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 AVAILABLE GPUS FOR PASSWORD CRACKING PURPOSES

ANALÝZA DOSTUPNÝCH GPU PRO ÚČELY LÁMÁNÍ HESEL

BACHELOR'S THESIS
BAKALÁŘSKÁ PRÁCE

AUTHOR   **RICHARD HRMO**
AUTOR PRÁCE

SUPERVISOR   **Ing RADEK HRANICKÝ, Ph.D.**
VEDOUCÍ PRÁCE

BRNO 2023

Bachelor's Thesis Assignment

147220

Institut:	Department of Information Systems (UIFS)
Student:	**Hrmo Richard**
Programme:	Information Technology
Specialization:	Information Technology
Title:	**Analysis of Available GPUs for Password Cracking Purposes**
Category:	Security
Academic year:	2022/23

Assignment:

1. Study the Hashcat password cracking tool. Learn the principles of using GPUs to accelerate password attacks with the OpenCL technology.
2. Explore supported attack modes and cryptographic algorithms in Hashcat.
3. Conduct a survey of GPUs available on the market. Focus on NVIDIA and AMD products. Study different model series and describe their characteristics.
4. In consultation with the supervisor, design a tool to automatically run password attacks under different configurations of Hashcat. The tool will implement appropriately chosen model jobs and collect information about the computation process.
5. Implement the proposed tool.
6. Use the tool to measure computing characteristics of cracking jobs on different GPUs. Monitor the computational performance, processor utilization, memory requirements, etc. Document any error messages produced by the OpenCL subsystem.
7. Thoroughly analyze the obtained data and identify critical points of the given graphics card series. Evaluate the results and discuss your conclusions.

Literature:

- Hranický R.: Digital Forensics: The Acceleration of Password Cracking. *Ph.D. thesis*. Faculty of Information Technology, Brno University of Technology. 2022.
- Munshi, A.: The OpenCL specification. In Proceedings of the 21st IEEE Hot Chips Symposium (HCS). Standford, CA, USA. August 2009. pp. 1–314
- Sprengers, M.: "GPU-based password cracking." *On the Security of Password Hashing Schemes regarding Advances in Graphics Processing Units, Radboud University Nijmegen* (2011).
- Murakami, T.; Kasahara, R.; Saito, T.: An implementation and its evaluation of password cracking tool parallelized on GPGPU. *In Proceedings of the 10th International Symposium on Communications and Information Technologies (SoICT)*. Hanoi, Vietnam. October 2010. pp. 534–538.

Requirements for the semestral defence:
Points 1 to 4

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

Supervisor:	**Hranický Radek, Ing., Ph.D.**
Head of Department:	Kolář Dušan, doc. Dr. Ing.
Beginning of work:	1.11.2022
Submission deadline:	31.7.2023
Approval date:	5.1.2023

Abstract

This thesis aims to determine which hardware parameters of graphic cards are the most important for password-cracking purposes. It includes a theoretical study of password cracking and an examination of the different attack methodologies. The thesis also contains a survey of currently available GPUs and their hardware parameters. Further, we look at the hashcat tool, which we will use for testing GPUs. Then the thesis explains the design of a tool for measuring password-cracking performance, which uses hashcat as its core and explains the implementation. In the analysis part, we look at the collected data, analyse them, study their similarities and opposites and determine which hardware characteristics of GPUs are the most important for good password-cracking performance.

Abstrakt

Táto práca sa zameriava na určenie najdôležitejších hardvérových parametrov grafických kariet pre účely lámania hesiel. Zahŕňa teoretickú štúdiu ohľadom lámania hesiel a preskúmanie rôznych metodík útokov. Práca obsahuje aj prehľad aktuálne dostupných GPU a ich hardvérových parametrov. Ďalej sa zaoberáme nástrojom hashcat, ktorý budeme používať na testovanie GPU. Ďalej práca vysvetľuje návrh nástroja na meranie výkonu pri lámaní hesiel, ktorého jadrom je hashcat a vysvetľuje jeho implementáciu. V analytickej časti sa zaoberáme zozbieranými údajmi, analyzujeme ich, skúmame ich podobnosti a protiklady a určujeme, ktoré hardvérové vlastnosti GPU sú najdôležitejšie pre dobrý výkon pri lámaní hesiel.

Keywords

password cracking, cryptographic hashes, cryptographic security, GPU, graphic-processing unit, hashcat

Klíčová slova

lámanie hesiel, kryptografické haše, kryptografická ochrana, hashcat, grafická karta, GPU

Reference

HRMO, Richard. *Analysis of Available GPUs for Password Cracking Purposes*. Brno, 2023. Bachelor's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor Ing Radek Hranický, Ph.D.

Analysis of Available GPUs for Password Cracking Purposes

Declaration

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

<div align="right">

. . . .
Richard Hrmo
July 31, 2023

</div>

Acknowledgements

I want to thank my supervisor Radek Hranický for his support, great feedback and other help. I would also like to thank my parents and my sister for their immense mental support, without which I don't think I could finish the thesis.

Contents

List of Figures

Chapter 1

Introduction

With the increasing reliance on technology and the internet in our daily lives, password cracking has become one of the primary concerns for individuals and organisations. The ability to gain unauthorised access to a system or application by guessing or determining the password that is used to protect it can have serious consequences, including the theft of sensitive information, the disruption of critical services, and the compromise of the integrity of data.

As a result, many companies try to fight this cyber warfare with stronger cyber security, using hashes with cryptographic salt and cryptographic pepper and using more robust passwords. Nevertheless, every password can be cracked with enough time and enough resources. For this very reason, new technologies are being developed even right now.

Password cracking does not always have to be used in a malicious way. For example, many criminals use computer technology just like regular businesses. They may have their own databases, encrypted messages, etc. Suppose law enforcement officers compromise devices with such content and want to use these devices for digital forensics (tracing traces of criminal activity). In that case, they might need to use password cracking to access valuable information.

Another side to password cracking is password recovery, which allows users to regain access to their accounts and systems in the event that they forget or lose their passwords. However, we do not always need to perform a password cracking attack in such cases.

To grasp what the password cracking performance might be like, we need to test it. And that is what this thesis is about. This thesis aims to analyse the performance of password cracking on graphic processing units (GPUs) and explore various factors that might influence the speed and effectiveness of these attacks. We will examine how GPUs from different manufacturers with different specifications compare to each other in terms of their suitability for password cracking. We will also compare the critical differences between GPUs and central processing units (CPUs). Finally, we will also examine different algorithms and cracking techniques and their impact on password cracking performance on different GPUs.

Overall, this thesis aims to provide a comprehensive overview of the performance of password cracking on GPUs, the key differences between the use of GPU and CPU and the difference in performance with the use of different password cracking techniques.

This thesis is divided into four main parts. Chapter 2 explains the essentials of password cracking, candidate password generation and verification. Chapter 3 describes GPUs, their use for general computing, why they are better than CPUs for password cracking purposes, acceleration of cracking attacks and the analysis of current available GPUs. Chapter 4 tells

us about hashcat, the attack modes it supports, and the hash families it can crack. Chapter 5 shows us the design of a tool for password cracking analysis on GPUs, what kind of input and output files it will use, what parameters will be measured, what information will be collected and the design of tests. We look at how the tool was implemented in Chapter 6. Chapter 7 explains the realisation of tests and the hashcat configurations that were used. Chapter 8 shows us how we analysed collected data from tests, what approach we used during analysis and why we used them, and the analysis results. Finally, everything is summed up in the Chapter 9.

Chapter 2

Password Cracking

This Chapter describes password cracking, the types of attacks, and how each attack works.

2.1 Password cracking essentials

Password cracking is a process of attempting to obtain unauthorised access to protected content by force. There are two types of password cracking attacks: offline and online attacks.

- **Online attack[11]:** When the attacker attacks a live system, we call it an online attack. The attacker creates the candidate passwords, and the live system verifies them. An online attack could include attacking a website or email account or breaking electronic locks. Shortcomings of such attacks can be a limited number of login attempts.

- **Offline attack[4]:** When the attacker has access to the password hash, we call it an offline attack. The attacker creates passwords and also verifies them on their own machines. An offline attack could include breaking into a stolen hard disk drive, word document, or PDF file.

Password cracking consists of two phases: password generation and password verification. We will look at these phases in the following sections [9].

2.2 Password Generation

The first step of password cracking is generating the candidate passwords. The password generation may use existing string fragments or entirely new ones from a pre-defined set of characters or their combination. More innovative methods also use mathematical probability and statistics. An attack mode or attack type defines the creation of candidate passwords. The attack configuration further specifies the creation of candidate passwords. For example, it can limit the length of strings, use of numbers, capital letters and other details [9].

2.2.1 Brute-force Attack

The main principle of brute-force attacks is the exhaustive search. This type of attack uses one or more alphabets and a series of rules which define how to build strings from them. An

alphabet is an ordered set of characters used for creating passwords. The classic incremental brute-force attack creates every possible sequence of characters of a given length from a single alphabet. More advanced brute-force attacks may use additional rules that can, for example, specify what characters are used in which position. The main advantage of a brute-force attack is that it eventually finds the correct password. The main disadvantage is that finding the correct password can take a long time, depending on the number of candidate passwords [8].

2.2.2 Dictionary-based Attack

Dictionary-based attacks use one or more wordlists of strings in which each line represents a candidate password, which means that generating password guesses is just reading a text file line by line. Password candidates can be further modified, for example, by substituting some characters for others or making some characters capital. Password-mangling rules define such modifications [17].

2.2.3 Probabilistic Methods

Nowadays, advanced password guessing techniques often employ the use of statistics and mathematical probability. In addition, these methods can utilise information about the password creator, like the country of origin, language and other personal details. That is why these methods are highly efficient against human-created passwords[18].

