to authenticate your identity whenever you log in. But, storing these passwords in plain text within the system's database would be a serious security risk. If the database was compromised, an attacker could easily steal all the passwords.

To mitigate this risk, systems often store not the passwords themselves, but hashes of these passwords. The hash function transforms your password into a unique string of characters, known as a hash, which is what the system actually stores.

When you attempt to log in, the password you enter is hashed again, and this hash is compared to the one stored in the system. If the hashes match, the system knows you've entered the correct password and lets you in. This process works because hash functions are deterministic, meaning that the same input will always produce the same output.

Even if a hacker were able to access the hashed passwords, they can't easily convert them back into the original password because hash functions are „one-way". That is, while it's straightforward to create a hash from an input, it's computationally impractical to perform the process in reverse to get the original input.

2.4 Salt

A „salt" in the context of password hashing is a random piece of data that is generated and added to each password before it's hashed [8]. This salt is then stored alongside the hashed password in the database.

The purpose of a salt is to defend against certain types of attacks on hashed passwords, such as the use of rainbow tables. A rainbow table is a precomputed table of hashes for common or possible password combinations. If an attacker gets access to a database full of hashed (but not salted) passwords, they can use a rainbow table to look up the plaintext passwords that correspond to the hashes.

However, when a unique salt is added to each password before it's hashed, the resulting hash will be different than the hash of the original password, even if two users have the same password. The salt essentially adds randomness to the password, making precomputed tables like rainbow tables ineffective unless the attacker also has the salt and has precomputed tables for all possible salts (which is generally computationally infeasible due to the large number of possible salts).

Moreover, the use of a unique salt for each user's password hash also ensures that even if two users have the same password, their hashes will be different. This helps to protect against another type of attack where someone could infer that users have the same password because their hashes are the same.

When a user tries to log in, the system will look up the salt associated with that user, add it to the entered password, hash the result, and compare it to the stored hash.

It's important to note that while salting adds an extra layer of security to password hashing, it does not substitute for using a secure, computationally intensive hash function, or for users choosing strong, unique passwords.

2.5 Cracking hashes

Cracking hashes often referred to as hash cracking, is a process where an attacker attempts to determine the original input given a hash output. Because hash functions are one-way, meaning they are designed to make it computationally difficult to get the original input from the hash, attackers typically have to resort to methods such as brute-force attacks, dictionary attacks, or rainbow table attacks.

- brute-force attack

  In a brute-force attack, the attacker simply tries all possible combinations of inputs until they find one that produces the same hash as the target hash. This can be a very time-consuming process, especially if the original input is long or complex. However, this method is guaranteed to eventually find the input if given enough time and computational resources.

- dictionary attack

  A dictionary attack is a bit more sophisticated. Instead of trying all possible combinations of inputs, the attacker tries a list of likely inputs. This list, or 'dictionary', typically includes common passwords and phrases. If the user's password is in the dictionary, this method can find it much faster than a brute-force attack. However, it won't find the password if it's not in the dictionary.

- rainbow table attack

  A rainbow table is a precomputed table of hash outputs and their corresponding inputs. It's essentially a huge lookup table that lets the attacker find the input for a given hash, provided the input is in the table. The downside of this method is that generating the table requires a significant amount of time and storage space, and it's ineffective if the password has been hashed with a unique salt.

- mask attack

  A mask attack is called so because it involves constructing a „mask“ that specifies the format of the passwords to be tested. The mask consists of placeholders for characters or character sets that might be used in the password. The placeholders are combined with brute-force attempts to generate all possible password combinations within the specified format.

- combinator attack

  A combinator attack is a combination of two dictionaries, where each word from a dictionary is appended to each word in another dictionary

- hybrid attack

  A hybrid attack is a combination of multiple password cracking techniques used in sequence to increase the efficiency of cracking hashed passwords. It aims to exploit the strengths of different attack methods while minimising their weaknesses, providing a more comprehensive approach to breaking passwords compared to using a single method alone. The two primary password-cracking techniques that are commonly combined in a hybrid attack are dictionary attack and brute-force attack

- rule based attack

  A rule-based attack is a password cracking technique that involves applying specific transformation rules to words in a wordlist or characters in a brute-force attempt to generate variations of potential passwords. The purpose of rule-based attacks is to increase the chances of cracking hashed passwords by exploring common password patterns and behaviours.

2.6 Hashcat

Hashcat[1] is a powerful and widely used password-cracking tool designed to recover lost or forgotten passwords. It leverages the computing power of modern GPUs (Graphics Processing Units) to perform high-speed password recovery by employing various attack techniques, including brute-force attacks, dictionary attacks, combinator attacks, and hybrid attacks.

Hashcat's primary function is to crack password hashes stored by computer systems. It supports a wide range of hash types commonly used in different applications, including popular hashing algorithms like MD5, SHA1, SHA256, bcrypt, and more. By comparing computed hash values with the target hash, Hashcat attempts to find the original password that generated the hash.

One of the key advantages of Hashcat is its ability to perform parallel processing on GPUs, allowing it to perform password-cracking operations significantly faster than traditional CPU-based methods. This makes Hashcat a valuable tool for security professionals, penetration testers, and researchers who need to assess the strength of password-based security measures.

Hashcat provides a flexible and customisable environment, allowing users to define attack modes, set specific rules for password generation, and configure various parameters to optimise performance. Additionally, it supports distributed computing across multiple systems, enabling the scalability required for tackling large-scale password-cracking tasks.

Hashcat is free, open-sourced, actively developed and supports the newest hardware and hashing functions. Runs on multiple operating systems, and multiple platforms (CPUs, GPUs, APUs). Supports cracking multiple hashes at the same time on multiple devices and multiple devices connected through a network. Has support for sessions, so in case of an interruption, will continue where the cracking stopped. Hashcat also includes the world's first and only in-kernel rule engine. All these qualities make hashcat the most popular password recovery program.

Hashcat offers number of parameters that can be tweaked and customised to achieve the ideal cracking speed. This thesis will focus on how these parameters affect the cracking performance.

[1] https://hashcat.net/

Optimised kernel

Hashcat provides an option to run with optimised kernels. This feature was implemented in hashcat version 4.0.0[2]. These optimised kernels provide faster cracking speeds, but have a disadvantage, that being the length of passwords that are supported has been reduced. For example, the maximum password length for the MD5 hash algorithm in the unoptimised kernel is 256 characters, while using the optimised kernel with the parameter `-O`, the maximum password length is reduced to just only 55 characters.

During testing and analysis, both unoptimised and optimised kernels must be tested.

Markov threshold

Markov threshold is an option in hashcat that will stop hashcat from accepting new Markov chains. Markov chains are an option that can significantly improve the efficiency of brute-force attacks.

A Markov chain is a statistical model that predicts the probability of certain events based on the state of a previous event [7]. In the context of password cracking, Hashcat uses Markov chains to predict the likelihood of certain characters following other characters in a password. This allows it to prioritize its attempts in a more intelligent and efficient way.

The Markov threshold defines the maximum level of complexity that will be used in the Markov chains when predicting which password guesses are most likely to be successful. This is an integer value and can be set using the `-markov-threshold` or `-t` command-line option.

The lower the threshold, the less complexity is allowed in the Markov chains, which can speed up the password-cracking process but may also reduce the chance of successfully cracking up the passwords. On the other hand, a higher threshold allowed for more complexity in the Markov chains, which can increase the chances of successfully cracking a complex password, but at the cost of a slower process.

It's important to note that the use of a Markov chain in password cracking isn't a guarantee of success. It's just a method of prioritising the password-cracking process based on statistical likelihood, rather than simply trying every possible combination in sequence.

Hashcat stores it's Markov table in a file called `hashcat.hcstat`, which can be generated based on a wordlist using the `hcstat2gen`[3]

2.7 Ethical considerations

While conducting research in the field of cybersecurity, it is of utmost importance to uphold ethical standards to ensure the integrity of the work and respect for privacy and ownership of data.

2.7.1 Test data selection

All passwords and hashes used in this research were created explicitly for this study, either randomly generated or from dummy user accounts. Under no circumstances were real-world passwords or hashes associated with real users. This ensures the privacy and security of any third-party data.

[2] https://hashcat.net/forum/thread-6965.html
[3] https://hashcat.net/wiki/doku.php?id=hashcat_utils#hcstat2gen

2.7.2 Resource utilisation

The tests were performed on dedicated hardware resources specifically allocated for this research. I've ensured not to overload shared resources, which might affect other users or services. Additionally, I was careful to monitor the hardware to prevent any potential damage due to excessive utilization.

2.7.3 Enviromental impact

Running intensive computational processes, such as those employed by hashcat, can have a significant energy footprint. To mitigate this, we attempted to run out test efficiently, reducing redundant tests and ensuring the system entered a low-power state when not in use.

2.7.4 Knowledge application

This research is intended to advance our understanding of password-cracking methods and their performance, primarily for the purpose of improving security measures and understanding potential threats. It's not intended to endorse or support malicious activities. I strongly discourage the use of the knowledge gained from this study for any unethical or illegal activities

Chapter 3

Data gathering

Until this point, we have covered the basics of password cryptography and its cracking. In this chapter, we look at how to obtain important data that will be used in the analysis. When gathering data, it's important to keep in mind, that there is a number of factors that can affect the data like programs running in the background or the temperature of the room the hardware is in.

3.1 Requirements

Obtaining data is a long process, due to the nature of the analysis, all data must be generated using the same hardware and software. A fresh install of Linux is used as a host for a Docker container. As hardware that will be used, an Nvidia RTX 3080 has been chosen mainly due to its high performance and large memory - 10GB of GDDR6X.

3.2 Testing environment

Creating a testing environment for Hashcat using Docker can help ensure a consistent and controlled setting for evaluating its performance. It should be noted, however, that because Docker operates within a container, there can be challenges and limitations when it comes to accessing the full potential of the host's GPU. Docker does provide some GPU support, but the level of support and setup steps can depend on your specific system configuration and the Docker version.

For this reason, an Nvidia CUDA-enabled Ubuntu image has been chosen as the base image, upon which this Docker image is based. This will ensure that Hashcat running in the Docker container can access the full power of the GPU. This Image then also includes instructions to install all necessary dependencies for Hashcat and automation tools. Docker enables the testing environment to be deployed on any system, which allows other researchers to replicate the findings of this thesis easily.

For versioning and change tracking, Git was chosen. Git is an open-source modern version control system developed in 2005 by Linus Torvalds. Git allows to work in branches, that can be easily merged. This allows for work on different parts of the codebase to be made without affecting each other until the merge, where all conflicts can be resolved.

3.3 Data

Analysing the performance of Hashcat (or any password-cracking tool) typically involves gathering and studying the following types of data:

- speed of execution

  perhaps the most critical performance metric. It's measured in hashes per second and indicates how many attempts Hashcat can make per unit of time

- resource utilisation

  it's important to measure how much computational resources Hashcat is using. Low resource utilisation hints at bottlenecks somewhere else in the pipeline, be it CPU or memory.

- algorithmic efficiency

  testing different algorithms and different configurations of Hashcat, we can measure how efficiently each one uses computational resources.

3.3.1 Methodology of capturing different hash types

One of the critical parameters in this study is the variety of hash algorithms used to evaluate Hashcat's performance. Different hash functions have different levels of computational complexity, which can significantly impact the time and resources required to crack them. To fully understand Hashcat's capabilities and performance, we need to consider a range of hash types, such as MD5, SHA-1, SHA-256, Whirlwind, and others.

The performance data for each hash type will be collected in a similar manner. For each hash type, a set of passwords will be hashed and then attempted to be cracked using hashcat. The time the attack will take, resource usage, hash speed and hardware status will be recorded for each attack on each hash type. This data will be analyzed to determine the effect of hash type on hashcat's performance.

One of the problems we can encounter is that the CPU is not capable of providing passwords fast enough for the GPU to crack. This can lead to degraded performance, as GPU hashing power is unused. The best example of such a problem is using a dictionary attack with the MD5 algorithm, where the utilisation of the GPU is only around `30%`.

3.3.2 Methodology of capturing different attack types

Hashcat employs a number of attack modes, such as brute-force attack, dictionary attack, mask attack, hybrid attack and rule-based attack. Each attack type has unique characteristics and potential use cases, and thy may perform differently under different conditions. Therefore, it's crucial to examine each attack type's performance across the different hash types mentioned earlier.

For each attack type, we will use a consistent methodology. The same set of passwords will be hashed with each hashed type and then cracked using each attack type. The time the attack will take, resource usage, hash speed and hardware status will be recorded for each attack and each hash type. This data will be analyzed to determine the effectiveness and efficiency of each attack type on each hash type.

3.3.3 Hash type selection

In the realm of cryptographic hash functions, a variety of hash types are commonly used, each with unique characteristics and vulnerabilities. To provide a comprehensive understanding of hashcat's performance across different hashing algorithms, I've selected a broad range of hash types for the tests.

MD5

Despite its known vulnerabilities and reduced use in secure environments, MD5 remain widely used in various applications, it serves as a base case due to its relatively low computational complexity.

SHA-1

Similar to MD5, SHA-1 is less commonly used in secure environments due to known vulnerabilities but is still prevalent in many systems.

SHA-256

Part of the SHA-2 family, SHA-256 is commonly used in secure environments. It presents a higher computational challenge than MD5 or SHA-1.

Bcrypt

Bcrypt is designed to be computationally intensive and slow, it resists brute-force attacks. Its inclusion provides insight into hashcat's on more secure, modern hash types.

3.4 Automation

Data gathering automation is crucial for thy type of thesis for several reasons

- reproducibility

  automated data collection makes the experiment reproducible. It ensures that the test conditions can be repeated precisely, which is essential for scientific accuracy

- volume of data

  the sheer amount of data that will be generated from testing different hash types and attack modes multiple times is massive. Manual data collection would be too time-consuming and prone to human error

- consistency

  automation allows for consistency across tests. For example, the time taken to run each test, the type of attacks launched, and the hash types tested can be kept consistent, eliminating any potential variation in the data due to inconsistency in testing procedures

- efficiency

  automated data collection saves time. Instead of manually running each test and recording the results, the process can run on its own

- data accuracy

  automated systems reduce the chance of human error, leading to more accurate data. This is particularly important when dealing with large volumes of data, as small errors can significantly affect the overall results

- ease of analysis

  automated systems offer data output in a format that is easy to import into statistical software for further analysis. This can simplify the data analysis process and reduce the potential for errors in data transcription

3.4.1 Automated data collection process

To ensure consistency, efficiency and reproducibility in the testing procedure, I have implemented a Python script to automate the process of running hashcat under various configurations and collecting performance data.

The Python script executes hashcat using Python's `subprocess module`, specifically the `subprocess.run` function, which allows the script to start a new process, run the given command (in this case hashcat), and then wait for the process to complete or kill it in case the specified timeout has expired.

The script invokes hashcat numerous times with different configurations, varying the hash types and attack modes, and adjusting specific parameters as defined in the testing framework outlined earlier.

For each hashcat invocation, we use the `-status` parameter, which forces hashcat to print out a status line to the standard output. Combining `-status` parameter with `-status-timer 1` and `-status-json`, hashcat will output a `json` formatted line every second to the standard output.

Once the process is either finished or killed by the `subprocess` module, as the timeout has expired, this output is then printed to the console to confirm that a run has been finished, as well as write the process standard output into a file created specifically for this run. This output is then parsed and prepared for analysis.

3.4.2 Parsing and preparing data for analysis

Given the nature of hashcat's output, it's essential to parse and filter the data to extract the relevant information for our performance analysis. Our Python script processes the output line by line to identify lines that are relevant to this study.

Hashcat outputs a significant amount of data that is not relevant to the analysis, mixed with JSON lines that include important performance metrics. To focus on the data relevant to the study, the script only parses lines that begin with the „{“ character. This character denotes the start of a JSON object and thus indicated a line of interest in the hashcat's output.

When the script identifies a line starting with „{“, it treats it as a JSON object. These JSON objects are then written into a new file, each line representing a JSON object and thus one second hashcat has been running. This process results in a clean, organised dataset of JSON lines, each representing a separate second hashcat run with its corresponding performance metrics. This dataset simplifies further analysis by reducing noise and focusing exclusively on the data of interest.