2.3 Password verification

The second step of password cracking is password verification. In this step, we verify whether the candidate password is correct. The method of password verification depends entirely on what kind of password we try to crack. As defined by Radek Hranický, password verification can be classified into three password verification schemes: hash-based, decryption-based and checksum-based password verification.

2.3.1 Hash-based Password Verification

The hash-based password verification is the simplest of all scenarios, where we have access to the hash of the correct password. Web applications and operating systems usually store passwords in a hashed form. When logging into the system, the application calculates the cryptographic hash of the inputted password and compares it to the stored one. If these hashes match, the user is allowed access to protected resources [9].

The verification process for password cracking is the same. First, we take the candidate password and create its cryptographic hash. After that, we compare this hash to the correct one. If these hashes match, we found the correct password.

2.3.2 Decryption-based Password Verification

When no verification value is stored with the password hash, we can perform a known-plaintext attack. To do this, we first need to get an encryption key. The process to get the key is defined by the protected media format's manufacturer. Once we get the key, we can decrypt the encrypted content. After the decryption, we can check for the expected

string, and if it is there, we have found the correct password. To automate the process of decryption-based password verification, we need to know a part of plaintext [9].

2.3.3 Checksum-based Password Verification

In cases where there is no verification value nor a known part of the plaintext, we can use checksum-based password verification. The protected media can include a checksum of its content, which we can compare to our own generated checksum. First, we need to generate an encryption key, just like in decryption-based password verification. After that, we decrypt the content or its part and calculate the checksum from the plaintext. We found the correct password if the result is identical to the known checksum [9].

2.4 Hash Functions

As we already discussed, hashes are a big part of cyber security and password cracking. Different hashes provide different attributes, some being more easily cracked than others. Generally speaking, we can divide hash functions into two categories, cryptographic and non-cryptographic [7] hash functions. However, for the sake of this thesis, we will only talk about cryptographic hash functions [2]. A good hash function should maintain a few properties, them being:

- **Uniformly distributed** - A perfect hash function produces unique output for every unique input.

- **Deterministic** - The hash function produces the same output for any specific input.

- **Low complexity** - It is easy to compute the hash value for any given input.

A good cryptographic hash function needs to have a few more attributes. Without these attributes, we could easily crack the hash and access the protected content. These attributes are:

- **Pre-image resistance**: It should be hard to find a message from which the hash has been created using the hash function. It should also be hard to find two different inputs from which the output hash is the same.

- **Collision resistance**: There should not be two identical hashes after using the hash function on two different inputs. „Such a pair is called a cryptographic hash collision [16]."

2.5 Password Cracking Tools

There are many password cracking tools, some better than others. A few of the most popular are John the Ripper[1], Hashcat[2], L0phtcrack[3], Cain and Abel, RainbowCrack[4] and Hydra[5]. There are commercial password cracking tools, like L0phtcrack, but there are

---

[1] https://www.openwall.com/john/
[2] https://hashcat.net/hashcat/
[3] https://gitlab.com/l0phtcrack
[4] http://project-rainbowcrack.com/
[5] https://github.com/vanhauser-thc/thc-hydra

also free, open-source solutions, like hashcat. These password cracking tools differ in their cracking performance and features; some use a graphical user interface, and some do not. It would be tough to compare every single password cracking tool, but because of its features and cracking performance, in this thesis, we talk about and use hashcat.

Chapter 3

Graphic Processing Unit

The graphic processing units, also known as GPUs, were first designed to accelerate the rendering of 3D graphics. However, in recent years GPUs are also utilised to parallelize processing of general purpose and not only image computing. Improvement of performance by effective use of GPU leads to the use of GPUs in fields other than just computer graphics. One of these fields is password cracking. In this Chapter, we will look at the use of GPUs in password cracking, the feats of using GPUs over CPUs and currently available GPUs and their main differences [14].

3.1 Difference Between Integrated and Discrete GPUs

There are two types of GPUs, integrated and discrete ones. The main difference between these two is that while an integrated GPU is just a chip built into the processor, a discrete GPU is its own card and is separated from the CPU. An integrated GPU shares resources with the CPU, and memory, for example, and a discrete GPU does not.

Because a discrete GPU is its own board and is separate from the CPU, it provides much more performance but also draws more power and generates more heat. In password cracking, where we mostly care about performance, we prefer discrete GPUs over integrated ones, which is why I do not take integrated GPUs into account in this thesis.

3.2 General-purpose Computing on GPUs

Offloading general processing from CPU to GPU is called General-Purpose computing on Graphics Processing Units (GPGPU). As the computer graphic processing technology grows and new different applications are created, programmable GPU was developed to meet the demand for various graphic applications, which enables us to apply GPU for general processing. A chip of a GPU has many cores, which can perform to compute independently. This is why GPUs are very suitable for processing parallel tasks. [14, 16].

OpenCL was developed for the purpose of general-purpose computing. It allowed programmers to write code that would be directly run on top of the GPU without any issues. NVIDIA developed their own software for developing a program on their GPUs called CUDA. CUDA provides an environment for implementing GPU program code. The CUDA driver operates directly on top of GPU hardware. In addition, the CUDA's runtime library supports the usage of GPU [14].

3.3 GPU vs CPU

We can perform password cracking on both CPU and GPU. However, while the CPU can process many tasks fast sequentially, the GPU is much better suited for parallel computing [12]. This is because the main difference between GPU and CPU is the number of cores. While a CPU usually does not exceed 64 cores, the GPU has a much larger number of cores, ranging from hundreds to thousands. In addition, each processing unit also differs in what kind of memory it uses. While a CPU uses random-access memory (RAM), GPU uses video random-access memory (VRAM), which is located right on the GPU card.

We can see the GPU and CPU architecture comparison in the Figure 3.1. The typical GPU and CPU consist of the same components [16]:

- **Arithmetic logic unit (ALU)**, which performs all the logical, arithmetic and shift operations. In a GPU, this unit can also be called a „thread processor" or a „stream processor."

- **Control Unit (CU)**, which controls the operations of the processor. For example, it controls the order of operations.

- **Memory** - Either **RAM** or **VRAM**, depending on the type of processing unit.

- **Cache memory** is a very fast but small-capacity memory that functions as a buffer between the processing unit and (V)RAM.
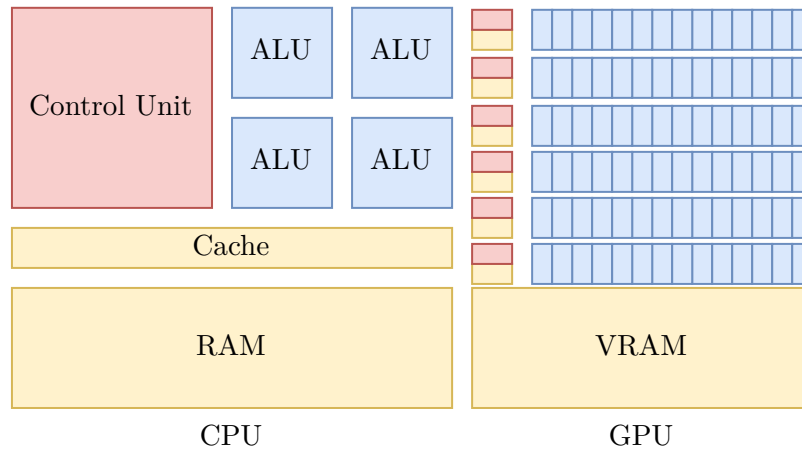
Figure 3.1: A comparison of CPU and GPU architecture.

3.4 Acceleration of Cracking Attacks

As the passwords get more robust and their encryption gets more complicated, it takes much more performance to crack them. To overcome this issue, we can use GPU-accelerated computing. GPU-accelerated computing uses a GPU alongside a CPU for better performance. Nowadays, this has been utilised in many computing areas that need high-intense computing performance.

3.4.1 OpenCL

OpenCL, also known as Open Computing Language, is a framework for parallel programming that includes API, libraries, a programming language and a runtime system to support software development. With the use of OpenCL, a programmer can write general-purpose programs that execute on GPU without the need to map their algorithms onto a 3D graphics API [13].

The OpenCL C programming language is based on the ISO/IEC 9899:1999 C programming language [13]. It can run on every modern GPU. We can compute parallelly on GPUs using OpenCL C and thus use their excellent computing power for general-purpose computing.

3.4.2 CUDA

CUDA - a parallel computing platform and programming model developed by NVIDIA. Just like with OpenCL, with CUDA, we can build GPU-accelerated applications. The CUDA toolkit includes GPU-accelerated libraries, debugging and optimisation tools, a C/C++ compiler, and a runtime library[1].

3.4.3 Comparison of CUDA and OpenCL

CUDA and OpenCL have similar functionality, and using NVIDIA's development tools, porting the kernel code from one to the other requires minimal changes. However, when comparing the two, CUDA performs better when transferring data from and to the GPU. CUDA's kernel execution also performs consistently faster than OpenCL's, even though the implementations run nearly identical code. Choosing CUDA over OpenCL would be wiser in password cracking, which is also projected in modern password cracking applications like hashcat [10].

3.5 Analysis of available GPUs

NVIDIA and AMD currently dominate the market. Both of them are well-known companies in the computer hardware sphere and have been around for years. Their newest competitor is Intel, which recently got back into GPU design and manufacturing, but has also been around in the computer hardware sphere for years.

3.5.1 NVIDIA

Jensen Huang, Chris Malachowsky and Curtis Priem founded NVIDIA in 1993. In 1999 NVIDIA invented the first graphics processing unit called GeForce 256[2]. Since then, NVIDIA has created many graphics cards that people with all kinds of computer machines use, ranging from low-cost computers to data centre servers. We will look at the GPU series ranging from 10 series up to 40 series.

---

[1]https://developer.nvidia.com/cuda-toolkit
[2]https://www.nvidia.com/en-us/about-nvidia/corporate-timeline/

10 Series

The 10 Series was released in 2016, and the GPUs are based on the Pascal architecture. The flagship of the 10 series was the Geforce GTX 1080 Ti graphics card. It was the most powerful card available on the market at the time of its release, having as much as twice the power of the following most powerful GPU in line, the Geforce GTX 1080. GeForce GTX 1080 Ti even went toe-to-toe with the next generation. In the Table 3.1 is the comparison of all 10 series graphics cards[3]. Interestingly, the number of Cuda cores in the GTX1080Ti is five times greater than in the GTX1050, which is not usually the case. Also, the Memory differences between each graphics card vary a lot.

	Geforce GTX 1080Ti/1080	Geforce GTX 1070Ti/1070	Geforce GTX 1060 (6GB/3GB)	Geforce GTX 1050Ti/1050
Architecture	Pascal	Pascal	Pascal	Pascal
Cuda Cores	3584 / 2560	2432 / 1920	1280 / 1152	768 / 640
Base Clock (GHz)	1.48 / 1.61	1.61 / 1.51	1.51	1.29 / 1.35
Boost Clock (GHz)	1.58 / 1.73	1.68	1.71	1.39 / 1.46
Standard Memory Config	11GB GDDR5X/ 8GB GDDR5X	8GB GDDR5	6GB GDDR5/3GB GDDR5	4GB GDDR5/2GB GDDR5
Memory Interface Width	352-bit / 256-bit	256-bit	192-bit	128-bit

Table 3.1: Comparison of NVIDIA GTX 10 Series GPUs

20 Series

The Turing architecture is the core of 20 Series GPUs. Released in 2018, ranging from 1920 CUDA Cores to 4352 CUDA cores, the 20 series almost doubled in performance compared to the previous generation, except for GeForce GTX 1080 Ti. In addition, the 20 Series was the first series with ray tracing, allowing for much better image quality. See the Table 3.2 for an exact comparison between the 20 Series graphics cards[4]. Compared to the previous generation, the number of Cuda cores did not change much, but every GPU has at least 8GB of memory, and the processors are running at a bit faster clock speeds. Because of this, the memory interface width also increased in some GPUs.

---

[3]https://www.nvidia.com/en-eu/geforce/10-series/
[4]https://www.nvidia.com/en-eu/geforce/graphics-cards/compare/?section=compare-20

	Geforce RTX 2080Ti/2080	Geforce RTX 2080 Super	Geforce RTX 2070 Super/ 2070	Geforce RTX 2060 Super/ 2060 (12/6GB)
Architecture	Turing	Turing	Turing	Turing
Cuda Cores	4352 / 2944	3072	2560 / 2304	2176 / 2176 / 1920
Base Clock (GHz)	1.35 / 1.52	1.65	1.61/1.41	1.47 / 1.47 / 1.37
Boost Clock (GHz)	1.64 / 1.8	1.82	1.77 / 1.71	1.65 / 1.65 / 1.68
Standard Memory Config	11GB GDDR6/ 8GB GDDR6	8GB GDDR6	8GB GDDR6/ 8GB GDDR6	8GB GDDR6/ 12GB GDDR6/ 8GB GDDR6
Memory Interface Width	352-bit / 256-bit	256-bit	256-bit / 256-bit	256-bit / 256-bit / 192-bit

Table 3.2: Comparison of NVIDIA RTX 20 Series GPUs [6]

16 Series

In 2019 NVIDIA introduced the 16 series, which is based on Turing architecture. Manufactured at the same time as the 20 Series, the 16 Series was supposed to fill the entry-level to the mid-range gap. The Table 3.3 shows us a comparison between each card of this series[5]. Even though this should have been an upgrade to 10 series low to midrange GPUs, we can see that the number of Cuda cores in some graphics is smaller. The memory was also not upgraded, and neither was the memory interface. Interestingly, the clock speeds are faster.

---

[5]https://www.nvidia.com/en-eu/geforce/graphics-cards/compare/?section=compare-16

	Geforce GTX 1660Ti/1660	Geforce GTX 1660 Super	Geforce GTX 1650 Super/ 1650 (G5/G6)	Geforce GTX 1630
Architecture	Turing	Turing	Turing	Turing
Cuda Cores	1536 / 1408	1408	1280 / 896 / 896	512
Base Clock (GHz)	1.5 / 1.53	1.53	1.53 / 1.49 / 1.41	1.74
Boost Clock (GHz)	1.77 / 1.79	1.79	1.73 / 1.67 / 1.59	1.7
Standard Memory Config	6GB GDDR6/ 6GB GDDR5	6GB GDDR6	4GB GDDR6/ 4GB GDDR5/ 4GB GDDR6	4GB GDDR6
Memory Interface Width	192-bit / 192-bit	192-bit	128-bit / 128-bit / 128-bit	64-bit

Table 3.3: Comparison of NVIDIA GTX 16 Series GPUs [6]

30 Series

30 Series, released in 2020, was a big jump in performance. The 30 Series GPUs had over two times more performance than the previous generation. The core of 30 Series GPUs is the Ampere architecture. The flagship of the 30 Series, the Geforce RTX 3090 Ti, reaches incredible 10752 CUDA cores. For more information about each graphic card of the 30 Series[6], see the Table 3.4 and Table 3.5. We can see that it is now a standard to have at least 8 GB of memory, whereas the RTX 3090Ti and RTX 3090 have an incredible 24 GB. Because of this, the memory interface also got larger. Interestingly, the clock speeds did not get faster but stayed the same or got slightly slower. One more GPU belongs to this series, even though its name is different, the A4000[7]. This GPU uses the same Ampere architecture but was designed only to use a single slot and be a small GPU for smaller computer builds. It is further described in the Table 3.5.

---

[6]https://www.nvidia.com/en-eu/geforce/graphics-cards/compare/?section=compare-specs
[7]https://www.nvidia.com/en-us/design-visualization/rtx-a4000/

	Geforce RTX 3090Ti/3090	Geforce RTX 3080Ti/3080 (12/10GB)	Geforce RTX 3070Ti/3070	Geforce RTX 3060Ti
Architecture	Ampere	Ampere	Ampere	Ampere
Cuda Cores	10752 / 10496	10240 / 8960 / 8704	6144 / 5888	4864
Base Clock (GHz)	1.67 / 1.40	1.37 / 1.26 / 1.44	1.58 / 1.50	1.41
Boost Clock (GHz)	1.86 / 1.70	1.67 / 1.71 / 1.71	1.77 / 1.73	1.67
Standard Memory Config	24GB GDDR6X/ 24GB GDDR6X	12GB GDDR6X/ 12GB GDDR6X/ 10GB GDDR6X	8GB GDDR6X/ GDDR6	8GB GDDR6/ 8GB GDDR6X
Memory Interface Width	384-bit / 384-bit	384-bit / 384-bit / 320-bit	256-bit / 256-bit	256-bit

Table 3.4: Comparison of NVIDIA RTX 30 Series GPUs [6]

	Geforce RTX 3060 (12 / 8 GB)	Geforce RTX 3050 (8 GB / OEM)	RTX A4000
Architecture	Ampere	Ampere	Ampere
Cuda Cores	3584 / 3584	2560 / 2304	6144
Base Clock (GHz)	1.32 / 1.32	1.55 / 1.51	0.74
Boost Clock (GHz)	1.78 / 1.78	1.78 / 1.76	1.56
Standard Memory Config	12 GB GDDR6 / 8 GB GDDR6	8 GB GDDR6 / 8 GB GDDR6	16GB GDDR6
Memory Interface Width	192-bit / 128-bit	128-bit / 128-bit	256-bit

Table 3.5: Comparison of NVIDIA RTX 30 Series GPUs [6]

40 Series

The most recent and powerful Series NVIDIA produced is the 40 Series, released in 2022. Based on the Ada Lovelace architecture, NVIDIA claims[8] that their 40 Series cards have up to two times more performance than their predecessor, the 30 Series. In addition, Nvidia GeForce RTX 4090 is the most powerful consumer GPU currently available. The Table 3.6 shows the comparison of 40 Series graphic cards[9]. The RTX 4090 has a more significant number of Cuda cores than the previous generation. It is also interesting that we got faster clock speeds after a few generations.

---

[8]https://www.nvidia.com/en-eu/geforce/graphics-cards/40-series/rtx-4090/
[9]https://www.nvidia.com/en-eu/geforce/graphics-cards/compare/?section=compare-40

	Geforce RTX 4090	Geforce RTX 4080
Architecture	Ada Lovelace	Ada Lovelace
Cuda Cores	16384	9728
Base Clock (GHz)	2.23	2.21
Boost Clock (GHz)	2.52	2.51
Standard Memory Config	24 GB GDDR6X	16 GB GDDR6X
Memory Interface Width	384-bit	256-bit

Table 3.6: Comparison of NVIDIA RTX 40 Series GPUs [6]

3.5.2 AMD

AMD was founded in 1969[10]. Since then, the company has become one of the leaders in the computer hardware manufacturing industry. AMD manufactured their first graphics programming unit in 2000, the Radeon R100. Since then, they have evolved alongside NVIDIA and have also delivered powerful GPUs throughout the years. We will look at the GPU series ranging from the Radeon R9 300 Series to the AMD Radeon RX 7000 Series.