3.4.3 Error handling and logging

As the subject of this thesis is the performance, almost all hashcat sessions are designed to run longer than the timeout set in the `subrocess.run` function is set to. This results in almost all sessions returning a non-zero return code and throwing the `subprocess.TimeoutExpired` exception. This is indeed the desired outcome and does not require any special error handling.

In the case, that the sub-process ended normally, two possibilities could arise. Either hashcat managed to find the password or already finished all possible hashes and ran out of hashes to try. The first case may happen whenever, and it's important to monitor if this happens and edit the hash being crack accordingly. The second case may occur when a dictionary attack consists of a small dictionary, or when performing a brute-force attack with a short mask, for example, an eight-character long mask consisting of lowercase letters. a

3.5 Visualisation

Once the necessary data is collected and parsed, I utilise Python's Matplotlib library to generate informative graphs that visually represent the performance metrics captured during the experiments. Plotting data like hash speed, hardware utilisation, hardware temperature, and estimated time to finish the cracking process will allow for easy comparison under different run configurations.

This process can't be automated, as each graph needs to consist of the import data for the given use case. Some graphs require multiple hash speeds and utilisation lines, while others might require the estimated time of the attack to finish and GPU temperatures.

3.6 Data validation

Ensuring the accuracy and reliability of the data is of paramount importance in this study. Therefore, there are several measures of data validation instituted to ensure the integrity of the results.

Through these measures, I've sought to ensure the accuracy and reliability of the data and findings, thereby providing a solid foundation for the analysis and conclusions.

Output verification

After each run of hashcat, I verified that the tool's output was complete and in the expected format before parsing it. I've checked for common error indicators and verified the presence of key data points in the output. In cases of discrepancies or errors, the test run was repeated to ensure accurate data collection.

Parsing verification

Once the output was parsed into JSON and relevant data was extracted, I performed additional checks on the parsed data. These checks included verifying that the extracted data was within expected ranges, and cross-checking the data against known benchmarks when possible.

Cross-validation

To further ensure data reliability, I've conducted cross-validation on the data. Multiple runs of the same test case were performed and compared for results consistency. Any significant discrepancies in the results were further investigated to identify potential sources of error and rerun.

Post-analysis checks

After the completion of the data analysis, I performed additional checks to validate the findings. These included sanity checks, such as ensuring that bcrypt hashes took longer to crack than MD5 hashes, which is a known property of these hashing algorithms.

Chapter 4

Analysis

The data gathered during the testing provides the foundation for this analysis. The objective of this analysis is to interpret the performance metrics of Hashcat under different configurations, yielding insights into its strengths, weaknesses, and potential areas for optimisation. In this chapter, we examine the data collected from the experimentation and aim to answer the research question: „How does the hash type and attack type affect hashcat's performance? What implication do these findings have for the application of hashcat in real-world scenarios?“

4.1 Analysis approach

The analysis approach centres on the comparison of performance metrics across the various test cases. We examine the hash speed and GPU utilisation for each combination of hash type and attack type, identifying trends and outliers. The goal is to determine how much changes in these variables correlate with hashcat's performance.

4.2 Algorithms

In this section, a deeper dive into the selected test algorithms is provided. Their history and the reason why they were selected for this thesis. Algorithms were mainly chosen in such a way, that they provide in sign into old, fast-cracking algorithms through the more modern, like Secured Hash Algorithm to modern and currently used hashing algorithms like bcrypt.

4.2.1 MD5

The Message Digest Algorithm 5 (MD5) is a widely used cryptographic hash function that produces a 128-bit (16-byte) hash value from input data. It is commonly utilised to verify data integrity. MD5 has been utilised in a vast array of security applications and is also commonly used to check the integrity of files. However, the algorithm is not deemed secure against further advancements in computational capabilities, leading to a phase-out in favour of more secure hashing algorithms such as SHA-256.

History

MD5 was designed by Ronald Rivest in 1991 as an improvement and successor to earlier hash functions, MD4 and MD2. Rivest was a Professor at MIT and one of the founding members of RSA Data Security, a company known for its work on various cryptographic algorithms. The MD series of hash functions was named after the company, with MD standing for „Message Digest“.

The MD5 algorithm works by taking an input (or 'message') and then follows a series of steps to generate a fixed-size output of 128 bits. It involves padding the input message to ensure that its length is a multiple of 512 bits, appending the original length of the message, and then dividing the message into 512-bit blocks. Each of these blocks is then processed in four rounds, each consisting of 16 operations based on non-linear functions, modular addition, and bitwise operations. [10]

Upon its inception, the MD5 hash function was widely accepted in the cryptographic community and found myriad uses across the digital spectrum, from password storage mechanisms to data integrity checks. Its capacity to generate a „fingerprint“ or „digest“ from input data of arbitrary length made it a valuable tool in data security.

However, as technology and computational power increased, vulnerabilities in the MD5 algorithm began to surface. The first signs of trouble came in 1996, just five years after its creation, when cryptanalyst Hans Dobbertin reported a collision within the compression function of MD5 [4]. A collision occurs when two different input values produce the same hash output, effectively breaking a secure cryptographic hash function's fundamental trait: uniqueness.

The collision issues were initially perceived as theoretical, but as computational power increased and more advanced methods of collision searching were developed, practical collisions were found. By 2005, security researchers demonstrated the ability to generate different documents with the same MD5 hash, a situation that posed significant security risks. These developments led to a gradual phasing out of MD5 in favour of more secure algorithms. In 2004, The IETF[1] recommended against using MD5 for TLS[2] and IKE[3], two key Internet security protocols.

While SHA-1 is no longer suitable for functions requiring collision resistance, such as SSL certificates or digital signatures that rely on the certificate, it can still be used for non-cryptographic purposes, such as checksumming, where collision resistance is not required.

SHA-1's vulnerabilities and subsequent deprecation led to the development of more secure hash functions, including SHA-256 and SHA-3. These successor algorithms were designed to improve upon the weaknesses of SHA-1 while providing a greater degree of security and resistance against collision attacks.

In summary, the trajectory of SHA-1 from its widespread adoption to its eventual phase-out provides a valuable case study in the evolution of cryptographic hash functions. It underscores the persistent struggle between the ever-advancing forces of both cryptography and computational power. Today, understanding the design, application, and vulnerabilities of SHA-1 provides an essential foundation for the study of cryptographic hashing and the development of future, more secure algorithms.

[1] Internet Engineering Task Force
[2] Transport Layer Security
[3] Internet Key Exchange

Present day

As of today, while MD5 is still used in certain non-security critical areas due to its speed and ease of implementation, it is considered broken in terms of cryptographic security. In the security-conscious landscape of the digital age, more secure cryptographic hash functions such as SHA-256, part of the SHA-2 family, have been developed and widely adopted. These functions continue to build on the lessons learned from the historical development and vulnerabilities of earlier hash functions like MD5.

In conclusion, the MD5 hash function is a pivotal milestone in the history of cryptographic hash functions. Its design, widespread adoption, and subsequent vulnerabilities have significantly contributed to the evolution of cryptographic techniques. While its use is no longer recommended for securing sensitive data, the study of its design and inherent flaws provides invaluable insights into the development of more secure hash functions and cryptographic protocols.

Given its historic importance and speeds this hashing function is being cracked with modern hardware, it is one of the hashing functions of interest for this thesis.

4.2.2 SHA-1

The Secure Hash Algorithm (SHA-1) is a cryptographic hash function that produces a 160-bit (20-byte) hash value from input data. It was developed by the NSA[4] and published by the NITS[5] in 1995 as a part of the Digital Signature Algorithm which is one of the Federal Information Processing Standards.

SHA-1 emerged as a revision of the original SHA standard[6], which was published a year earlier in 1993. SHA-0 was withdrawn due to discovered „unspecified weaknesses“, which were rectified in SHA-1. The design of SHA-1 was inspired by the earlier MD4 and MD5 algorithms developed by Ronald Rivest. [1]

The first significant theoretical weaknesses in SHA-1 were reported by cryptanalysts in 2005 [15], and practical collisions were found by 2017 [14]. A collision occurs when two distinct pieces of data produce the same hash output, thereby violating the unique hash principle that is crucial for data integrity and security in digital communications and storage. The ability to generate such collisions indicates a vulnerability in the hash function.

4.2.3 SHA2-256

The Secure Hash Algorithm 2 (SHA-2) is a set of cryptographic hash functions designed by the National Security Agency (NSA) and first published by the National Institute of Standards and Technology (NIST) in 2001. SHA-2 includes several distinct hash functions with digest sizes of 224, 256, 384, 512, 512/224, and 512/256 bits. [6] Among these variants, SHA-256 is the most commonly used, offering an optimal balance of security and performance for most applications.

SHA-2, and particularly SHA-256, emerged as successors to the SHA-1 and MD5 algorithms, both of which demonstrated vulnerability to collision attacks, thus jeopardising their intended applications in digital signatures and certificates. SHA-256 was specifically designed to improve upon these weaknesses while providing a greater level of security.

[4] National Security Agency
[5] National Institute of Standards and Technology
[6] SHA-0

SHA-256 operates in a similar fashion to its predecessors and counterparts. It starts by padding the input message to achieve a length that's a multiple of 512 bits. Next, it appends the length of the original message and then breaks down the message into 512-bit blocks. Each block is subsequently processed in a series of complex transformations that involve bitwise operations, logical functions, and modular arithmetic. The final output is a 256-bit hash value, unique to each unique input. [3]

The result is a function that provides a high level of security, rendering it practically impossible for two different inputs to result in the same hash output, thereby avoiding collisions. Moreover, the complexity of the operations within SHA-256 makes it resistant to known cryptographic vulnerabilities, providing robust protection against threats such as preimage and second preimage attacks.[2]

In the years following its inception, SHA-256 has found widespread adoption in a variety of applications, particularly those requiring high degrees of security. It has become the algorithm of choice for SSL and TLS certificate signatures, Blockchain technology, and more. Many cryptographic libraries and standards, including OpenSSL and IPsec, have transitioned from using MD5 and SHA-1 to SHA-256.

However, while SHA-256 remains secure as of the date of writing, it's important to understand that no cryptographic function can be considered secure indefinitely. Technological advancements, specifically the advent of quantum computing, pose potential threats to SHA-256 and similar cryptographic hash functions. Quantum computers, once practical, could dramatically reduce the time needed to solve problems that classical computers find challenging, such as factoring large numbers or finding discrete logarithms, which could potentially compromise the security of SHA-256.

Anticipating these developments, NIST initiated a competition in 2007 to develop a new cryptographic hash algorithm, which would be less susceptible to potential quantum attacks. The result was the creation of the Secure Hash Algorithm 3 (SHA-3) family, which was finalized in 2015 [13]. However, the adoption of SHA-3 has been slow due to the continuing effectiveness of SHA-2, especially SHA-256.

In summary, SHA-256 represents an important milestone in the evolution of cryptographic hash functions. It enhanced the security of its predecessors and became a standard in the world of data integrity and digital security. Understanding its strengths and weaknesses provides crucial insight into the development of future cryptographic tools and standards. It's a living testament to the ever-advancing nature of cryptographic techniques, which continue to evolve in response to the escalating arms race between security mechanisms and computational power.

4.2.4 Bcrypt

Bcrypt is a password-hashing function designed by Niels Provos and David Mazières in 1999, originating from the Blowfish cipher. bcrypt is a key derivation function which is built to be resistant to brute-force search attacks. While it shares conceptual similarities with hash functions like MD5 or SHA-1, it was explicitly developed for password hashing and thus, includes unique features aimed at thwarting certain attack vectors. [9]

Bcrypt operates by taking a password input and a salt, which is a random string, and then combining these inputs to generate a hashed output. The algorithm includes a work factor or cost, which determines the complexity of the hash and can be scaled up as hardware becomes more powerful, maintaining the algorithm's resistance to attacks over time.

The working mechanism of bcrypt involves several steps. Initially, the user-selected plain text password and a salt, typically a 22-character random string, are taken as inputs. The cost factor, a component that controls the number of iterations or rounds the function has to perform, is also defined. These iterations essentially increase the time it takes to hash the password, thereby making brute-force attacks less effective.

The password and salt are then input to a modified version of the Blowfish key setup algorithm, which includes a complex and expensive computation that repeats for a specified number of rounds dictated by the cost factor. The result is a 60-character string that includes the algorithm identifier (bcrypt), cost factor, salt, and the hashed password.

Upon its inception, bcrypt offered significant advantages over other password storage mechanisms due to its resistance to brute-force attacks and rainbow table attacks, owing to its unique salt and adaptability features. It provided a computationally intense, adjustable mechanism to secure passwords, making it much harder for attackers to utilise increasing computational power to their advantage.

However, bcrypt is not without potential weaknesses. While the cost factor serves to deter brute-force attacks, it also introduces a potential vulnerability to Denial of Service attacks. A high-cost factor can significantly increase the CPU and memory required to hash a password, and an attacker can exploit this by repeatedly requesting password hashes, straining system resources.

In the ever-advancing world of technology, bcrypt stands as a testament to thoughtful design for password security. Its resistance to brute-force attacks, adaptability to increasing computational power, and widespread adoption make it an essential tool in the cryptographic toolbox. Understanding its design and limitations is crucial in advancing our knowledge of cryptographic techniques and developing future, more secure algorithms. As hardware capabilities continue to grow, the importance of adaptive, scalable algorithms like bcrypt only becomes more pertinent.[12]

4.2.5 Benchmarks

Benchmarks serve a critical role in the process of data analysis. They provide a standardised point of reference, or a 'baseline', against which we can measure and compare the performance of different test scenarios. This aids in interpreting the results of the study and drawing meaningful conclusions.

In the context of the study, benchmarks are established based on a benchmark functionality that hashcat includes. These benchmarks are run by iterative hashing the output of the previous hash. Table 4.1 shows the bench-marked values for each selected hash function.

Table 4.1: Table showing results of benchmark in both unoptimised kernel using the nightmare workload profile and optimised kernel

Hash function	**Hash speed**	
	`unoptimised kernel`	`optimised kernel`
MD5	25443.8	58999.5
SHA-1	12973.6	18822.3
SHA2-256	2358.6	7792.2
Bcrypt	89429	77390

4.3 Brute-force/mask attack

This section will focus on cracking speeds using the basic brute-force[7] attack. Hashcat's name for this style of attack is mask attack[8]. A mask attack is different to brute-force attack in that mask attack requires a mask that will be used to generate the passwords. In a brute-force attack keyspace is not limited, while in a mask attack, a mask is used to limit the possible combinations and thus reduce the overall number of passwords generated. A brute-force attack can be achieved using a mask that consists of the entire keyspace, in hashcat represented by `?a`.

Mask attacks' advantage is mainly that people tend to follow patterns while creating passwords. Using masks, such patterns can be reproduced and cracking will be limited to passwords matching this pattern. For example, a seven-letter name followed by a year would be represented by `?u?l?l?l?l?l?l?d?d?d?d`, where `?u` represents all uppercase letters, `?l` represents all lowercase letters and `?d` represents numbers from 0 to 9. If one were to reproduce a brute force attack, a `?a` mask would be used, where `?a` represents all upper and lowercase letters, numbers from 0 to 9 and special characters.

The important parameters for mask attacks are:

- mask,
  - length,
  - character set,
- Markov threshold 2.6,
- single-hash vs multi-hash,
- salt.

4.3.1 MD5

In this section, the performance of hashcat using the MD5 algorithm with mask attack is studied.

Mask length

Using masks of length of less than eight resulted in the hashcat finishing the session before the timeout of three minutes has ended, thus not providing enough data. On the other hand, hashcat does not support masks longer than twelve characters using the `?a` character set.

As seen in Figure 4.1, the speeds are overlapping. The average speed of the cracking is `22 921.5699 MH/s` with a standard deviation of `31.84 MH/s`, which is `0.1389%`. The GPU utilisation stayed at `100%` most of the time, which indicates full usage of available hardware resources. From this data, we can assume that the length of the mask does not matter.

But we can see that the hashing speed is just `90.08%` of the benchmark speed.

When we take a look at the optimised kernel in Figure 4.2, the story is more interesting, with the average speed of cracking being `57 798.4251 MH/s` and the standard deviation being `658 MH/s`, which comes out to `1.14%`. These differences are clearly visible in the

[7] https://hashcat.net/wiki/doku.php?id=brute_force_attack
[8] https://hashcat.net/wiki/doku.php?id=mask_attack

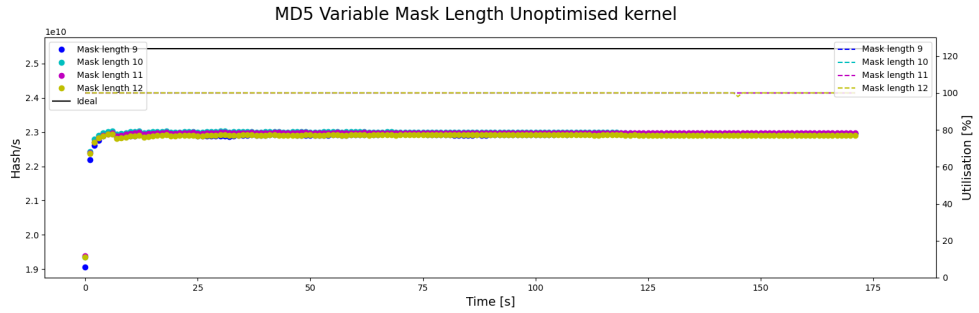

Figure 4.1: **Unoptimised kernel mask length attacks**

Figure, where the speed fluctuates. There's also a visible dip in performance around the 40-second mark on all tests, which might indicate a possible optimisation error in the kernel.

The hashing speed is `97.96%` in comparison to the benchmark, which indicates that the optimised kernel solves bottlenecks that are on the kernel level. On the other hand, a more fluctuating GPU utilisation is visible. The average hardware utilisation is `97.79%`.

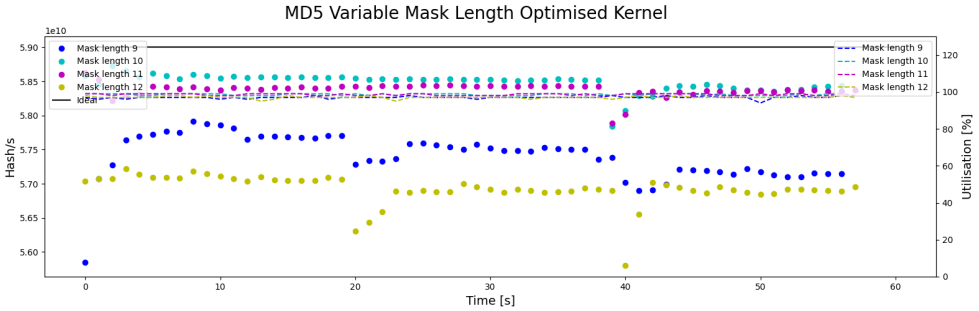

Figure 4.2: **Optimised kernel mask length attacks**

Character set

For all runs, the combined character set has been set to `?a?l?u?d?h?s?a?l?u?d?h?s`. As clearly visible from Figure 4.3, limiting the mask to only numbers drastically reduces performance. The difference to combined character sets is `3 577.9 MH/s`, which results in `14.5%` speed decrease to combined character sets. This is most probably the result of the limited character set, which consists of only ten characters. Looking at the `h` character set, a degraded performance is also visible. The `h` represents all numbers and „`abcdef`“, resulting in 16 characters.

The `a` character set, which represents all lower and upper case letters, numbers and special characters, thus being the largest character set, also achieves only `91.9%` performance of the combined character set. The overall hardware utilisation is at `99.99%`, where it is in the ballpark of statistical error.

The cracking speed of character sets `u l s` and combined are all within of `1%` of each other. This hints that a well-designed mask is the key to maximum performance while cracking passwords using the mask attack.

Optimised kernel 4.4 runs show a similar story, the only big change is that character set `a` is now similarly preferment as character sets `u l s` and combined. The overall hardware

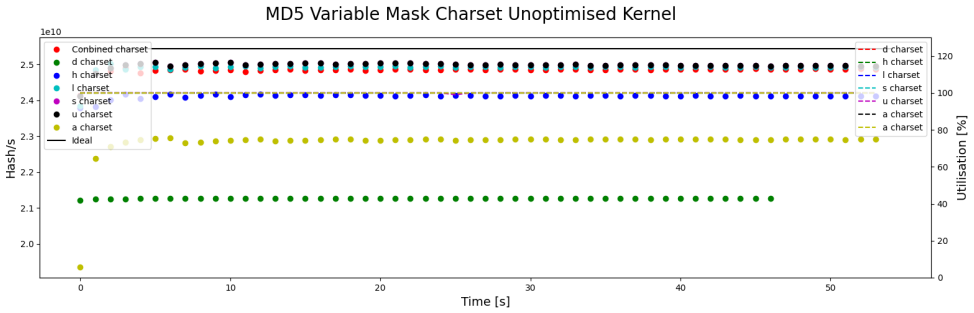

Figure 4.3: **Unoptimised kernel variable character sets in mask attack**

utilization is hovering at `98.47%`. This is still `122.19%` faster than the same runs on an unoptimised kernel. With this data, a conclusion is that the optimised kernel is always

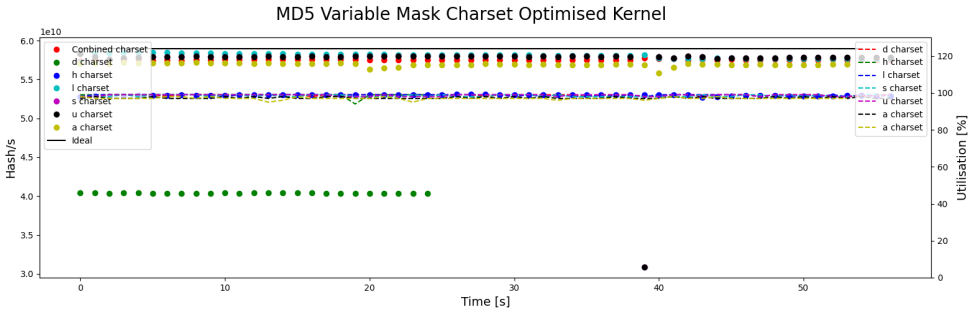

Figure 4.4: **Optimised kernel variable character sets in mask attack**

preferred, as it will allow for optimal usage of the `a` character set. There are no down sights to using it. On the other hand, there are hash speed downlights to using only `d` or `h` character sets. Using purely these will result in degraded performance.

Markov threshold

As far as the Markov threshold goes, a visible drop performance can be seen in Figure 4.5 with thresholds under 20. Hardware utilisation is `100%` all around. Markov threshold 40 achieves almost the same performance as 0, with small degradation going up from there. This may be caused by the Markov table that hashcat is using. The standard deviation is `1093872077` which comes out to `4.46%` of the overall performance.

When running on an optimised kernel, the differences are negligible as seen in Figure 4.6. We can see that threshold `5`, `10`, `15` and `20` won't even have time to get up to speed, as the range of passwords is highly limited. Thus these data are not incorporated into the analysis. The overall hardware utilisation stays at `96.31%`, with the average speed of hashing being `54 279.5761 MH/s`. The standard deviation is `2 572MH/s`, which comes down to `4.74%`. When the password cracking is about to end, the hashing speed and hardware utilisation is reduced, as single cores run out of passwords to hash. This can be visible in experiments `30` and `35`.

An important test is to see if combining all previous parameters will result in degraded performance. Knowing that limiting the possible character set to only numbers will result in regarded performance, a test running multiple character sets with multiple Markov

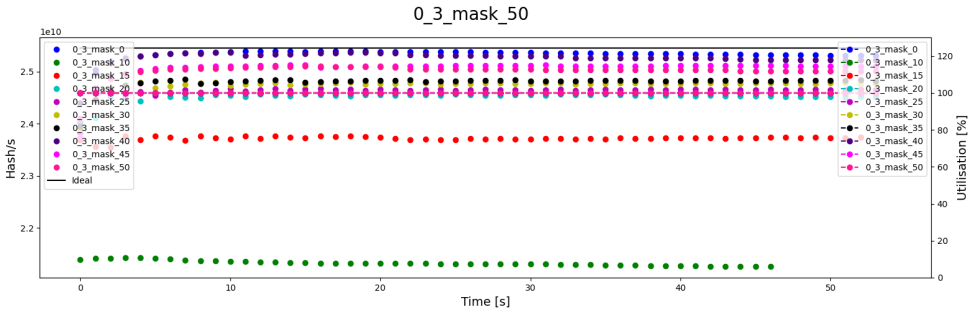

Figure 4.5: **Unoptimised kernel Markov threshold**

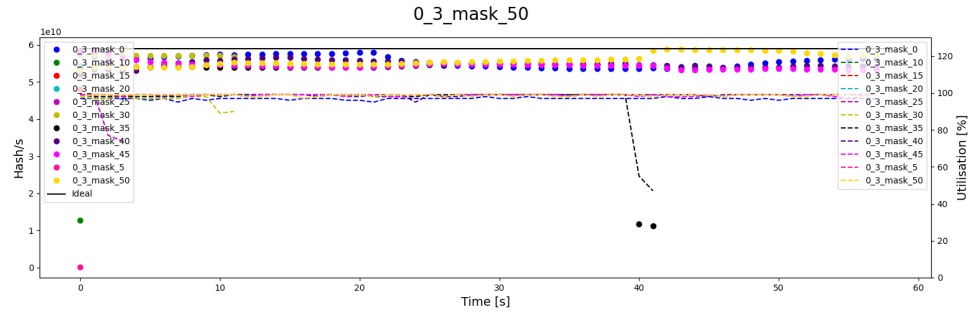

Figure 4.6: **Optimised kernel Markov threshold**

thresholds has been conducted. From these tests, it's clear that the character set has no real impact, the standard deviation comes down to `2.80%` on optimised and `3.69%` on unoptimised kernel.

Multi-hash vs single-hash

For the algorithm MD5, multi-hash sessions run at the same speed as a single hash on the unoptimised kernel. This is however not true when we move to an optimised kernel, where the multi-hash session runs at `34 575.295 MH/s`, compared to `57 798.6066 MH/s`, which is only `60%` as fast. This can be seen in Figure 4.7.

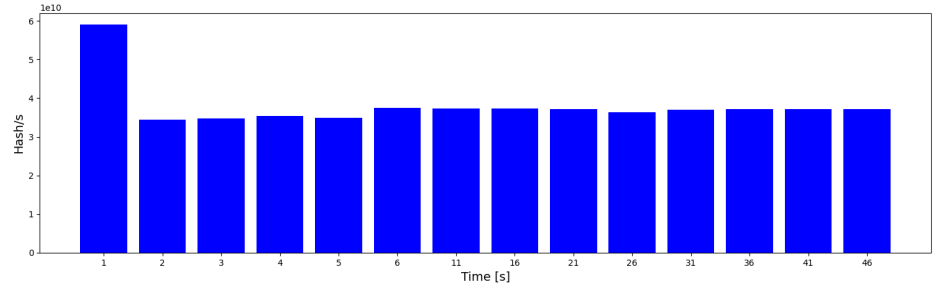

Figure 4.7: **Single hash vs multihash**

Salt

From Figure 4.8, several conclusions can be drawn. Salts affect the pure hacking performance of unoptimised kernel and runs at only `90%` of its normal performance. It's a little better for the optimised kernel, where the performance drops only to `95%`. Running multiple salted hashes on the unoptimised kernel results in only `32.56%` of the unsalted multi hash. Optimised kernel again runs closer to its multi hash performance and that is at `92.38%`.

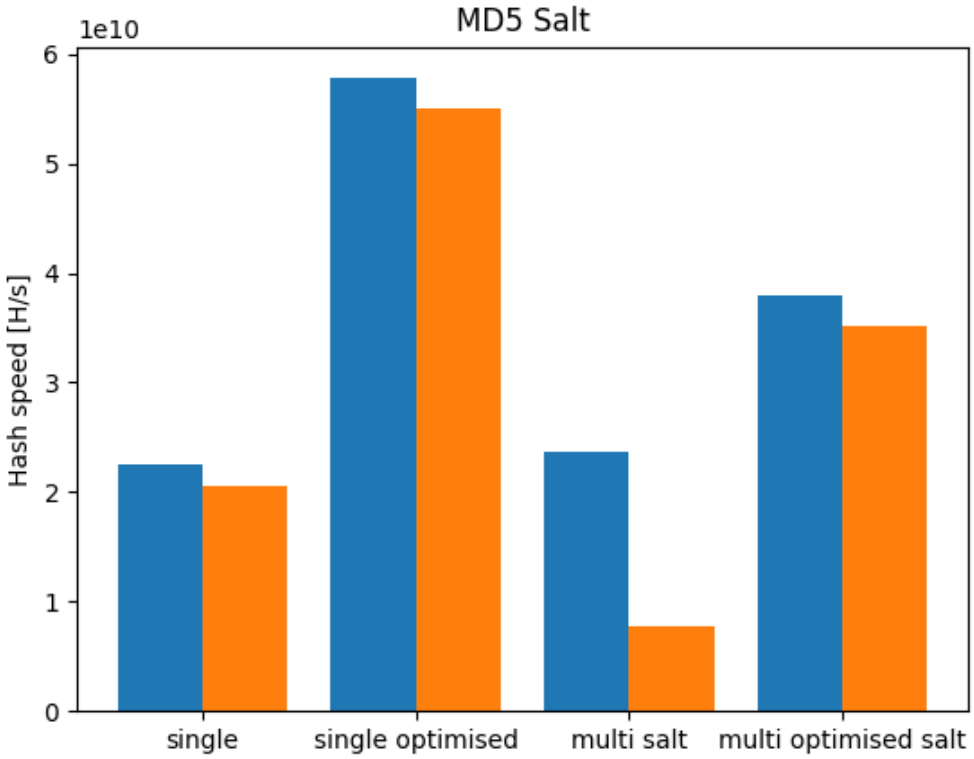

Figure 4.8: **Salted performance**

Conclusion

In conclusion, it's clear that the difference between optimised and unoptimised kernels is massive. The presence of salt also adds a performance penalty. It's important to select a good value for the Markov threshold, as small values result in degraded performance. The character set that is used also plays a role. The only thing that does not affect performance is the mask length. When combining these facts, it's possible to construct the best-case and worst-case scenarios for hashing speed.

The worst-case scenario consists of an unoptimised kernel, low Markov threshold, small character set, and running multiple salted hashes.

The best-case scenario consists of an optimised kernel, a high Markov threshold, like 50, a large character set and running single unsalted hash. These two runs are compared in

table , where we can see that the best-case scenario runs `366.67%` faster than the worst-case scenario.

4.3.2 SHA-1

In this section, the performance of hashcat using the SHA-1 algorithm with mask attack is studied.

Mask length

Using masks of a length of less than 7 results in hashcat finishing the session before the timeout has ended. thus not providing enough data. As seen from Figure 4.9, it's clearly visible that mask length does not have a big performance difference on the hash speed. The standard deviation is `689.8 MH/s`, which comes down to `5.56%`.

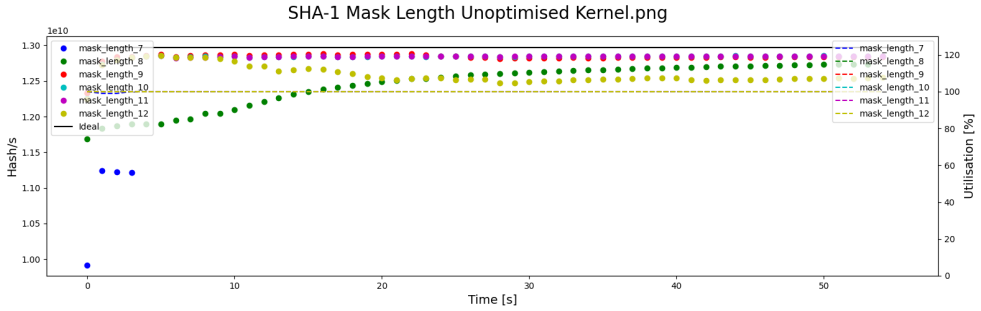

Figure 4.9: **Unoptimised kernel mask length attacks**

From the get-go, the difference in performance during benchmarking between optimised and unoptimised kernels is only `45%` speed gain in the optimised kernel. In a real-life attack, this difference comes down to `42%`. As seen in Figure 4.10, the speeds for different mask lengths do not change drastically and are within `2.25%` of the mean speed.