Radeon R9 200, Radeon R9 300, Radeon R9 Fury Series and Radeon RX 400 Series

Radeon R9 300 and Radeon R9 Fury Series were both released in 2015. Their predecessor Radeon R9 200 Series, was released two years before that. They were the go-to solution for budget graphics cards. Released in 2016, the Radeon R9 400 Series was a budget option compared to NVIDIA's GTX 10 Series. The performance was lacking compared to AMD's previous GPU Series.

Radeon RX 500 and Vega Series

While remaining the budget option on the market, the RX 500 Series[11] delivered more performance than the previous AMD GPU series. The RX 500 Series was released in 2017, and its core was the 4th Gen GCN Architecture. Released in the same year as the RX 500 Series, the Vega Series offered more power for more price. The RX Vega-64[12] GPU could hold its own compared to the NVIDIA GTX 1080, which was NVIDIA's 2nd most powerful consumer-grade GPU at the time, and the RX Vega-56[13] was not left in the dust either. For a comparison of each GPU in these series, see the Table 3.7 and the Table 3.8. Interestingly, Vega series GPUs have a 2048-bit memory interface width, much larger than any other GPU. They also have a large number of stream processors and 8GB of ram.

---

[10]https://www.amd.com/en/corporate.html

[11]https://www.amd.com/en/RX-series

[12]https://www.amd.com/en/products/graphics/radeon-rx-vega-64

[13]https://www.amd.com/en/products/graphics/radeon-rx-vega-56

	Radeon RX 580	Radeon RX 570	Radeon RX 560	Radeon RX 550
Architecture	4th Gen GCN	4th Gen GCN	4th Gen GCN	4th Gen GCN
Stream Processors	2304	2048	896/1024	512
Base Clock (GHz)	1.26	1.17	1.18	1.1
Boost Clock (GHz)	1.34	1.24	1.28	1.18
Standard Memory Config	8GB GDDR5	8GB GDDR5	4 GB GDDR5	4 GB GDDR5
Memory Interface Width	256-bit	256-bit	128-bit	128-bit

Table 3.7: Comparison of AMD RX 500 Series GPUs [1]

	Radeon RX Vega 64	Radeon RX Vega 56
Architecture	Vega	Vega
Stream Processors	4096	3584
Base Clock (GHz)	1.25	1.16
Boost Clock (GHz)	1.55	1.47
Standard Memory Config	8GB HBM2	8 GB HBM2
Memory Interface Width	2048-bit	2048-bit

Table 3.8: Comparison of AMD Vega Series GPUs [1]

Radeon RX 5000 Series

Released in 2019, the Radeon RX 5000 Series was the first to use the 7nm RDNA architecture. Even though NVIDIAs graphics cards still outperformed AMDs, it was a huge step. The Table 3.9 shows a comparison of each GPU of the series. Here we can see that some cards have the same amount of stream processors but different amounts of memory, clock speeds, and memory interface width. During the password cracking performance analysis, this can lead to some exciting finds.

	Radeon RX 5700XT/5700	Radeon RX 5600XT/5600	Radeon RX 5500XT/5500	Radeon RX 5300 XT 5300
Architecture	AMD RDNA	AMD RDNA	AMD RDNA	AMD RDNA
Stream Processors	2560 / 2304	2304 / 2048	1408	1408
Base Clock (GHz)	1.61 / 1.47	1.13	1.61 / 1.5	1.67/1.33
Boost Clock (GHz)	1.91 / 1.73	1.56	1.85	1.85/1.65
Standard Memory Config	8GB GDDR6	6GB GDDR6	8GB GDDR6/ 4GB GDDR6	4GB GDDR5/ 3GB GDDR6
Memory Interface Width	256-bit	192-bit	128 bit	96-bit

Table 3.9: Comparison of AMD RX 5000 Series GPUs [1]

Radeon RX 6000 Series

The Radeon RX 6000 Series[14] was released in 2021 as the successor to the Radeon RX 5000 Series. It was the first series that could compete with the NVIDIA RTX 30 Series, even on the top level. The core of the Radeon RX 6000 Series is the AMD RDNA 2 architecture. See the Table 3.10 and the Table 3.11 for the compassion of each graphics card of the series.

According to the tables, we can see that the AMD GPUs also jump on a standard minimum of 8GB of memory, except for Radeon RX 6400. Interestingly, Radeon RX 6500XT uses 8GB of memory, but only a 64-bit interface, while the other cards with the same memory use a 128-bit one. In other GPUs with a larger amount of memory, the interface width also gets larger. We can also see that the number of stream processors and the clock speeds got faster than in the previous generations.

	Radeon RX 6950XT/ 6900XT	Radeon RX 6800XT/6800	Radeon RX 6750XT	Radeon RX 6700XT/6700
Architecture	AMD RDNA2	AMD RDNA2	AMD RDNA2	AMD RDNA2
Stream Processors	5120	4608 / 3840	2560	2560 / 2304
Base Clock (GHz)	1.89 / 1.83	1.83 / 1.70	2.15	2.32 / 1.94
Boost Clock (GHz)	2.31 / 2.25	2.25 / 2.11	2.60	2.58 / 2.45
Standard Memory Config	16GB GDDR6	16GB GDDR6	12GB GDDR6	12GB GDDR6 / 10GB GDDR6
Memory Interface Width	256-bit	256-bit	192-bit	192-bit / 160-bit

Table 3.10: Comparison of AMD RX 6000 Series GPUs [1]

---

[14] https://www.amd.com/en/graphics/radeon-rx-graphics-6000-series

	Radeon RX 6650XT	Radeon RX 6600XT/6600	Radeon RX 6500XT	Radeon RX 6400
Architecture	AMD RDNA2	AMD RDNA2	AMD RDNA2	AMD RDNA2
Stream Processors	2048	2048 / 1792	1024	768
Base Clock (GHz)	2.06	1.97 / 1.63	2.31	1.92
Boost Clock (GHz)	2.64	2.59 / 2.49	2.82	2.32
Standard Memory Config	8GB GDDR6	8GB GDDR6	8GB GDDR6	4 GB GDDR6
Memory Interface Width	128-bit	128-bit	64-bit	64-bit

Table 3.11: Comparison of AMD RX 6000 Series GPUs [1]

Radeon RX 7000 Series

The newest GPU series made by AMD is Radeon RX 7000 Series[15]. Based on the AMD RDNA 3 architecture, the Radeon RX 7000 Series competes with the NVIDIAs RTX 40 Series on every level. The Table 3.12 shows the comparison of each graphics card from the Radeon RX 7000 series. The number of stream processors and the memory size got larger than in the previous generation, and the clock speeds also got faster.

	Radeon RX 7900 XTX	Radeon RX 7900 XT
Architecture	AMD RDNA 3	AMD RDNA 3
Stream Processors	6144	5376
Base Clock (GHz)	1.86	1.5
Boost Clock (GHz)	2.5	2.4
Standard Memory Config	24GB GDDR6	20 GB GDDR6
Memory Interface Width	384-bit	320-bit

Table 3.12: Comparison of AMD RX 7000 Series GPUs [1]

3.5.3 Intel

Intel got back into Discrete GPU manufacturing only recently. However, with their new Arc Series, they can compete in the midrange GPU market.

Intel Arc Series

Intel Arc Series GPUs are built using the Xe HPG architecture. Ranging from low-end Arc 3 GPUs up to high mid-range Arc 7GPUs, Intels GPUs bring the competition to the table. For more information about each specific graphics card, see the Table 3.13.

---

[15]https://www.amd.com/en/graphics/radeon-rx-graphics

	Intel Arc A770 (16/8GB)	Intel Arc A750	Intel Arc A380	Intel Arc A310
Architecture	Xe HPG	Xe HPG	Xe HPG	Xe HPG
Shading units	4096	3584	1024	768
Base Clock (GHz)	2.10	2.05	2.00	2.00
Boost Clock (GHz)	2.40	2.40	2.05	2.00
Standard Memory Config	16GB GDDR6 / 8GB GDDR6	8GB GDDR6	6GB GDDR6	4GB GDDR6
Memory Interface Width	256-bit	256-bit	96-bit	64-bit

Table 3.13: Comparison of Intel Arc Series GPUs [5]

Chapter 4

Hashcat

In this section, I describe the Hashcat tool and the attack modes it provides.

Hashcat is the self-proclaimed world's fastest and most advanced password recovery tool[1]. Jens „atom" Steube created Hashcat in 2009 as a freeware cracking solution with proprietary code. Since then, it has been made publicly available under an MIT licence and nowadays is an open-source project. The newest version combines the previous CPU-based hashcat, now called hashcat-legacy[2], and GPU-based oclHashcat[3].

Hashcat has consequently won the last four years of Crack me if you can contests organised by KoreLogic[4] and has placed in the top 2 positions during previous competitions. Based on that, the pure cracking performance of hashcat is its core benefit. Unfortunately, unlike many other cracking tools, hashcat has no graphical user interface (GUI) and can only be launched with a command-line interface, thus requiring a user to be advanced.

Over 70 users have contributed to the repository, and hashcat is currently being developed on GitHub. Hashcat also has a community of people that help others with its use and answer questions on their forum and other social media. Not only do hashcat developers provide the source code, but they also provide pre-compiled binaries for both Linux and Windows systems [9].

4.1   Attack Modes in Hashcat

As introduced in the Section, an attack mode represents the process behind creating candidate passwords. A dictionary attack and a brute-force attack are the two most knowns attack modes.

Hashcat supports the brute-force attack, the combinator attack, the dictionary attack, the hybrid attack, the mask attack, the rule-based attack, the toggle-case attack and the association attack modes.

Each attack mode is better suited for a different situation; to get the correct password, we must use the correct one. For example, we might use a brute-force attack on a password that is twenty characters long, which might take years to find. However, if the password consists of two worlds found in a dictionary, a dictionary attack would find it in no time. That is why choosing the correct attack mode is crucial in password cracking.

---

[1] https://hashcat.net/wiki/doku.php?id=hashcat
[2] https://hashcat.net/wiki/doku.php?id=hashcat-legacy
[3] https://hashcat.net/wiki/doku.php?id=oclhashcat
[4] https://contest.korelogic.com/

4.1.1 Brute-Force attack and Mask attack

The brute-force attack tries every possible combination of characters until it finds the correct password. While the tool will eventually find the correct password, this can be very time-consuming, depending on the length and complexity of the password. Because of this, it is better to use some optimisation, like the use of masks.

Mask attack is the only kind of brute-force attack hashcat provides. The reason behind this is to reduce the password candidate keyspace (the number of possible candidate passwords [15]) to a more efficient one and save the time it takes to crack the password. Therefore, there is no downside compared to the traditional brute-force attack.

„A password mask is a template defining allowed characters for each position in the password [9]." Masks in hashcat have the form of a string, which configures the keyspace for the password candidates using placeholders. A placeholder can be either a build-n charset variable, a static letter or a custom charset variable. A letter „?" followed by one of the built-in charset (for example, `l`, `u`, `s`, `d`, `a`, `b`) or one of the custom charset (1, 2, 3, 4) variable names indicates a variable. The Table 4.1 shows all of these charsets. For example, a custom charset consisting of the chars „0123456789abcdef" can be defined by the command `-1 ?dabcdef`. The mask is always the same length as the password, meaning that if we have a mask `?l?l?l?l` the password will only be four characters long. That is why we must repeat the attack several times to try cracking passwords of different sizes. The use of the „`-increment`" flag automates the process and increments candidate passwords after each iteration, begging at the length of 1 character and going all the way up to the length of the whole mask. The minimum and maximum length can also be set using the „`-increment-min`" and the „`-increment-max`" flags.

Symbol	Description	Charset
?d	digits	0123456789
?l	lower-case Latin alphabet	abcdefghijklmnopqrstuvwxyz
?u	upper-case Latin alphabet	ABCDEFGHIJKLMNOPQRSTUVWXYZ
?h	digits and first six lower-case characters from Latin alphabet	0123456789abcdef
?H	digits and first six upper-case characters from Latin alphabet	0123456789abcdef
?s	special characters	«space»!„#$%&'()*+,-./:;<=>?@[\]^_'{|}~
?a	lower-case and upper-case Latin alphabet and special characters	?l?u?d?s
?b	all values from 0x00 to 0xFF	
?1	custom charset number one	
?2	custom charset number two	
?3	custom charset number three	
?4	custom charset number four	

Table 4.1: Hashcat mask charset.

4.1.2   Dictionary Attack and Combinator Attack

The dictionary attack, also known as a "wordlist attack," or in the hashcat called "straight mode," is straightforward. The cracking tool reads a text file, also known as a dictionary or wordlist, line by line and tries each line as a password candidate.

The combinator attack takes every word of a dictionary and appends it to each word in a dictionary. So, for example, if we have a dictionary containing the words "goal, sleight, Peter," the cracking tool creates the following candidate passwords: goalsleight, goalPeter, sleightgoal, sleightPeter, Petergoal, Petersleight.

In hashcat, we must specify precisely two dictionaries in the command line. For example, a command could look like this: .
`hashcat64.exe -m 0 -a 1 hash.txt dictionary1.txt dictionary2.txt`. The `-m` argument sets the hash mode and the `-a` argument sets the attack mode, in this case, MD5 hash and combinator attack mode. We can also add rules to the dictionaries, which can be called rule-based attack[5]. It is one of the most complicated attack modes. Rule-based attack is like a programming language designed for password candidate generation. For example, these rules can be applied on top of wordlists. John the Ripper and PasswordsPro are other tools that use these rule-based attacks. The functions for creating rules are identical in these three password cracking tools, with the exception of new functions added to Hashcat. These new functions have unique names to avoid conflicts.

With the clever use of these rules, we can make the cracking performance much higher. If we know some information about the habits of the password creator, we can easily modify what kind of password candidates the hashcat will try. For example, if we know that the system requires the use of a capital letter and a number, we will not create candidate passwords without these.

4.1.3   Hybrid Attack

The hybrid attack is very similar to the combinator attack. It combines a dictionary with the result of a brute-force attack. In other words, the brute-force keyspace is either prepended or appended to every dictionary word. An attack like this can be used when we know part of the correct password to crack it faster. An example command for hashcat is: `hashcat64.exe -m 0 -a 6 dictionary.txt ?l?l?l`. To emulate a hybrid attack, we can also use so-called brute-force rules. We need to generate a rule and pass it to hashcat via the "-r bf.rule."

4.1.4   Toggle-Case Attack

The toggle-case attack is pretty straightforward. It creates all possible combinations of upper-case and lower-case variants for each candidate password in the wordlist. This attack was separate in the legacy hashcat, and in the current release of hashcat(6.2.6), we can emulate this attack using specialized rules[6].

4.1.5   Association Attack

We use the association attack when a likely password or password component is already known. It tries every single word in a single wordlist against a single hash. To use this

---

[5] https://hashcat.net/wiki/doku.php?id=rule_based_attack
[6] https://hashcat.net/wiki/doku.php?id=toggle_attack_with_rules

method, we must meet the requirements: the wordlist must be the same length as the target hash list, and if the target list has work factors, they must all be identical; for example, all bcrypts are cost 10 and cannot be cost 11[7]. In hashcat, we can also use rules with this attack.

4.2   Supported Hash Families

Hashcat supports several different types of hashing algorithm families[8], which can be grouped into the following categories:

- Message-Digest Algorithm (MD) family: This group includes popular algorithms such as MD4, MD5, and Half MD5. These algorithms create a hash from an input message of any size.

- Secure Hash Algorithm (SHA) family: This group includes algorithms such as SHA-1, SHA-224, SHA-256, SHA-384, and SHA-512. These algorithms create a hash from an input message and are considered more secure than the MD family.

- Key Derivation Function (KDF) family: This group includes algorithms such as bcrypt, scrypt, and PBKDF2-HMAC. These algorithms are used to create a cryptographic key from a password. They are considered more secure than the algorithms in the previous two groups, as they are designed to be more resistant to brute-force attacks.

- Keyed-Hash Message Authentication Code (HMAC) family: This group includes algorithms such as PBKDF2-HMAC-SHA1, PBKDF2-HMAC-SHA256, and PBKDF2-HMAC-SHA512.

- Cryptographic Algorithm family: This group includes algorithms such as ChaCha20. These algorithms encrypt or decrypt a given input message.

- Specific Hash Family: This group includes several specific hash functions that are used in specific systems or applications, such as WPA/WPA2, MS-SQL, MYSQL323, MYSQL4.1, POSTGRESQL, PDF 1.1 - 1.7, Office 2003-2013, Apple Secure Notes or Mac OS X.

---

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

Chapter 5

Design of a Tool for Password Cracking Analysis on GPUs

This Chapter describes the design of a tool for testing the password cracking performance of individual GPUs. It explains how the tool will work, what kind of information it will collect, and its input and output.

5.1   General Concept

To test GPUs' processing power and capability, all we need is the hashcat tool. However, running tests one by one manually would be time-consuming and ineffective. Because of this, we need a tool that will automate the process. The tool for testing GPUs will use the hashcat as its core. The tool will control the hashcat and its input, collect the output information, parse it, and save it into a file. The tool will run multiple tests on each GPU. The tests will be saved in configuration files, and the tool will parse these files and run the tests based on the input parameters saved in them. The collected information are further explained in the secion 5.8.

The figure 5.1 illustrates the general concept of the tool. First, the input files will contain the configuration of the hashcat and the hashes that will be cracked, which will be parsed and then passed to the hashcat input. This configuration will also contain the location for dictionaries and rulesets, which the hashcat will use. Dictionaries are wordlists that hashcat uses for password-cracking purposes, and rulesets contain rules that modify these wordlists. After that, the hashcat output and hardware info are collected for the duration of the hashcat running. Hardware info is other information about GPU hardware that hashcat does not provide. Parsed and saved into output files.
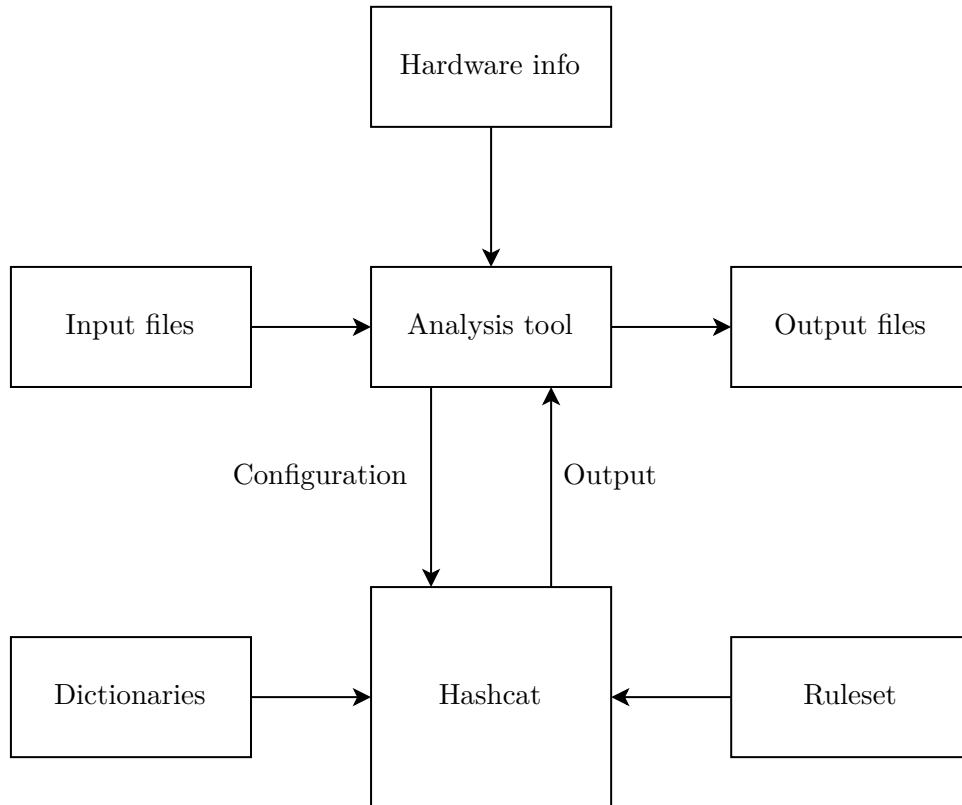
Figure 5.1: General concept of analysis tool.