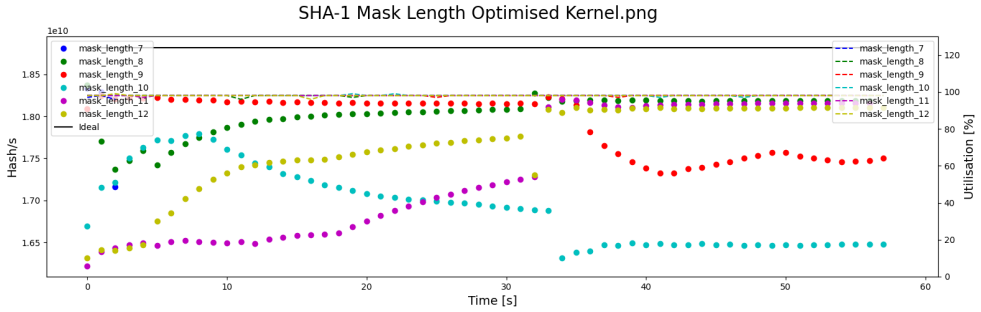

Figure 4.10: **Optimised kernel mask length attacks**

Character set

For all runs, the combined character set has been set to `?a?l?u?d?h?s?a?l?u?d?h?s`. In contrast to MD5 4.3, SHA-1 does not have big differences between longer and shorter character sets. As seen in Figure 4.11, even the character set represented by `d` is closely

comparable to others. The average hashing speed is `11 782.26 MH/s`, with standard deviation of `573.7 MH/s`, which is `4.87%`. When it comes to the optimised kernel 4.12, the

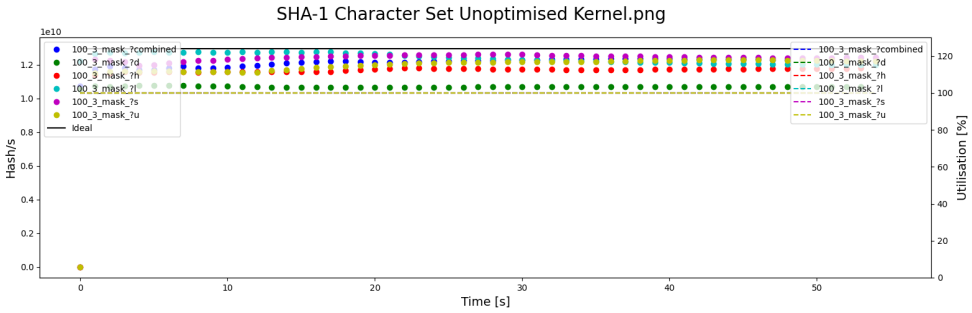

Figure 4.11: **Unoptimised kernel mask length attacks**

results are the same. The average hashing speed is `16 540 MH/s` with a standard deviation of `827 MH/s` which is `5.01%` of the overall speed. The biggest difference is again the character set represented by `d`, which indicates that with faster hashing speed, the smaller character set is becoming a bigger and bigger bottleneck.

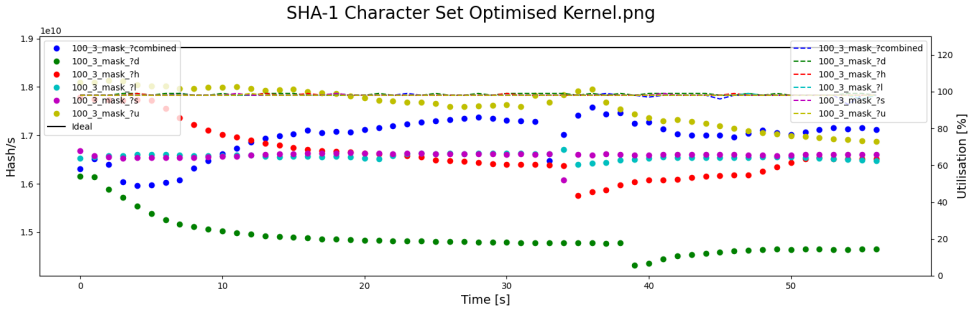

Figure 4.12: **Optimised kernel mask length attacks**

Markov threshold

Markov threshold has little to no effect on the overall performance for both optimised and unoptimised kernels. As seen in the Figures 4.13 and 4.14. Low thresholds show low hashing speed, as they are finished in the first second, thus they were removed from the calculations, to avoid skewing the data. For an unoptimised kernel, the standard deviation is just `1.52%`, while for the optimised kernel it is `2.21%`. As before, we can see that the run with Markov threshold set to `30` has degraded performance towards the end. This is due to the fact, that some GPU cores were not provided with work and were idle. This can also be seen in the utilisation line.

Multi-hash vs single-hash

For the algorithm SHA-1, multi-hash sessions run at the same speed as a single hash on the unoptimised kernel. This is similar to MD5 4.3.1 algorithm. This however is also not true for the optimised kernel, which acts in a similar way the MD5 4.3.1 did. The hashing speeds fall off as soon as more hashes are introduced. This can be seen in Figure 4.15

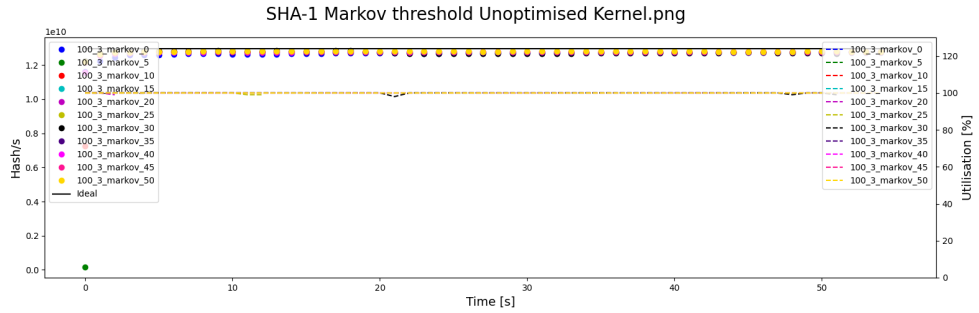

Figure 4.13: **Unoptimised kernel Markov threshold**

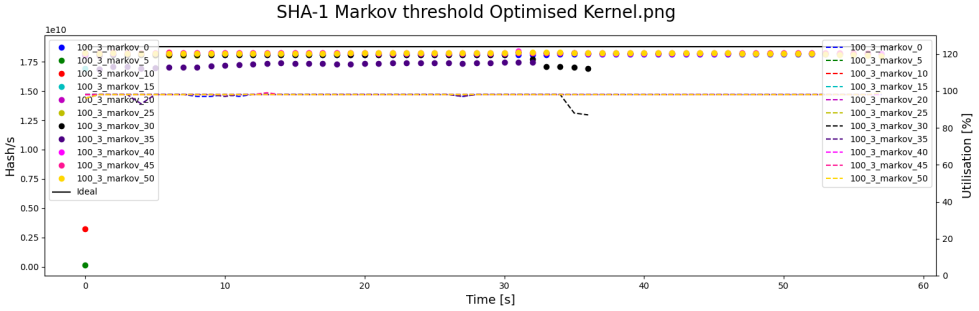

Figure 4.14: **Optimised kernel Markov threshold**

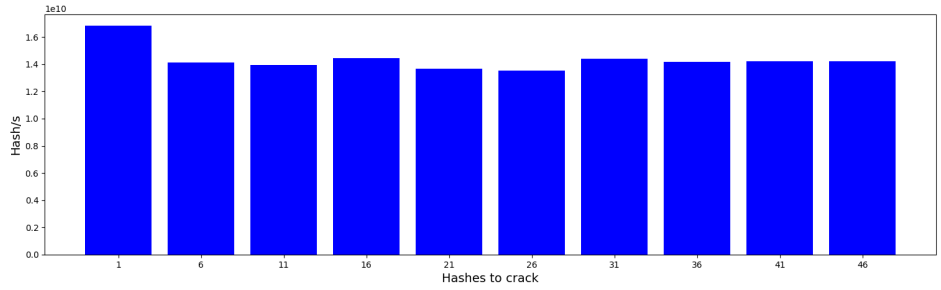

Figure 4.15: **Optimised kernel multi-hash performance**

Salt

Salt for SHA-1 does not provide a big difference, mainly for optimised kernels 4.16, where there's even better performance achieved when salt is introduced in the cracking of the password. This might look unproductive, implying the question „Why would you use salt with SHA-1?“. The main reason is to prevent cracking using recomputed rainbow tables. Even when the cracking speed is faster, it still will take a lot of time, while looking up a password in a rainbow table is instantaneous.

Conclusion

SHA-1 is well-optimised in hashcat for mask attacks. There are no real performance degradations, as there are with MD5. This means that there is no special need to structure the attacks to achieve the best performance. The worst-case scenario will be a single hash with

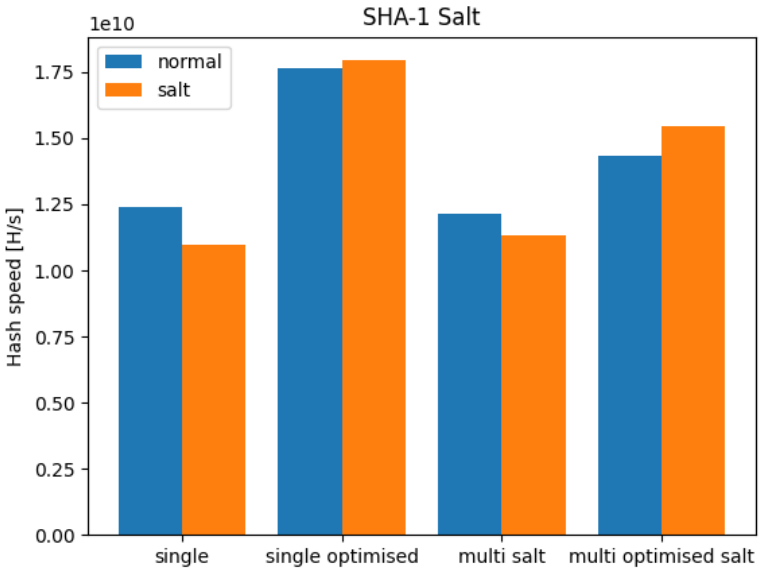

Figure 4.16: **Optimised kernel multi-hash performance**

salt, on an unoptimised kernel. The rest of the parameters are irrelevant. The best-case scenario is a single slated hash on an optimised kernel.

4.3.3 SHA2-256

In this section, the performance of hashcat using the SHA2-256 algorithm with mask attack is studied.

Mask length

As seen in Figure 4.17, the performance differences are minimal. If we ignore length seven, which has lower performance due to the limited passwords that can be generated and thus shared between cores, the standard deviation is `21.6 MH/S` which makes it `0.36%` of the average speed. It's safe to say that mask length has no effect on the overall performance

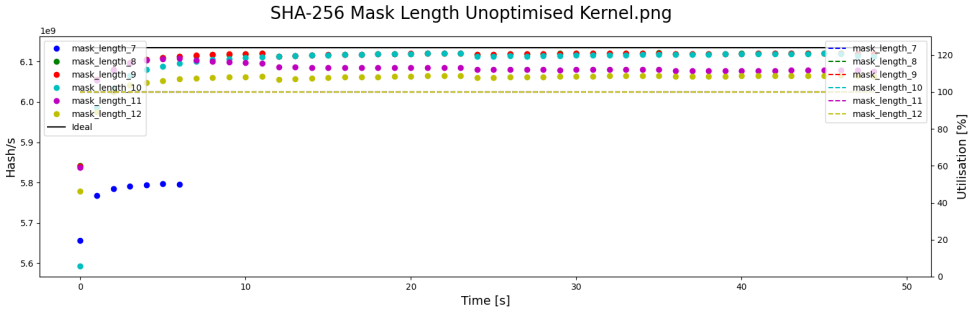

Figure 4.17: **Unoptimised kernel mask length performance**

In Figure 4.18, it's visible that even the mask of length `7` has the same performance as other mask lengths. There are no differences between mask lengths, as the standard deviation is `34.98 MH/S` which makes it `0.44%` of the average speed.

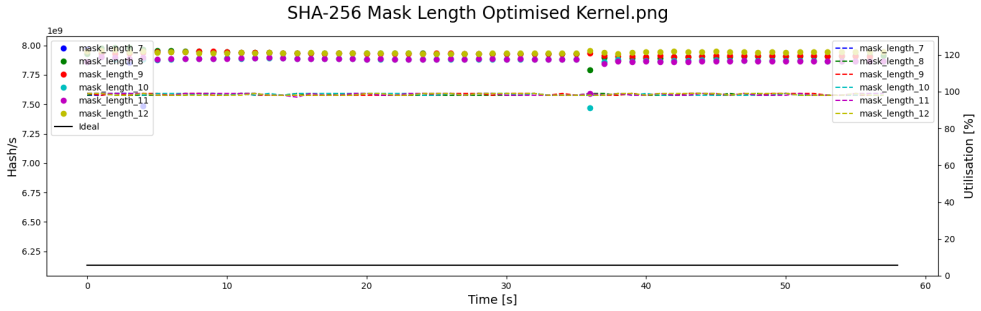

Figure 4.18: **Optimised kernel mask length performance**

Character set

It's visible in Figure 4.19 that all character sets run with similar hashing speeds. Standard deviation of only `110.90 MH/s`, which makes it `1.93%` of the average speed.

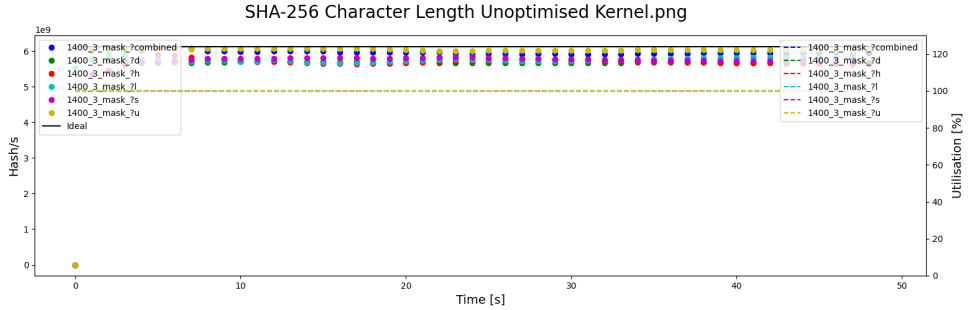

Figure 4.19: **Unoptimised kernel character set performance**

As seen in Figure 4.20 the optimised kernel reintroduces the problem with smaller character sets, indicating that faster hashing speeds favour larger character sets. The standard deviation is only `147.49 MH/s`, which comes down to `1.88%`. The hashing speed on average is `7 831.93 MH/s`, which is `1.88%` faster than the benchmarked speed. A notable phenomenon happens at the thirty-seven second for the `h` character set, where the performance drops to `6 501.89 MH/s`. This happened during multiple runs and is unknown why it's happening.

4.3.4 Markov threshold

Figures 4.21 and 4.22 show no degraded performance. A small utilisation drop can be seen at the twenty-fourth second when the session was about to end and some GPU cores finished their calculations. Otherwise, there's no visible performance drop at either the unoptimised or optimised kernels

The only speed drop-off is due to runs finishing in a second or less. In the optimised kernel graph 4.22, it's visible that it again ran faster than the benchmark at `7 940.65 MH/s`, which is `1,90%` faster.