5.2 Input Files

There are two kinds of input files, one that contains the hashes and one that contains the configuration. The tool will load up these files and then run the tests consecutively. The YAML language, in which the configuration files will be saved, is a data serialisation language that is easily readable by humans and programs. The YAML language is a superset of JSON language but is more suited for configuration files than JSON because humans can read it more easily, which is the reason why I have chosen one over the other.

Figure 5.2 shows the file containing hashes, which includes the hash-mode[1] number used in hashcat settings that specifies the type of hash used, for example, MD5, AESCrypt or TrueCrypt. Then there is the name of the hash and the hash that the hashcat will be cracking. These hashes are taken from hashcat example hashes in the hashcat wiki[2].

In the Figure 5.3, we can see an example of a configuration file. The attack-mode parameter specifies which attack mode is used, for example, brute-force or dictionary attack. The charset parameter specifies the charset used in the brute-force (mask) or hybrid attack types. The increment parameter specifies if the hashcat will use increment, and the increment-min and increment-max parameters specify the increment range in a mask. The dictionaries parameter specifies the path to the folder where the dictionaries are saved or to the actual dictionaries, and the ruleset parameter specifies the path to the file with rules.

---

[1]https://hashcat.net/wiki/doku.php?id=hashcat#options
[2]https://hashcat.net/wiki/doku.php?id=example_hashes

Figure 5.2: An example hash input file of analysis tool.

Figure 5.3: An example config input file of analysis tool.

5.3 Output Files

The program will create output files and then save the output information inside them. There will be separate output files with the test results for each test. In one output file, there will be data collected from hashcats output. This output file will also contain the error log and the settings with which the hashcat was launched.

Figure 5.4 shows us an example of an hashcat output file. The most crucial info which we will use in the analysis is the speed[hashes/s], temperature[°C] and utilisation[%]. We will save other hardware information in the second output file, like memory used[MB] and bus interface utilisation[%]. An example of this file is shown in the Figure 5.5.

Figure 5.4: An example of hashcat data output file from the analysis tool.

```
 1    MEMORY_MAX,MEMORY_USED,MEMORY_UTIL,BUS_UTIL
 2    16376.0,3624.66796875,22.134025212200783,60.0
 3    16376.0,5770.66796875,35.238568446201754,40.0
 4    16376.0,5770.66796875,35.238568446201754,97.0
 5    16376.0,5770.66796875,35.238568446201754,97.0
 6    16376.0,5770.66796875,35.238568446201754,97.0
 7    16376.0,5770.66796875,35.238568446201754,97.0
 8    16376.0,5770.66796875,35.238568446201754,97.0
 9    16376.0,5770.66796875,35.238568446201754,97.0
10    16376.0,5770.66796875,35.238568446201754,96.0
11    16376.0,5770.66796875,35.238568446201754,97.0
12    16376.0,5770.66796875,35.238568446201754,97.0
13    16376.0,5770.66796875,35.238568446201754,97.0
14    16376.0,5770.66796875,35.238568446201754,97.0
15    16376.0,5770.66796875,35.238568446201754,97.0
16    16376.0,5770.66796875,35.238568446201754,97.0
17    16376.0,5770.66796875,35.238568446201754,97.0
18    16376.0,5770.66796875,35.238568446201754,97.0
19    16376.0,5770.66796875,35.238568446201754,97.0
20    16376.0,5770.66796875,35.238568446201754,97.0
21    16376.0,5774.41796875,35.26146781112604,97.0
```

Figure 5.5: An example of other hardware information output file from analysis tool.

After the testing, the testing tool will be able to run with different settings that will create min, max, mean and median values of important collected data. These data will be stored in two types of files, one for each test and one for each type of attack mode. We can see how the CSV file header of these files will look like in the Table 5.1. The only difference is that in the attack mode files, there is one more item in the header: the name of the used hash.

GPU	Name of the GPU.
HASH	Name of the hash, only in the attack mode CSV.
CONFIG	The attack mode used.
SPEED_MIN	The lowest recorded speed.
SPEED_MAX	The highest recorded speed.
SPEED_MEAN	The mean value counted out of speed values.
SPEED_MEDIAN	The median value counted out of speed values.
UTIL_MIN	The lowest recorded utilisation of GPU.
UTIL_MAX	The highest recorded utilisation of GPU.
UTIL_MEAN	The mean value counted out of utilisation values.
UTIL_MEDIAN	The median value counted out of utilisation values.
TEMP_MIN	The lowest recorded temperature of GPU.
TEMP_MAX	The highest recorded temperature of GPU.
TEMP_MEAN	The mean value counted out of temperature values.
TEMP_MEDIAN	The median value counted out of temperature values.
MEM_USED_MIN	The lowest recorded memory usage of GPU.
MEM_USED_MAX	The highest recorded memory usage of GPU.
MEM_USED_MEAN	The mean value counted out of memory usage values.
MEM_USED_MEDIAN	The median value counted out of memory usage values.
MEM_UTIL_MIN	The lowest recorded memory utilisation of GPU.
MEM_UTIL_MAX	The highest recorded memory utilisation of GPU.
MEM_UTIL_MEAN	The mean value counted out of memory utilisation values.
MEM_UTIL_MEDIAN	The median value counted out of memory utilisation values.
BUS_UTIL_MIN	The lowest recorded memory bus utilisation of GPU.
BUS_UTIL_MAX	The highest recorded memory bus utilisation of GPU.
BUS_UTIL_MEAN	The mean value counted out of memory bus utilisation values.
BUS_UTIL_MEDIAN	The median value counted out of memory bus utilisation values.

Table 5.1: Explained header of CSV analysis files.

5.4   Hashcat

We have chosen hashcat as the core for this tool for multiple reasons. Reason number one is that hashcat is the fastest cracking tool currently available. The second reason is that hashcat is an application controlled through the command-line interface. Thus we can easily make a controller tool that will automate the password cracking process. The third reason being the hashcat also has a benchmark mode, showing us the theoretical maximum password cracking performance. Theoretical, because the actual cracking performance is usually lower. Finally, the hashcat has machine-readable output as well as JSON format output. Because of this, we can parse the results of the tests more efficiently and effectively.

The analysis tool will make use of both the benchmark mode and of cracking actual hashes. Firstly the benchmark mode will be run with the same hash mode as the actual test. Then we will run the test and compare the characteristics of these two measurements and see how the theoretical performance differs from the actual one. Unfortunately, the

benchmark output is not the same as the default status output, and it can not run for the same duration, but we can still collect cracking performance data and the data from hardware and compare what we have.

5.5 Measurement methodology

We will measure cracking performance, as well as other hardware information. The measurement methodology differs when using the benchmark mode and running the tests. When running the benchmark mode, we only collect all hashcat output information and save it to a file. When running a test, we will also collect all information from the hashcat output, but we skip the first 2 seconds of collected data because the cracking performance in these 2 seconds differs from the rest of the test. The test runs for an additional 10 seconds (or more, depending on the type of the attack), and the hashcat output data gathered during this time will be used for analysis. Other GPU parameters are another set of data collected alongside the hashcat output. These parameters are the maximum GPU memory, GPU memory utilisation, the used GPU memory, and the bus utilisation.

5.6 Design of the Tool

The tool will be an application controlled through the command-line interface. There will be no graphical user interface because it is not needed for the purpose of this tool. Figure 5.6 describes the workflow of how the program will work. The application will have two modes, one for running tests and collecting data and the other one for creating comparison data from the collected data. In the testing mode, the application will start reading the input files from a specified folder. After it loads in the first input file, it will parse it and load up the CSV file containing hashes. Then it sets up the tests based on the settings from an input file and hashes from the CSV file. Then the application will launch hashcat with these settings. After each test is collected, the toll will save the collected data.