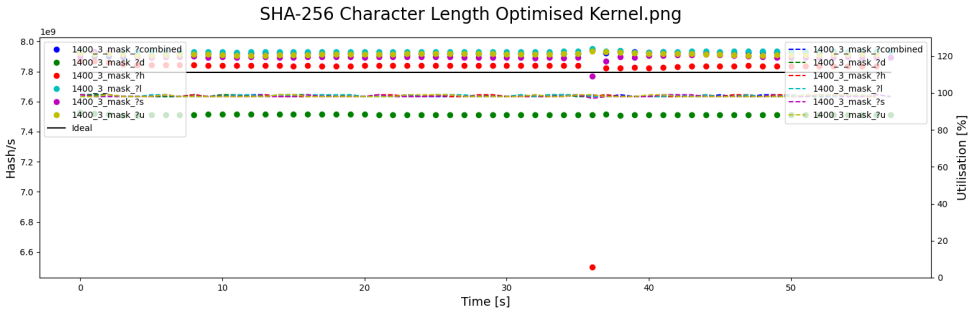

Figure 4.20: **Optimised kernel character set performance**

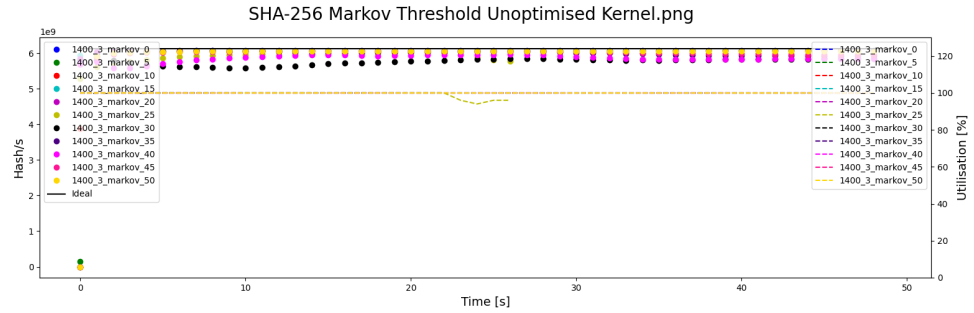

Figure 4.21: **Unoptimised kernel Markov threshold performance**

Multi-hash vs single-hash

As with previous hash functions, the multi-hash speeds of SHA2-256 are unaffected by running in a multi-hash configuration on the optimised kernel. This however is not true while running on the optimised kernel, where speeds compared to single crack fall by `12.37%`.

Salt

SHA2-256 has just a small performance drop when it comes to salt. As seen in Figure 4.24 runs with salt run approximately `10%` slower than their un-salted counterparts.

Conclusion

SHA2-256 does not suffer from almost any performance degradation while trying different run parameters while performing the mask attack. The performance degradations that are happening are small and can be ignored. This is most probably mainly because of the overall slower hashing speed, as performance degradations aren't as visible, or other bottlenecks are eliminated.

4.3.5 Bcrypt

As bcrypt is a newer hashing algorithm, it requires salts by default. This salt is saved in the hash string and is then extracted when a comparison is needed, thus salt tests will not be conducted, as there are no tests to do. Bcrypt also specified the number of iterations

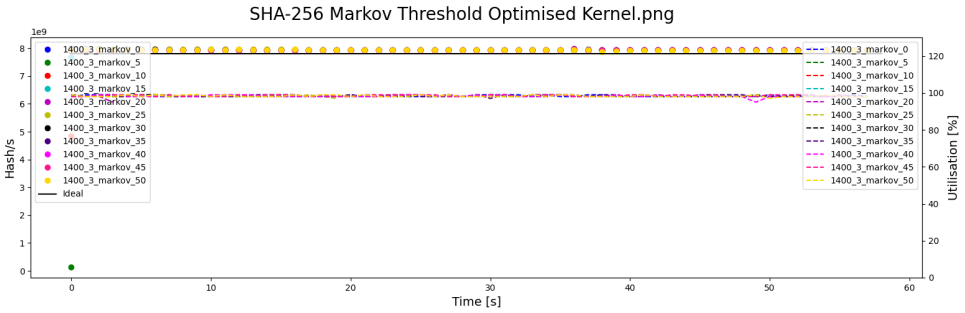

Figure 4.22: **Optimised kernel Markov threshold performance**

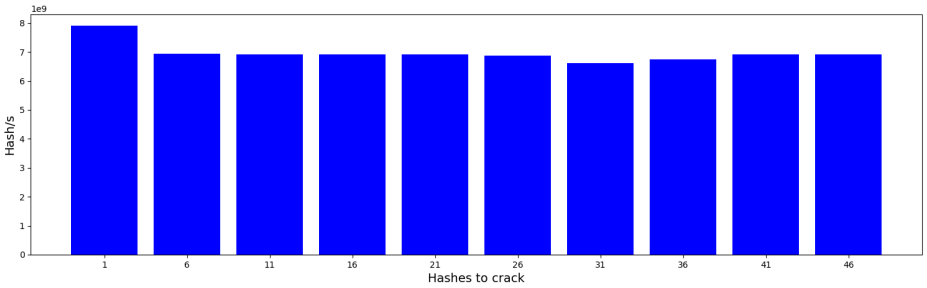

Figure 4.23: **Optimised kernel multi-hash performance**

that happen, which is also saved in the hash itself. I've chosen to test on twelve iterations, as it provides good security and the generation of the hashes didn't take too long.

Another difference here is that the benchmark line is not now shown in graphs, as it depends on the settings the bcrypt hashes were created with. Benchmarks show speeds around `77000 H/s`, while the real speeds are around one-tenth at `700 H/s`.

It's also necessary to note that spe speeds are now in `H/s` and not in `MH/s`.

Mask length

Mask length has next to no effect on the bcrypt algorithm on the unoptimised kernel 4.25. The speeds are stable and comparable with the standard deviation of `7.66 H/s`.

On the contrary, the optimised kernel hash speed fluctuates constantly 4.26, but the standard deviation is `11.23 H/s`. It is important to note, that the hashing speed on the optimised kernel is `697.97 H/s`, while on the unoptimised kernel, it is `705.94 H/s`.

Character set

The situation is similar in character set size variation tests, as the unoptimised kernel provides more stability and overall faster hashing speed 4.27. The standard deviation is `12.12 H/s` and average hashing speed `692 H/s`.

The optimised kernel again shows large fluctuations and lower hashing speeds for character sets `u` and `d`. The fastest character set hashing is with character set `l` at `720.04 H/s`, beating the unoptimised kernel. But the overall speed is just at `689.75 H/s`, while the most important, combined, character set's hashing speed average is at `695.9 H/s` coming hand in hand with the average unoptimised kernel.

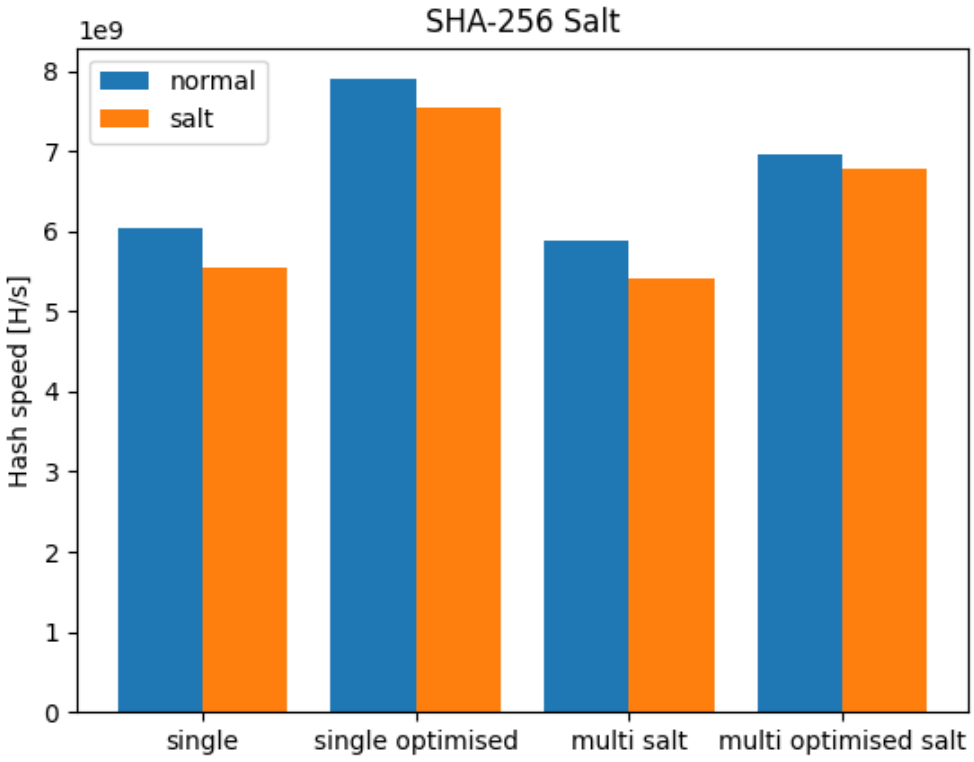

Figure 4.24: **Salt performance**

Markov threshold

Similarly to the two previous tests, it's clear that the unoptimised kernel4.29 is a better choice, as the optimised kernel4.30 contains a lot of fluctuations. The optimised kernel has stable speeds for all Markov threshold numbers. The optimised kernel on top of that shows degraded performance for Markov threshold of `50`, with further tests showing similar performance for higher numbers.

Multiple-hash vs single-hash

Surprisingly, the unoptimised kernel is showing 4.31 fluctuations when multiple hashes are being cracked at the same time, a trait that was usual for the optimised kernel. The standard deviation is `11.13%`, while the average speed is `731.77́/s`.

The fluctuations are not present while using the optimised kernel 4.32, as the standard deviation is only `1.22 H/s`, while the average speed is `28.13 H/s`.

Conclusion

Bcrypt during the mask attack the unoptimised kernel is comparably just as fast as the optimised. Even more so, when the fluctuations of the speeds in the optimised kernel are taken into regard. Using the optimised kernel might result in degraded performance, while using the unoptimised will results in comparably the same speeds with any parameters.

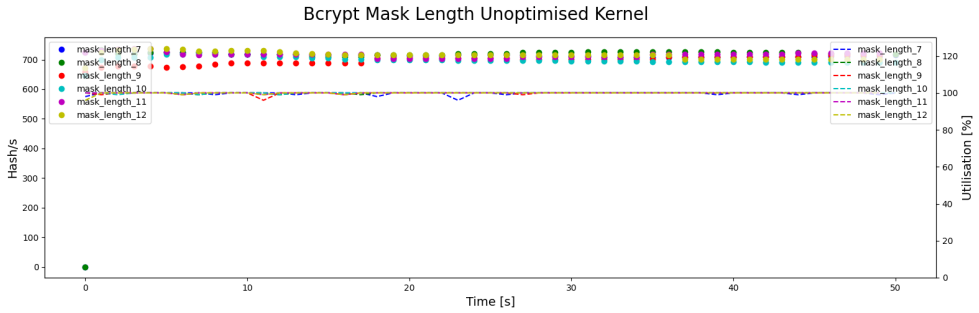

Figure 4.25: **Bcrypt Mask Length Unoptimised Kernel performance**

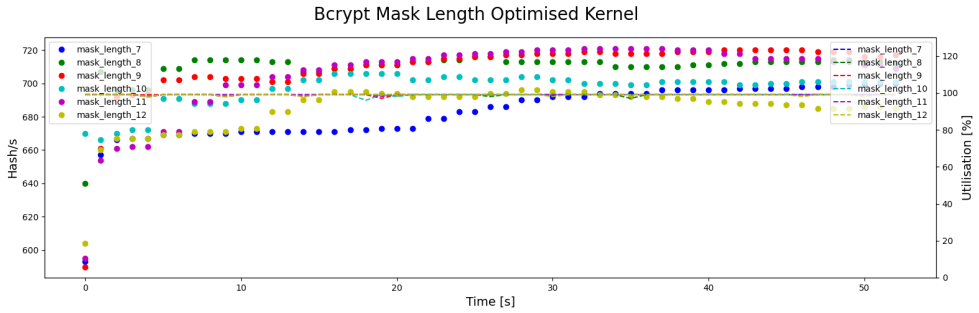

Figure 4.26: **Bcrypt Mask Length Optimised Kernel performance**

4.3.6 Conclusion

As far as mask attacks go, it's preferred to use the optimised kernel, if the password length allows for it, as it will result in faster cracking speed and better stability. The exception is of course the bscrypt algorithm, which performs better using the unoptimised kernel.

This shows that using the wrong parameters can hinder the hashing performance and it is important to know about it, so it can be avoided.

4.4 Dictionary attack

This section will focus on cracking speeds using the dictionary attack. For all attacks, a large dictionary `CrackStation's Password Cracking Dictionary`[9] has been used. This attack is an outdated one and a combinator attack is more suitable. As dictionary attacks have a big disadvantage and are way slower than their `optimal` speed, this line will not be present in the graphs.

4.4.1 MD5

Basic dictionary attack

From Figures 4.33 and 4.34, it's clearly visible that the dictionary attack does not utilise the GPU to its fullest potential, but use it at `24.4%` and `25.05%` respectively. This is a huge potential waste, as the hashing speeds are only at `17.13 MH/s`.

[9] CrackStation's Password Cracking Dictionary

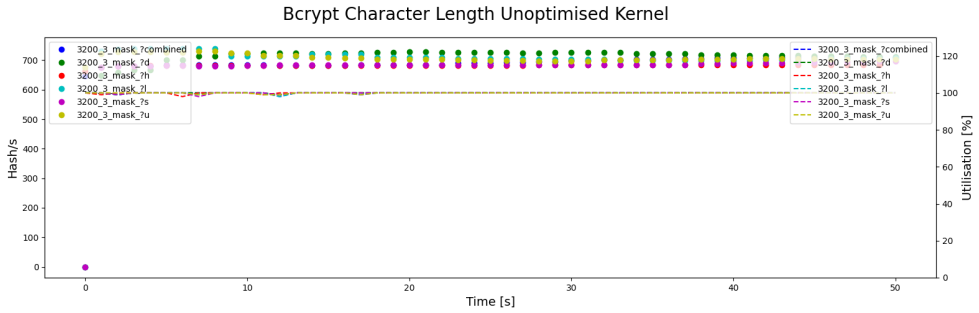

Figure 4.27: **Bcrypt Character Length Unoptimised Kernel performance**

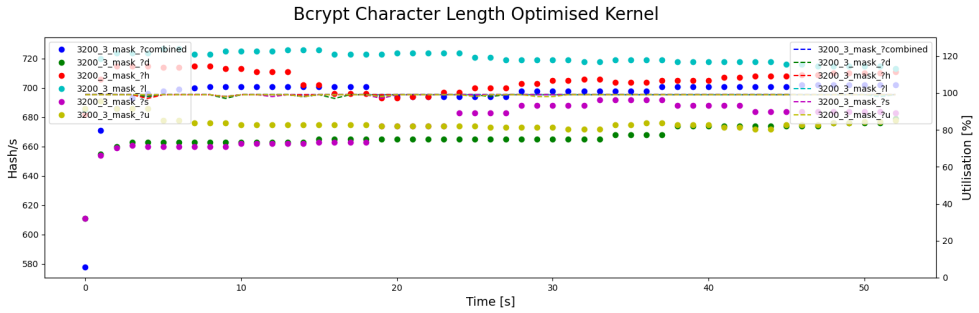

Figure 4.28: **Bcrypt Character Length Optimised Kernel performance**

Multi-hash vs single-hash

As seen in Figures 4.35 and 4.36, the situation is the same as with the basic attack. Hardware utilisation is 20% for both optimised and unoptimised kernels.

Salt and multi-hash salt

The situation is the same for both single and multi-hash salt. The results can be seen in Figures 4.37, 4.38, 4.39 and 4.40.

The only difference is that when using multi-hash with salted hashes on the unoptimised kernel, the hardware utilisation goes up to 35%, but hashing speeds stay the same.

Conclusion

A dictionary attack is not well suited to be used with fast hashing algorithms such as MD5. The most probable bottleneck is that the passwords cannot be provided fast enough, even from PCIe 4.0 4x NVMe SSD. In all Figures, there's a visible drop in speed at the start. This is due to the fact, that hashcat hashes part of the wordlist before it runs. The drop is caused because hashcat can't load new data from the wordlist fast enough and the cache is depleted.

One possible solution could be a direct access memory[10], that could eliminate the need for a CPU to load the passwords into ram and then send them to the GPU, but the GPU could load them directly from the SSD.

[10] Direct access memory

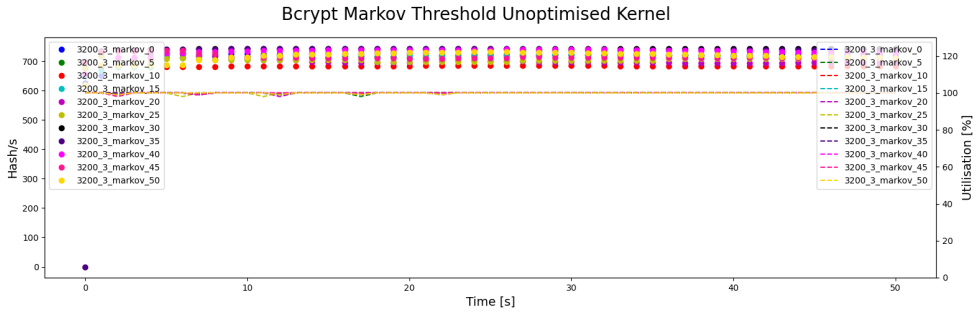

Figure 4.29: **Bcrypt Markov Threshold Unoptimised Kernel performance**

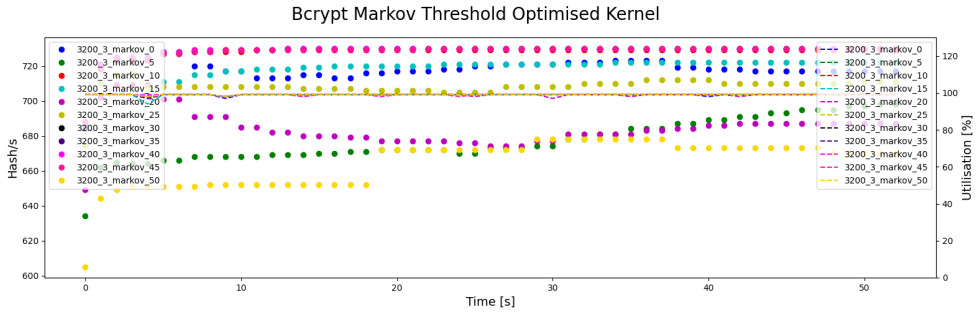

Figure 4.30: **Bcrypt Markov Threshold Optimised Kernel performance**

4.4.2 SHA-1

SHA-1 has the same problem as MD5. The CPU is not able to feed the GPU with enough passwords to crack, so the performance is low and the GPU uninitialised. Same as with MD5, a drop in hashing speed is visible at the start thanks to caching. Figures 4.41 and 4.42 represent the data from the SHA-1 unsalted tests with unoptimized kernel. The data with optimised kernel are comparable, as the bottleneck is still the drive with the passwords.

4.4.3 SHA2-256

SHA2-256 has similar results as SHA-1 and MD5, the GPU utilisation hovers around `25%`, the hashing speed is around `13 MH/s`, which is way slower than `6 036 MH/s` we can achieve using a mask attack. There are no differences between the optimised and unoptimised kernel. As before, a visible drop in hashing performance is present, as the cache of passwords is being emptied 4.43.

4.4.4 Bcrypt

Bcrypt is a slow hashing algorithm, the speeds are in the ballpark of `700 H/s`. This means that a dictionary attack is a viable attack vector, as the bottleneck of reading from a drive is eliminated. Because bcrypt does not allow usage without salts, only optimised, unoptimised kernels and single and multi-hash configurations are tested.

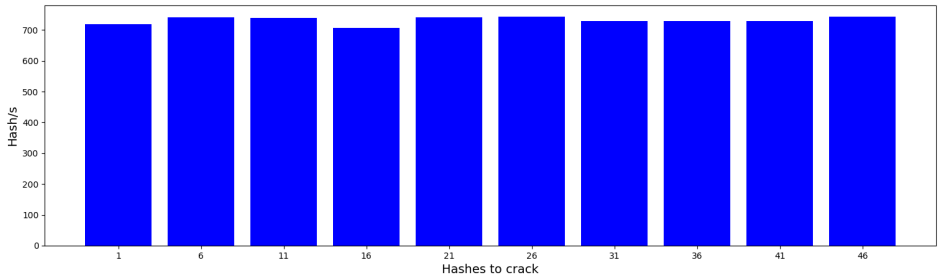

Figure 4.31: **Bcrypt Multi-hash Unoptimised Kernel performance**

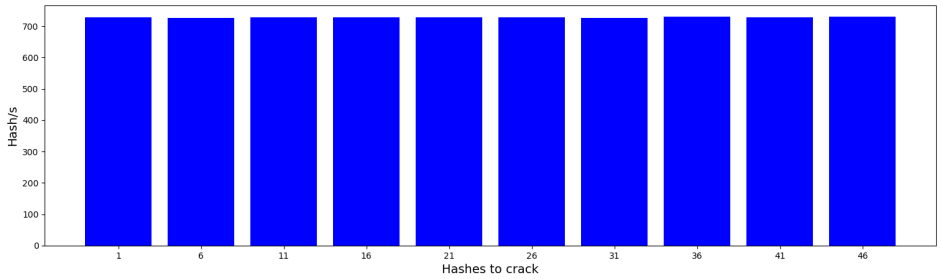

Figure 4.32: **Bcrypt Multi-hash Optimised Kernel performance**

Basic speeds

From Figure 4.44, it's visible that bcrypt does not suffer from the bottleneck as MD and SHA algorithms do. The hardware utilisation is at `99.95%`, while the hashing speed is `732.24 H/s`, which is comparable with the basic speeds using mask attack as seen in Figure 4.44.

The optimised kernel has a little slower hashing speed at `692.47 H/s` and lower GPU utilisation at `89.39%` as seen in Figure 4.45.

Multi-hash vs single-hash

Using a multi-hash on an unoptimised kernel results in no speed reduction and utilisation stays at `100%`. This can be seen in Figure 4.46

However, using the optimised kernel, as in previous tests, the speed fluctuates and is not uniform. It is better to stick to an unoptimised kernel. If absolute performance is necessary, it's important to test how the specific number of hashes to be cracked effects the performance, as it changes from number to number.

4.4.5 Conclusion

Dictionary attacks are slow, as the main bottleneck is the disk that stores the passwords. They are not suitable for hashes that have high hashing speed, as the disk will not be able to provide passwords fast enough. For hash functions that have slow hashing speed, like bcrypt, a simple dictionary attack is a viable option.

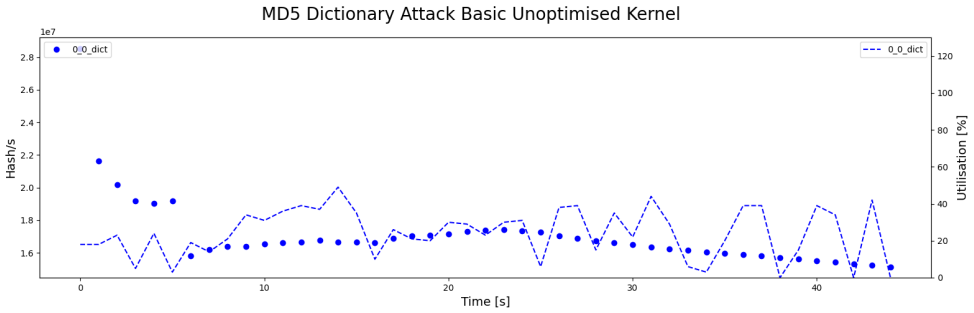

Figure 4.33: **MD5 Dictionary Attack Basic Unoptimised Kernel**

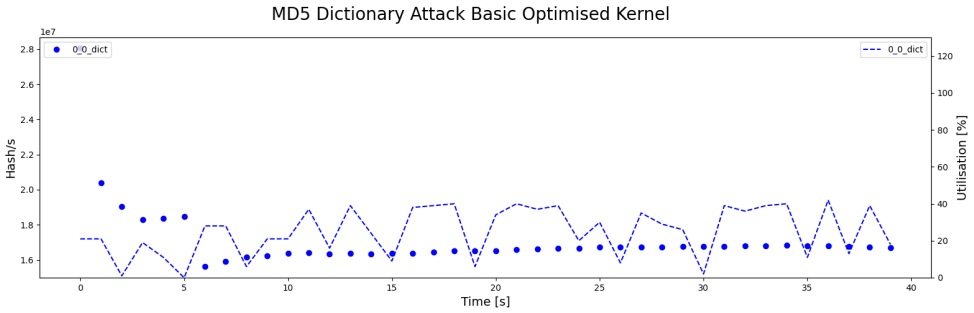

Figure 4.34: **MD5 Dictionary Attack Basic Optimised Kernel**

4.5 Combinator attack

This section will focus on cracking speeds using the combinator attack. For all attacks, a large dictionary `CrackStation's Password Cracking Dictionary`[11] with the combination of `RockYou`[12].

4.5.1 MD5

Basic attack

In Figure 4.48, it's visible that the problem of disk bottleneck is no longer present. The hardware utilisation is near `99%`. The fluctuations are no longer present when compared to pure dictionary attacks. What is interesting is that on the optimised kernel, the hashing speed is `24 395.65 MH/s`, while on unoptimised it is `18 507.45 MH/s`, which is only `75.86%` of the optimised kernels speed.

Multi-hash vs single-hash

An unoptimised kernel has the same speeds for any number of hashes, that is around `17 376.9 MH/s`. This can be seen in 4.49. The hardware utilisation stays at `100%`.

Optimised kernel however shows faster performance in a single-hash session, but multi-hash sessions are still after than on the optimised kernel hashing at `22 870.7 MH/s`. This can be seen in Figure 4.50.

[11] CrackStation's Password Cracking Dictionary
[12] RockYou

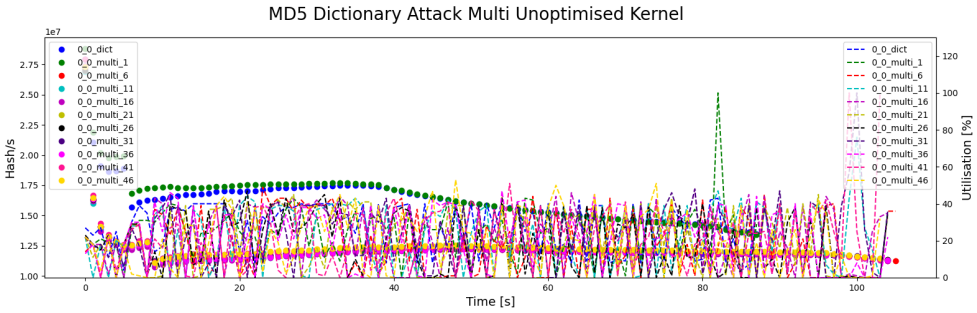

Figure 4.35: **MD5 Dictionary Attack Multi Unoptimised Kernel**

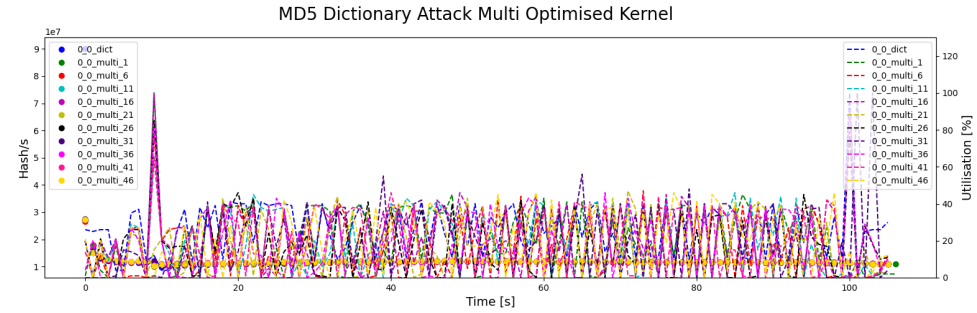

Figure 4.36: **MD5 Dictionary Attack Multi Optimised Kernel**

Salt

Running this attack with salts degrades performance. As seen in Figure 4.51, running on optimal kernel will provide `39.51%` faster cracking speeds.

When combining salt with multiple hashes, the optimised kernel will be faster. Speeds average at `20 288.6 MH/s`, which is faster than unoptimised `14 542.76 MH/s`. There are some differences in speed between different numbers of hashes as seen in Figure 4.52, but the speed benefit still stands.

Conclusion

MD5 algorithm runs at slower speeds using the combinatory attack than it does when using the mask attack. As the disk bottleneck has been removed, the biggest difference is between using an optimised kernel and not, as an optimised kernel can significantly boost performance.

4.5.2 SHA-1

Basic attack

The difference in basic speeds between optimised and unoptimised kernels is around `1 913.26 MH/s`, which makes the optimised kernel `19.0%` faster. This can be seen in Figure 4.53.

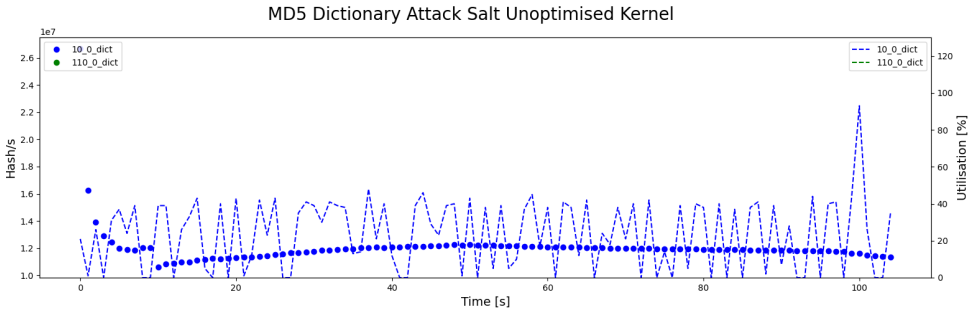

Figure 4.37: **MD5 Dictionary Attack Salt Unoptimised Kernel**

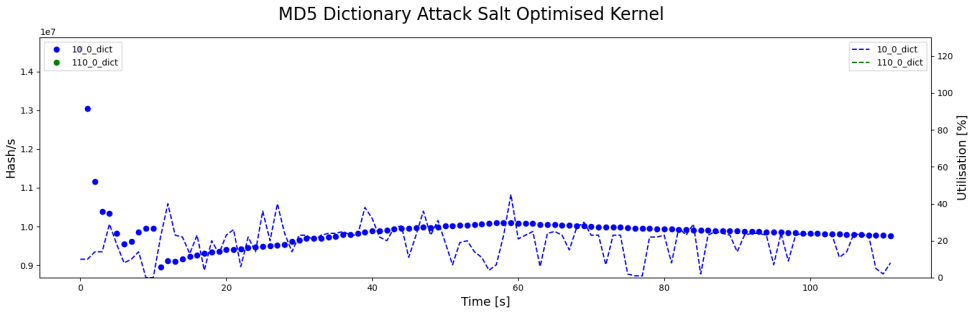

Figure 4.38: **MD5 Dictionary Attack Salt Optimised Kernel**

Multi-hash vs single-hash

In Figure 4.54, it's clearly visible that the hashing speed does not change depending on the number of hashes being tested.

While in Figure 4.55, there are clear differences between speeds. The single-hash is running the fastest at `11 923.62 MH/s`, while the multi-has of eleven hashes is running just at `11 298.27 MH/s`. This difference is reduced when a larger hash pool is introduced.

Salt

Single-hash salt runs about `8 309.22 MH/s` slower on both versions of the kernel, being reduced from `11 022.76 MH/s` to just `9 767.46 MH/s`. The difference between optimised and unoptimised kernel is about `2 916.47 MH/s` in favour of the optimised kernel. This can be seen in Figure 4.56.

Salting reduces the speed of hashing in multi-hash configuration from `10 099.17 MH/s` to `8 579.36 MH/s`. As before, the unoptimised kernel does not have big differences between the number of hashes as seen in Figure 4.57.

Salting in the multi-hash on the optimised kernel runs at `10 525.74 MH/s` with a standard deviation of `395.45 MH/s`. This can be seen in Figure 4.58.

4.5.3 SHA2-256

In this section, it is summarised how SHA2-256 performs under a combinator attack.