The second mode, in which we can run the tool, will load up all the data it created during testing, and then it creates comparison data used during the analysis. It will create mean and median values from collected data and also show their min and max values.
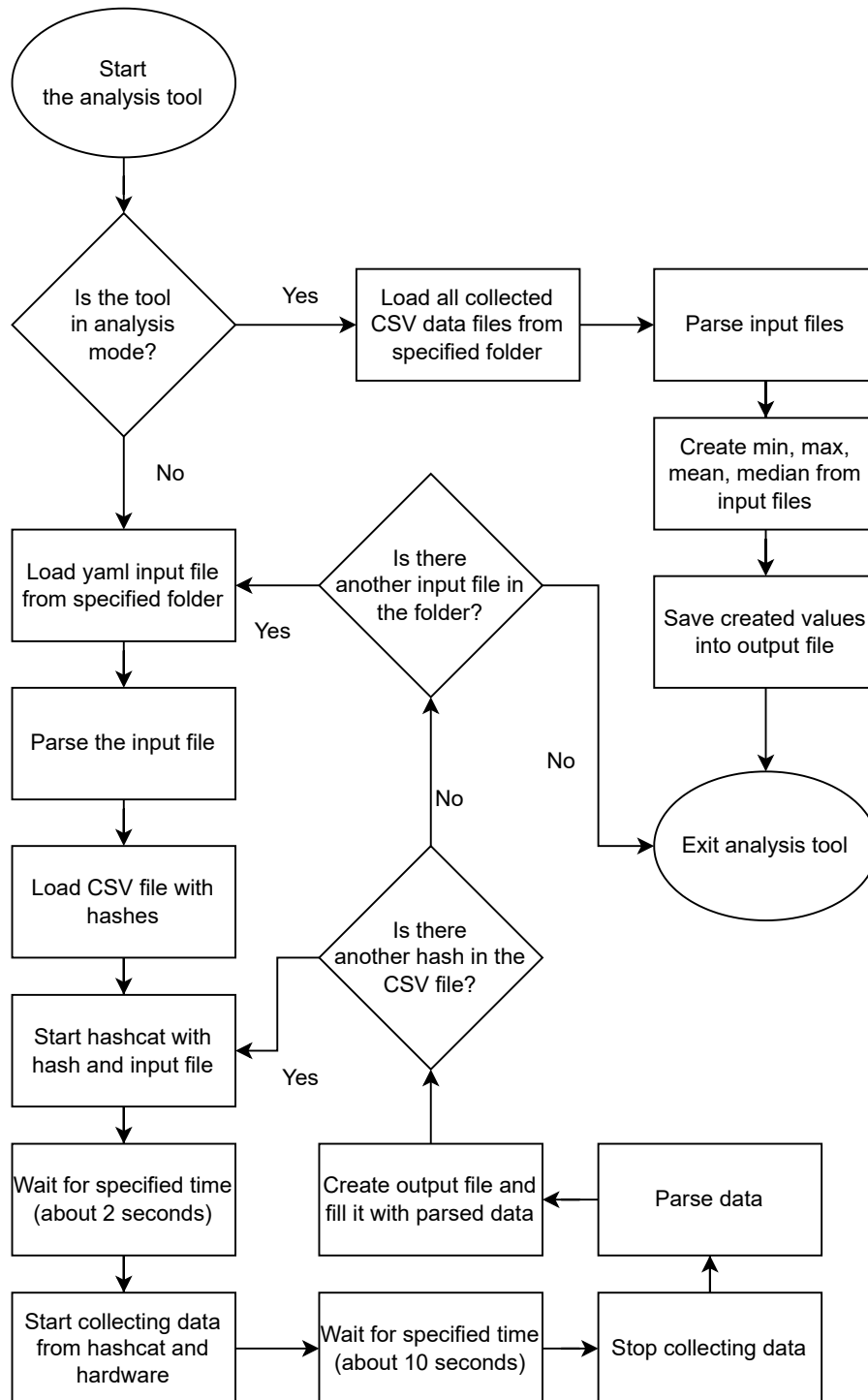
Figure 5.6: Workflow diagram of the analysis tool.
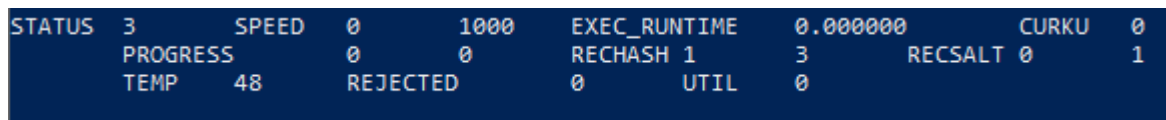
5.7 Tests

The tests will consist of hashes for every hash mode that we can currently run in hashcat. For each hash mode, there will be one hash that represents this hash mode. There will

be multiple tests for each hash. We will use the example hashes provided by hashcat on their website [3]. These tests will differ in attack mode, dictionaries and rules. We will have tests for combinator, dictionary, force (mask) and hybrid attacks. Each attack will have one configuration, except for a dictionary attack, where we will use different dictionaries and rules. Furthermore, the tests are described in the Chapter 7

5.8 Collected Information

The tool will collect different types of information about a GPU. First and foremost, the output of the hashcat tool's status will be collected. The program will collect this status in a machine-readable format[4], allowing us to work with it more easily. The figure 5.7 shows us what the output of status looks like. The STATUS field shows us if the hashcat tool is running (3), exhausted (5), cracked (6), aborted (7) or quit (8). The SPEED field shows us the number of hashes per unit of time and the unit of time in milliseconds per device. The EXEC_RUNTIME field shows us the execution runtime in seconds. The CURKU field shows us the restore point. The PROGRESS field shows us two values, the number of hashes that hashcat has tried so far and the number of hashes that remain. The RECHASH field shows us the number of recovered hashes. The RECSALT field shows us the number of recovered salts. The TEMP field shows us temperatures in Celcius per device. Finally, the REJECTED field shows us the number of incorrect passwords. We will not be using data from all of these fields, only the ones that matter in the performance comparison: the speed, the temp, and the utilisation.


```
STATUS   3        SPEED   0        1000     EXEC_RUNTIME   0.000000        CURKU   0
         PROGRESS        0        0        RECHASH 1       3       RECSALT 0       1
         TEMP    48       REJECTED        0       UTIL    0
```

Figure 5.7: An example output of Hashcat status.

The tool will also collect other information about graphic cards, which are memory and bus utilisation. The tool will collect this information using Python libraries or nvidia-smi[5] and rocm-smi[6]. Nvidia-smi, also known as NVIDIA System Management Interface, is a command-line utility. This utility allows us to see the information about memory usage - its total size and how much is being used, the utilisation of GPU, the fan speed, the temperature, the clock speeds and more. For the AMD GPUs, rocm-smi similarly collects information about GPUs.

The Collected information from hashcat is summarised in the Table 5.2. The status parameter is collected so that the analysis tool can ensure the hashcat is running correctly throughout the whole test length. The speed is the main parameter we collect and compare. It shows the actual cracking performance. Exec_runtime is collected so that we know how long the hashcat is running. The collected information from GPU is summarised in the table 5.3.

---

[3]https://hashcat.net/wiki/doku.php?id=example_hashes
[4]https://hashcat.net/wiki/doku.php?id=machine_readable
[5]https://developer.nvidia.com/nvidia-system-management-interface
[6]https://rocm.docs.amd.com/projects/rocm_smi_lib/en/latest/.doxygen/docBin/html/index.html

STATUS	What is the current status of hashcat.
SPEED	Shows us the cracking performance.
TEMPERATURE	The temperature of GPU.

Table 5.2: Collected information from Hashcat.

Total memory	Total memory size.
Used memory	Used memory size.
Bus Interface Usage	How much % of bus interface is used.

Table 5.3: Collected information from GPU.

After all the tests are done, and we no longer use hashcat, we calculate the minimum, maximum, mean and median values from these data. We save these data into two files, one for each hash and one for each config. Figure 5.8 shows an example of a file in which data are distributed based on configuration. Figure 5.9 shows us an example of a file where the data are distributed based on a hash. Both of these files have almost identical data. They are just distributed differently. We use these files later in the analysis. The headers (the first line) of these files are explained in the Table 5.1.

```
1   GPU,HASH,CONFIG,SPEED_MIN,SPEED_MAX,SPEED_MEAN,SPEED_MEDIAN,UTIL_MIN,UTIL_MAX,UTIL_MEAN,UTIL_MEDIAN,TEMP_MIN,
    TEMP_MAX,TEMP_MEAN,TEMP_MEDIAN,MEM_USED_MIN,MEM_USED_MAX,MEM_USED_MEAN,MEM_USED_MEDIAN,MEM_UTIL_MIN,MEM_UTIL_MAX,
    MEM_UTIL_MEAN,MEM_UTIL_MEDIAN,BUS_UTIL_MIN,BUS_UTIL_MAX,BUS_UTIL_MEAN,BUS_UTIL_MEDIAN
2   NVIDIA GeForce GTX 1080 Ti,1Password  cloudkeychain,dict2,8.0,13.0,9.78,9.0,87.0,93.0,91.39,92.0,52.0,57.0,53.39,
    53.0,2115.89,8501.79,7697.15,8436.54,18.78,75.48,68.33,74.9,1.0,80.0,63.04,74.0,
3   NVIDIA GeForce GTX 1080 Ti,7-Zip,dict2,2.0,2.0,2.0,2.0,75.0,79.0,76.67,77.0,57.0,59.0,57.28,57.0,1677.54,6681.15,
    6054.08,6677.84,14.89,59.31,53.75,59.28,0.0,86.0,62.71,76.0,
4   NVIDIA GeForce GTX 1080 Ti,AESCrypt(SHA256),dict2,105.0,127.0,115.62,114.5,57.0,67.0,63.62,64.5,58.0,60.0,58.62,58.
    0,1657.49,7667.54,6463.67,7664.79,14.71,68.07,57.38,68.05,0.0,93.0,48.6,60.0,
5   NVIDIA GeForce GTX 1080 Ti,AndroidBackup,dict2,117.0,123.0,120.43,121.0,25.0,87.0,76.29,84.0,59.0,61.0,59.86,59.0,
    1672.23,6606.27,5619.22,6605.77,14.85,58.65,49.89,58.65,1.0,98.0,62.2,85.0,
6   NVIDIA GeForce GTX 1080 Ti,AndroidFDE(SamsungDEK),dict2,193.0,193.0,193.0,193.0,61.0,64.0,62.5,62.5,60.0,60.0,60.0,
    60.0,1623.84,7618.38,5620.53,7617.38,14.42,67.63,49.9,67.63,0.0,88.0,42.78,69.0,
7   NVIDIA GeForce GTX 1080 Ti,AnsibleVault,dict2,73.0,80.0,76.33,75.5,37.0,68.0,64.08,67.0,58.0,60.0,58.58,58.0,1646.
    09,7657.2,6719.38,7635.95,14.61,67.98,59.65,67.79,2.0,100.0,58.46,72.0,
8   NVIDIA GeForce GTX 1080 Ti,AppleFileSystem(APFS),dict2,35.0,39.0,36.44,36.0,63.0,68.0,66.67,67.0,57.0,59.0,57.33,
    57.0,1664.6,6651.65,6148.08,6645.34,14.78,59.05,54.58,59.0,1.0,87.0,68.0,82.0,
9   NVIDIA GeForce GTX 1080 Ti,AppleiWork,dict2,451.0,451.0,451.0,451.0,60.0,60.0,60.0,60.0,61.0,61.0,61.0,61.0,1658.
    14,6382.79,4020.85,4021.61,14.72,56.67,35.7,35.7,0.0,98.0,28.83,0.0,
10  NVIDIA GeForce GTX 1080 Ti,AppleSecureNotes,dict2,35.0,39.0,36.5,36.0,63.0,76.0,67.61,67.0,57.0,59.0,58.0,58.0,
    1641.34,6628.32,6000.65,6622.01,14.57,58.85,53.27,58.79,3.0,72.0,57.71,69.0,
11  NVIDIA GeForce GTX 1080 Ti,AxCrypt1,dict2,149.0,152.0,150.6,151.0,25.0,71.0,58.4,65.0,59.0,60.0,59.6,60.0,1620.67,
    6183.2,6600.53,6177.07,14.39,54.89,41.38,54.84,0.0,98.0,48.78,69.0,
```