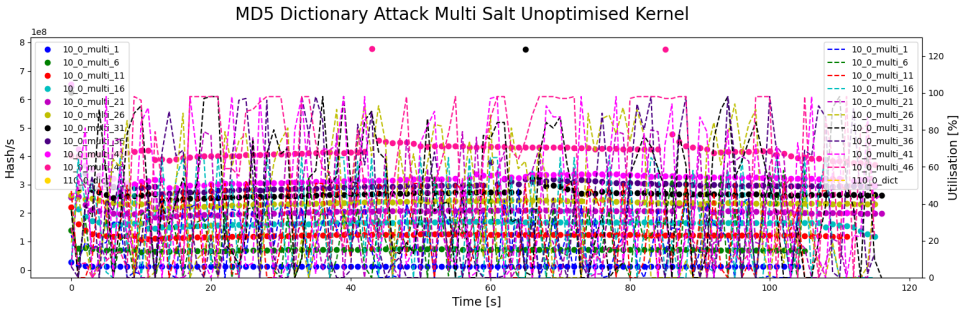

Figure 4.39: **MD5 Dictionary Attack Multi Salt Unoptimised Kernel**

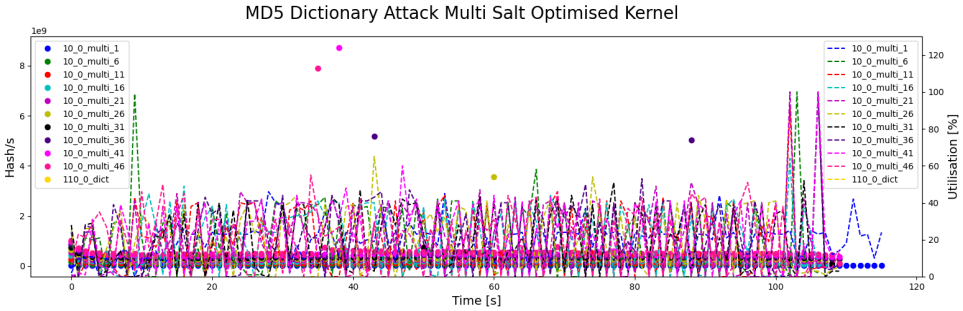

Figure 4.40: **MD5 Dictionary Attack Multi Salt Optimised Kernel**

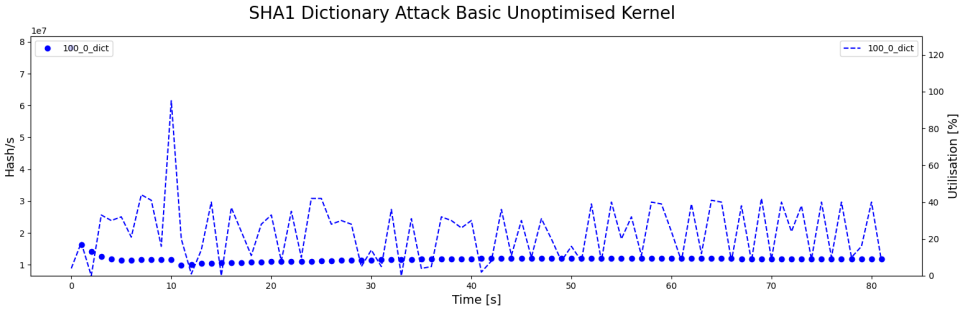

Figure 4.41: **SHA1 Dictionary Attack Basic Unoptimised Kernel**

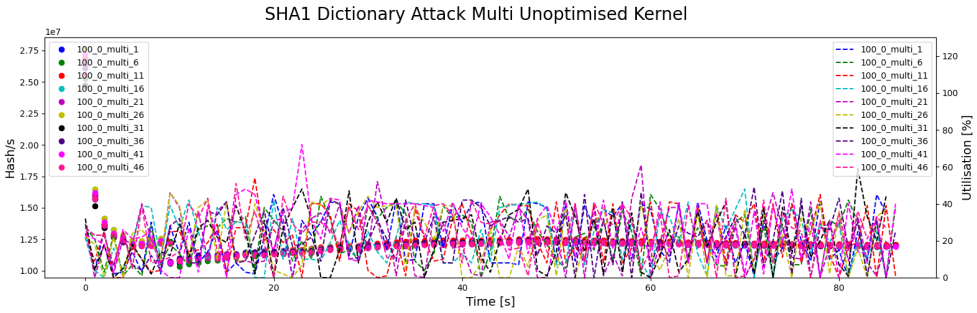

Figure 4.42: **SHA1 Dictionary Attack Multi Unoptimised Kernel**

Overall

Under the same conditions, SHA2-256 and SHA-1 exhibit decreased performance, but this decrease isn't uniform across both functions. Given the naturally slower hashing speed of

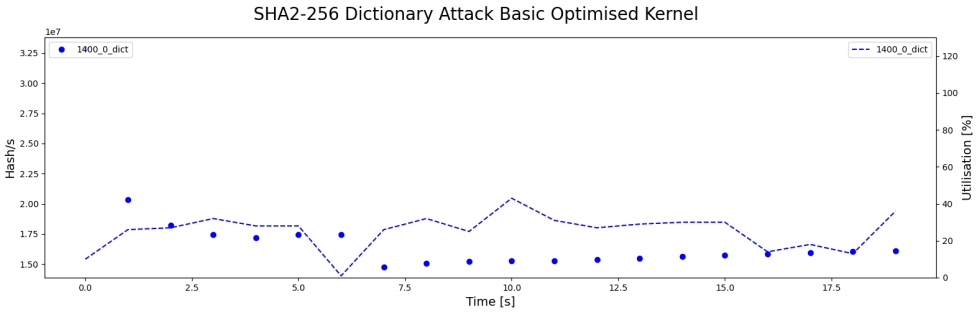

Figure 4.43: **SHA2-256 Dictionary Attack Basic Optimised Kernel**

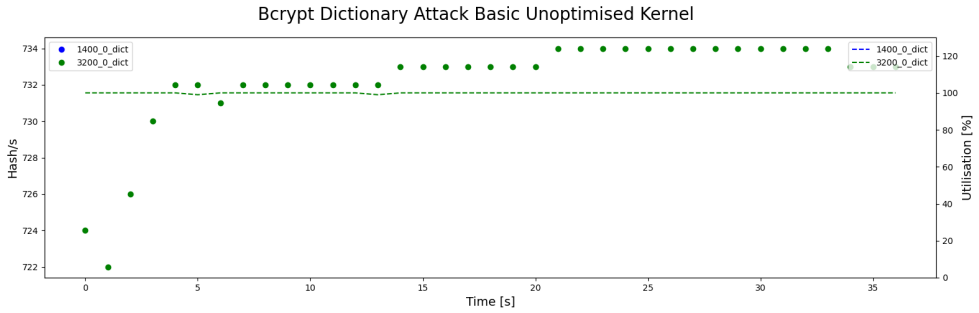

Figure 4.44: **Bcrypt Dictionary Attack Basic Unoptimised Kernel**

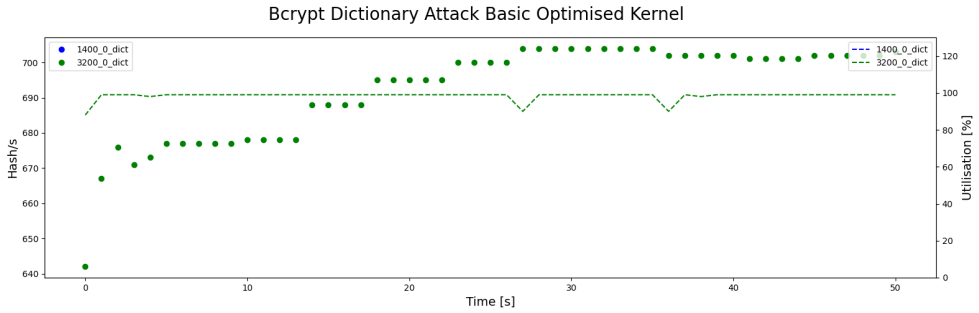

Figure 4.45: **Bcrypt Dictionary Attack Basic Optimised Kernel**

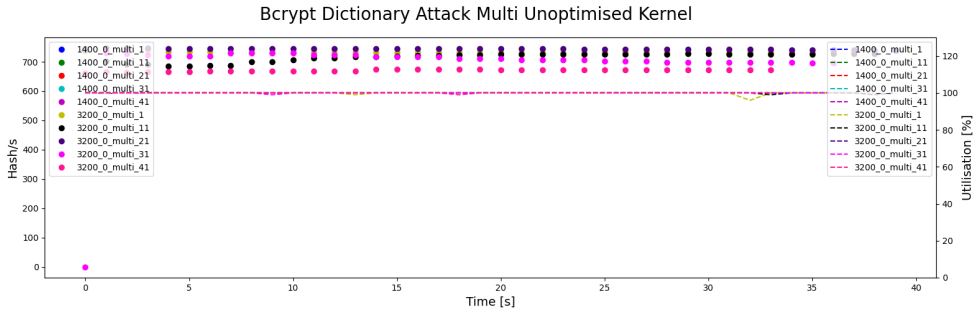

Figure 4.46: **Bcrypt Dictionary Attack Multi Unoptimised Kernel**

SHA2-256, the impact of parameter adjustments would typically result in slower speeds as compared to SHA-1.

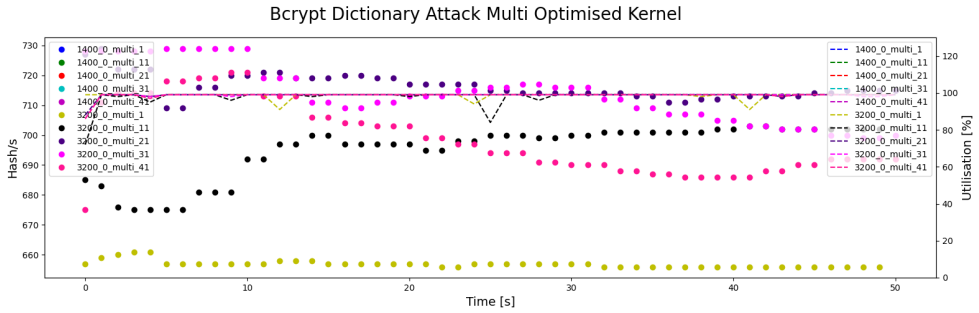

Figure 4.47: **Bcrypt Dictionary Attack Multi Optimised Kernel**

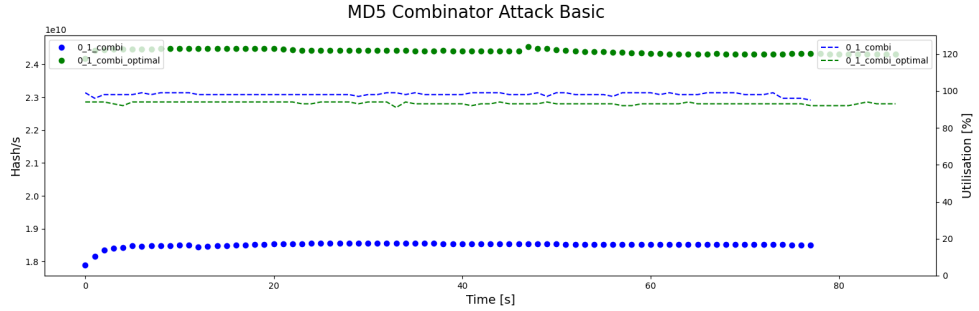

Figure 4.48: **MD5 Combinator Attack Basic**

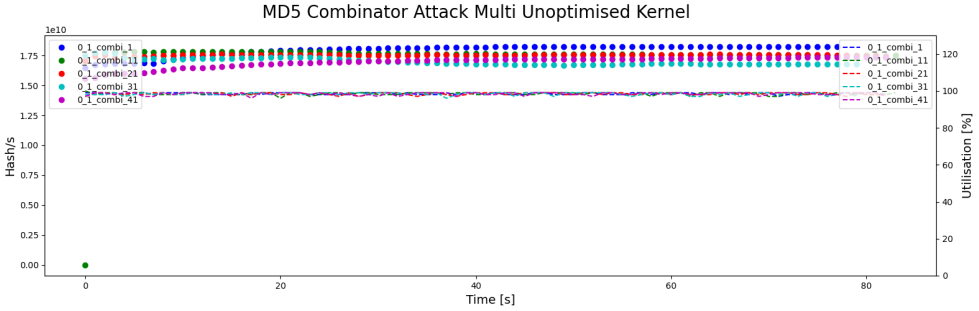

Figure 4.49: **MD5 Combinator Attack Multi Unoptimised Kernel**

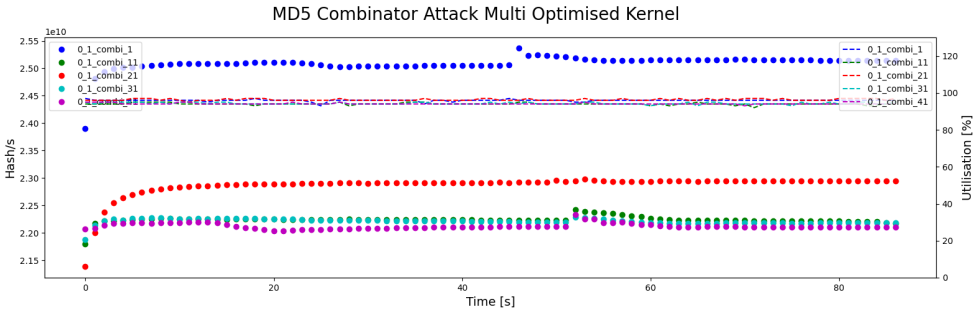

Figure 4.50: **MD5 Combinator Attack Multi Optimised Kernel**

Figure 4.59 shows that the optimised kernel is again better. This trend continues with all other tests.

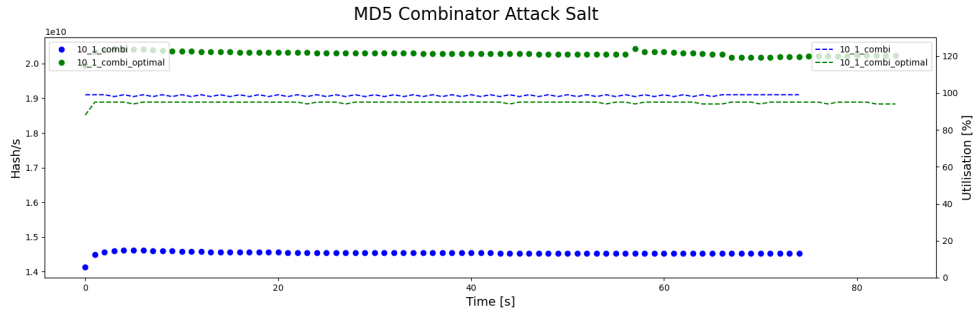

Figure 4.51: **MD5 Combinator Attack Salt**

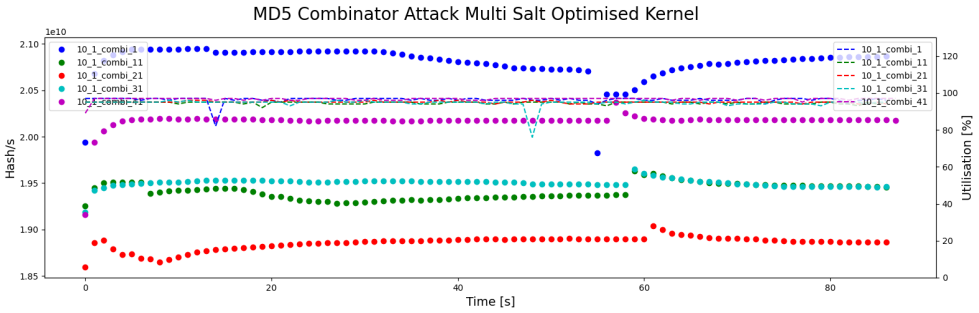

Figure 4.52: **MD5 Combinator Attack Multi Salt Optimised Kernel**

4.5.4 Bcrypt

In this section, the performance of bcrypt while performing the combinator attack is discussed.

Basic attack

As seen in Figure 4.60 it is apparent, that the unoptimised kernel is again faster, achieving speeds around `697.9 H/s` in comparison to the `690.8 H/s` of the optimised kernel. It's also apparent, that the unoptimised kernel is working in batches, as the hash speed stays the same for a couple of seconds before jumping drastically, creating a stair effect. The optimised kernel seems to have these steps in intervals of two seconds. Both kernels seem to be creating a sinus-like wave when their speed is plotted with time.

Multi-hash vs single-hash

In Figure 4.61 it's visible that both kernels are back to back to each other, with unoptimised kernel winning by small margins in most of the test cases. It is safe to assume, that the optimised kernel will be the better choice most of the time.

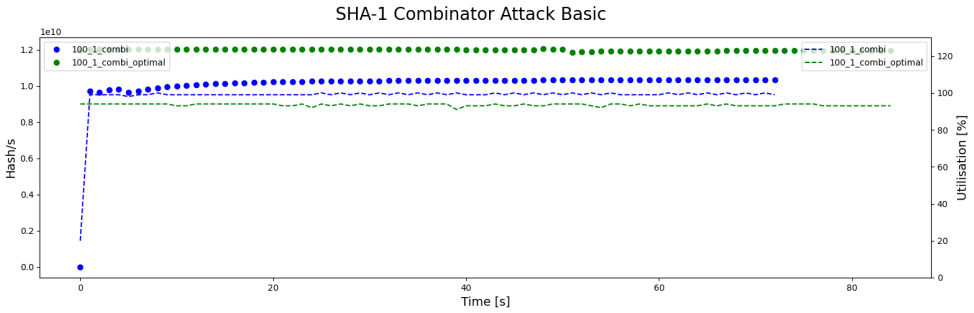

Figure 4.53: **SHA-1 Combinator Attack Basic**

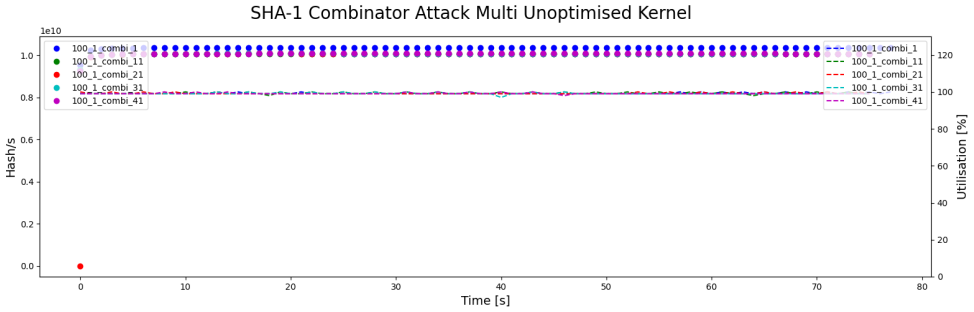

Figure 4.54: **SHA-1 Combinator Attack Multi Unoptimised Kernel**

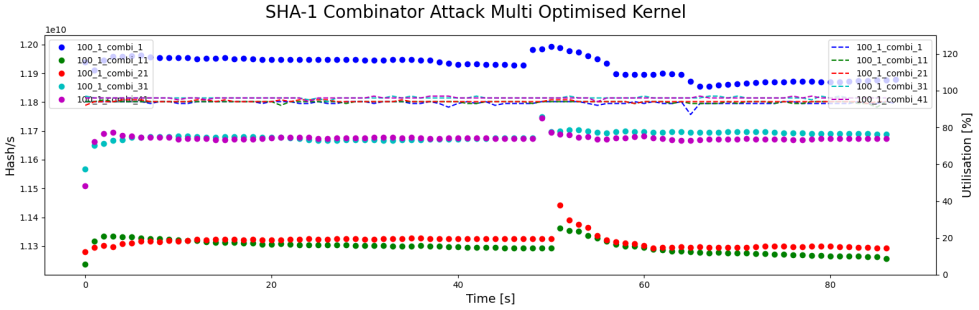

Figure 4.55: **SHA-1 Combinator Attack Multi Optimised Kernel**

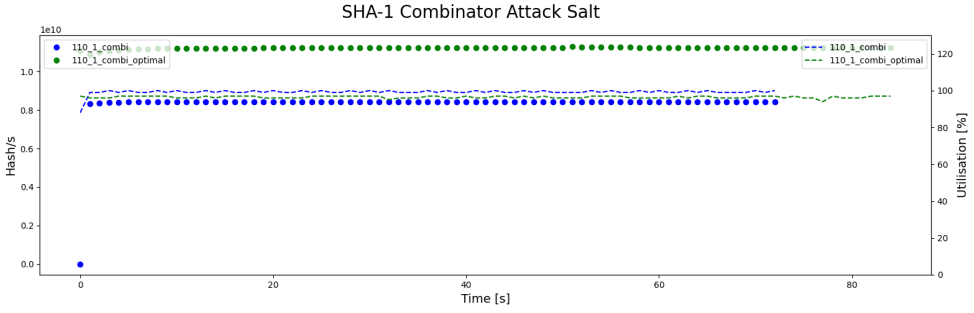

Figure 4.56: **SHA-1 Combinator Attack Salt**

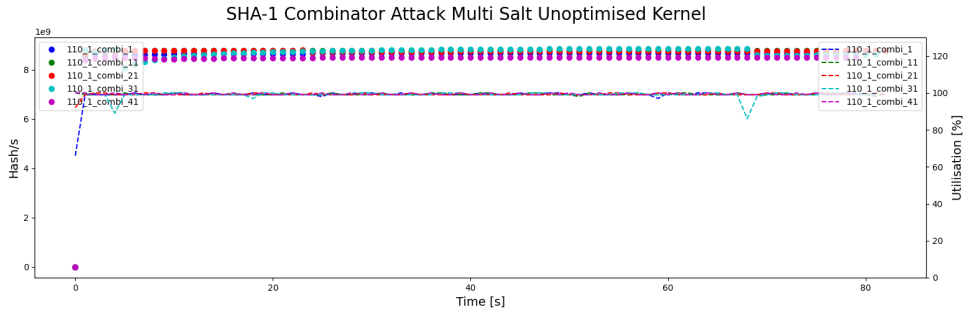

Figure 4.57: **SHA-1 Combinator Attack Multi Salt Unoptimised Kernel**

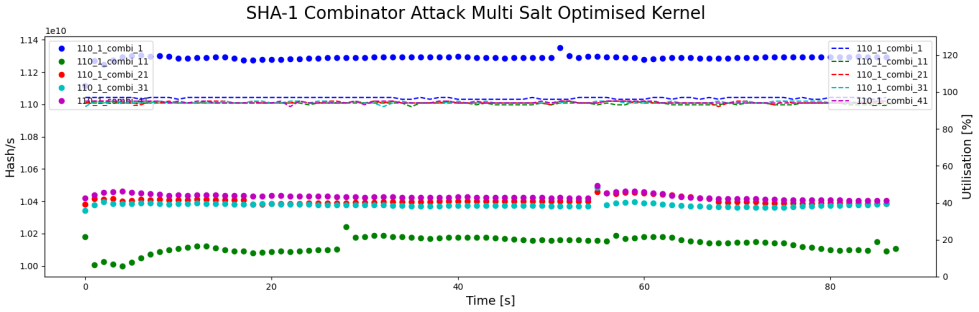

Figure 4.58: **SHA-1 Combinator Attack Multi Salt Unoptimised Kernel**

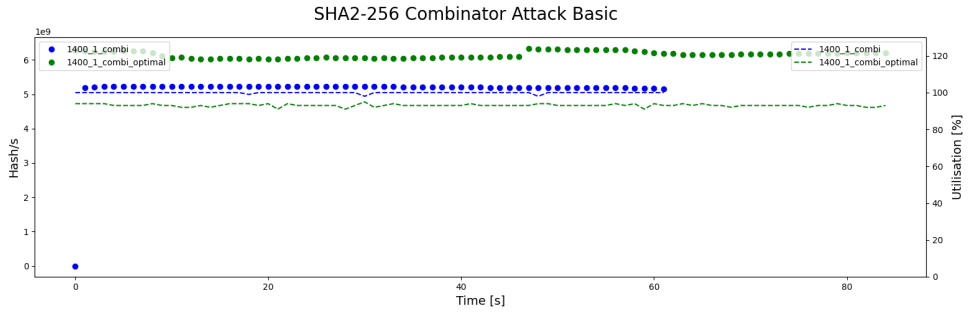

Figure 4.59: **SHA2-256 Combinator Attack Basic**

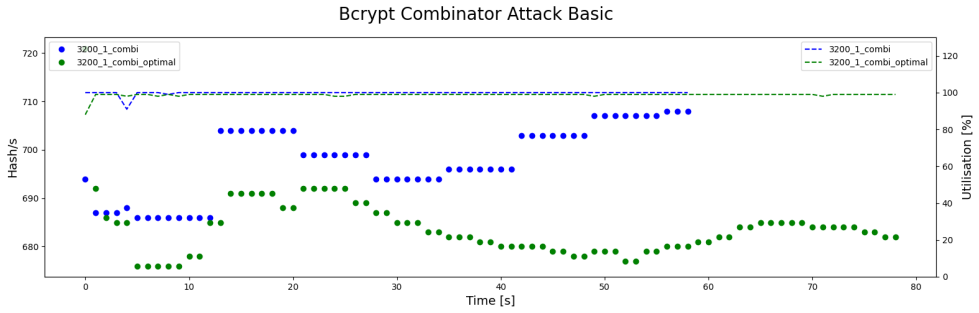

Figure 4.60: **Bcrypt Combinator Attack Basic**

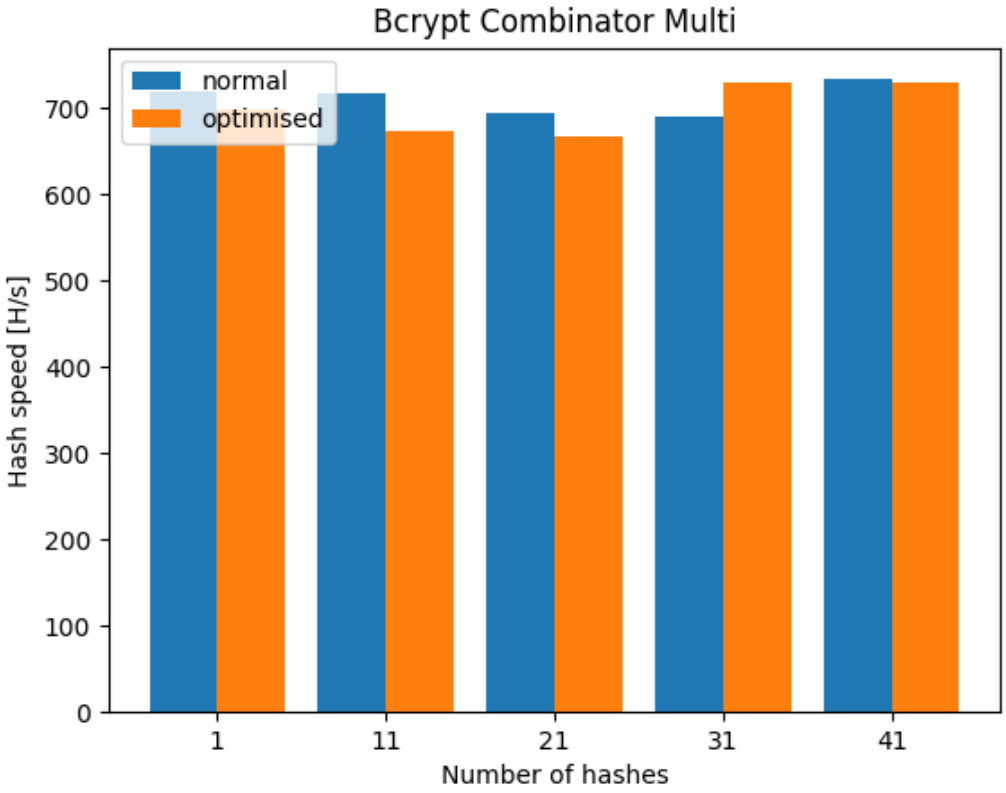

Figure 4.61: **Bcrypt Combinator Multi Hash**

Chapter 5

Conclusion

In this chapter, we summarise the result and outline possible future expansions and improvements.

5.1 Future expansions and improvements

This thesis can be expanded by testing more hash functions. As hashcat supports cracking of the majority of modern hash functions, those might be added to aid in the decision-making when constructing the hashcat session parameters. Expanding the number of hashes that the tests are run on could provide more useful information when working with large datasets.

Another possibility would be an AI predictive model, that would provide the best parameters for a given combination of requirements, such as hash function, whether hashes are salted, the attack type and the hardware. Training of such AI model would require data sets from a large number of different hardware setups.

Another possibility would be a tool, that would run all possible combinations on given hardware and store the data, so once a user needs to prepare the parameters for a hashcat session, this tool could provide insight into how these parameters will affect the performance without needing to run the tests himself. This will save time when preparing the parameters, as the data will be available once generated and only browse them.

5.2 Summary

The appropriate selection of parameters is crucial in order to optimize the hashing speed for any given attack. The required parameters differ significantly not only between different types of attacks but also across diverse algorithms. The predominant element influencing the efficacy of a hashcat session hinges on the selection of the optimized kernel.

The findings of this thesis highlight the significant advantages that choosing an optimized kernel can bring about, particularly with algorithms like MD5 or SHA-1. However, it's important to underscore that this isn't always the case. This is exemplified by the bcrypt algorithm, which experienced a decline in performance along with considerable variations in speeds when run on the optimized kernel.

A pure dictionary attack proves to be suboptimal when used in conjunction with a high hash rate function, primarily due to the bottleneck created by the disk where the dictionary is stored. When feasible, this type of attack should be avoided, particularly considering the availability of alternatives that eliminate this bottleneck. These alternatives offer a larger

dictionary by either combining two dictionaries—as seen in a combinator attack—or by appending a mask to dictionary passwords—as in a hybrid attack.

Bibliography

[1] 3rd, D. E. E. and Jones, P. *US Secure Hash Algorithm 1 (SHA1)* [RFC 3174]. RFC Editor, september 2001. DOI: 10.17487/RFC3174. Available at: https://www.rfc-editor.org/info/rfc3174.

[2] Aoki, K., Guo, J., Matusiewicz, K., Sasaki, Y. and Wang, L. Preimages for step-reduced SHA-2. In: Springer. *Advances in Cryptology–ASIACRYPT 2009: 15th International Conference on the Theory and Application of Cryptology and Information Security, Tokyo, Japan, December 6-10, 2009. Proceedings 15*. 2009, p. 578–597.

[3] Dang, Q. H. *Secure Hash Standard*. July 2015. Available at: https://doi.org/10.6028/nist.fips.180-4.

[4] Dobbertin, H. Cryptanalysis of MD5 compress. *Rump session of Eurocrypt*. Citeseer. 1996, vol. 96, p. 71–82.

[5] Dods, C., Smart, N. P. and Stam, M. Hash based digital signature schemes. In: Springer. *Cryptography and Coding: 10th IMA International Conference, Cirencester, UK, December 19-21, 2005. Proceedings 10*. 2005, p. 96–115.

[6] IAB, Daigle, L. and Kolkman, O. *RFC Streams, Headers, and Boilerplates* [RFC 5741]. RFC Editor, december 2009. DOI: 10.17487/RFC5741. Available at: https://www.rfc-editor.org/info/rfc5741.

[7] Norris, J. R. *Markov chains*. Cambridge university press, 1998.

[8] Picolet, J. *Hash crack*. North Charleston, SC: Createspace Independent Publishing Platform, september 2017.

[9] Provos, N. and Mazieres, D. A future-adaptable password scheme. In: *USENIX Annual Technical Conference, FREENIX Track*. 1999, vol. 1999, p. 81–91.

[10] Rivest, R. L. *The MD5 Message-Digest Algorithm* [RFC 1321]. RFC Editor, 1. april 1992. DOI: 10.17487/RFC1321. Available at: https://www.rfc-editor.org/info/rfc1321.

[11] Sobti, R. and Geetha, G. Cryptographic hash functions: a review. *International Journal of Computer Science Issues (IJCSI)*. Citeseer. 2012, vol. 9, no. 2, p. 461.

[12] Sriramya, P. and Karthika, R. Providing password security by salted password hashing using bcrypt algorithm. *ARPN journal of engineering and applied sciences*. Citeseer. 2015, vol. 10, no. 13, p. 5551–5556.

[13] Standards, N. I. of and Technology. *Announcing Request for Candidate Algorithm Nominations for a New Cryptographic Hash Algorithm (SHA-3) Family.* E7-21581. 2007.
Https://www.govinfo.gov/content/pkg/FR-2007-11-02/pdf/E7-21581.pdf.

[14] Stevens, M., Bursztein, E., Karpman, P., Albertini, A. and Markov, Y. The first collision for full SHA-1. In: Springer. *Advances in Cryptology–CRYPTO 2017: 37th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 20–24, 2017, Proceedings, Part I 37.* 2017, p. 570–596.

[15] Wang, X., Yin, Y. L. and Yu, H. *Collision search attacks on SHA1.* Feb, 2005.

Appendix A

Contents of the included storage media

- xpupak01.pdf - Thesis text
- tex/ - LaTeXsource files
- src/ - source files and data