Figure 5.8: An example min/max/mean/median analysis tool output file.

```
1   GPU,CONFIG,SPEED_MIN,SPEED_MAX,SPEED_MEAN,SPEED_MEDIAN,UTIL_MIN,UTIL_MAX,UTIL_MEAN,UTIL_MEDIAN,TEMP_MIN,TEMP_MAX,
    TEMP_MEAN,TEMP_MEDIAN,MEM_USED_MIN,MEM_USED_MAX,MEM_USED_MEAN,MEM_USED_MEDIAN,MEM_UTIL_MIN,MEM_UTIL_MAX,
    MEM_UTIL_MEAN,MEM_UTIL_MEDIAN,BUS_UTIL_MIN,BUS_UTIL_MAX,BUS_UTIL_MEAN,BUS_UTIL_MEDIAN
2   NVIDIA GeForce GTX 1080 Ti,combinator,1862.0,1949.0,1901.0,1894.0,98.0,98.0,98.0,98.0,62.0,64.0,62.5,62.0,2193.25,
    7258.55,6695.77,7258.55,19.47,64.44,59.44,64.44,0.0,98.0,79.06,97.0,
3   NVIDIA GeForce GTX 1080 Ti,dict1,17.0,17.0,17.0,17.0,78.0,84.0,79.11,79.0,56.0,57.0,56.11,56.0,1594.56,6609.11,5976.
    0,6597.73,14.16,58.67,53.05,58.57,2.0,72.0,58.29,70.0,
4   NVIDIA GeForce GTX 1080 Ti,dict2,2.0,2.0,2.0,2.0,75.0,79.0,76.67,77.0,57.0,59.0,57.28,57.0,1677.54,6681.15,6054.08,
    6677.84,14.89,59.31,53.75,59.28,0.0,86.0,62.71,76.0,
5   NVIDIA GeForce GTX 1080 Ti,dict3,2465.0,2467.0,2466.33,2466.0,97.0,98.0,97.5,97.5,80.0,82.0,81.11,81.0,1152.25,6890.
    18,5884.72,6150.98,10.23,61.17,52.24,54.61,0.0,98.0,84.3,97.0,
6   NVIDIA GeForce GTX 1080 Ti,dict4,2855.0,2871.0,2858.89,2857.5,97.0,98.0,97.11,97.0,75.0,77.0,76.44,76.5,1148.1,6144.
    46,5492.95,6144.46,10.19,54.55,48.77,54.55,0.0,98.0,81.65,97.0,
7   NVIDIA GeForce GTX 1080 Ti,force,0.0,0.0,0.0,0.0,59.0,61.0,59.67,60.0,53.0,54.0,53.44,53.0,1144.35,6209.65,4270.84,
    6209.65,10.16,55.13,37.92,55.13,0.0,62.0,31.89,41.0,
8   NVIDIA GeForce GTX 1080 Ti,hybrid,1042.0,1226.0,1098.44,1077.0,98.0,99.0,98.67,99.0,55.0,56.0,55.11,55.0,1143.79,
    6210.84,4640.54,6210.84,10.15,55.14,41.2,55.14,0.0,98.0,55.84,84.0,
9   NVIDIA GeForce RTX 2080 SUPER,combinator,3049.0,3088.0,3065.54,3064.0,97.0,97.0,97.0,97.0,81.0,85.0,83.46,84.0,327.
    78,5371.68,3857.16,5371.68,4.0,65.57,47.08,65.57,0.0,80.0,52.22,77.0,
10  NVIDIA GeForce RTX 2080 SUPER,dict1,30.0,31.0,30.84,31.0,68.0,72.0,69.21,69.0,60.0,60.0,60.0,60.0,327.78,5305.68,
    4032.65,5305.68,4.0,64.77,49.23,64.77,0.0,73.0,51.56,72.0,
```

Figure 5.9: An example min/max/mean/median analysis tool output file.

Chapter 6

Implementation

In this Chapter, we will look at the implementation of the testing/analysis tool, what language it is implemented in, what libraries it uses and the reasons for these decisions. We also look at the implementation of scripts for displaying data in jupyter notebook that greatly helped in analysing the data.

6.1 Testing Tool

The testing tool is implemented on Windows in the Python programming language, and its minimal version to run the tool needs to be at least 3.10. We chose Python because it offers many modules and libraries that simplify controlling other applications, like the one we used - subprocess. We used this library to control hashcat and the nvidia-smi (which we used to get bus utilisation data). Other modules we used for collecting data are pynvml and psutil. The tool has two modes, one where it runs all the tests and the other one where it takes the data it collected from tests and prepares them for analysis.

6.1.1 Files

The tool is divided into four files: main.py, hashcat.py, output.py and analysis.py.

Main.py

Main.py is the main file through which the tool is launched. The tool parses the input using the argparse module in the main class. It also prepares all the other classes and creates all folders needed. If the tool is launched in the testing mode, the main class calls the dummy() function from the hashcat class to heat up the GPU. It does this so that the GPU is about the same temperature throughout all the tests, and the first tests do not have the advantage of a cold GPU. The benchmark() function is called from the hashcat class, which runs the default benchmark of the hashcat tool. After that, the main class parses input files and runs the hashcat with the run() function from the hashcat class. If the tool is run in the analysis mode, it loads the CSV data files. It creates their min, max, mean and median values using the countAverageMedianMaxMin() function of the analysis class.

Hashcat.py

The hashcat.py file contains the hashcat class as well as non-class functions. Hashcat class is the core of the testing. The hashcat is controlled with functions in this class. The tool

uses a subprocess module to run the hashcat application. This module can supply input to the application, read its output, set the timeout and more. To get other information about GPUs, the tool uses the pynvml module for NVIDIA GPUs and the psutil module for AMD GPUs. The Table 6.1 shows all functions the hashcat class contains. The rest of functions that the hashcat.py file contains are shown in the Figure 6.2.

dummy()	Used for heating up the GPU.
benchmark()	Launches the hashcat in benchmark mode and collects data from it.
getGpuName()	Returns the name of the GPU inside the Computer.
getGpuInfo()()	Returns the output of hashcat launched with the -I argument.
runHashcat()	Launches hashcat with input from the test and saves the output.

Table 6.1: Functions of hashcat class in the Hascat.py file

get_nvidia_gpu_memory_utilization()	Returns memory utilisation information for NVIDIA GPUs.
get_nvidia_gpu_bus_utilization()	Returns memory bus utilisation information for NVIDIA GPUs.
get_amd_gpu_memory_utilization()	Returns memory utilisation information for NVIDIA GPUs.
setAttributes()	Returns set of arguments that is inputted into subprocess that runs hashcat.
getRuntime()	Returns how long should the hashcat run for different type of attack mode.
getAttackMode()	Returns number of attack mode based on attack mode string.
fixHashName()	Returns hash with replaced signs that cannot be used in windows file names.

Table 6.2: Collected information from GPU.

Output.py

Output.py file contains the functions for saving output from the tests. These functions are called in the run function of the hashcat class after the test is done. The saveOutput() function parses the output of the hashcat tool using the re module. Then the output is saved into the file, and it is named the same as the name of the hash that was tested. The saveBM() function saves the other hardware information which is not collected from the output of the hashcat tool.

Analysis.py

Analysis.py does not have a specific class. The function in this file is called in the main class, and it prepares the data for data analysis. It loads up the collected CSV data from a file where the data from testing are collected, calculates median and mean values, and finds min and max values. Then it saves the output into two files, one based on the hash and the other based on the type of attack. These files are further explained in Chapter 5.3

6.1.2 Using the Tool

Launching the tool is done by using the Windows command line interface. The Figure 6.1 shows us all the arguments we can use when running the script. Table 6.3 shows us an explanation of each argument and if it is required.

```
main.py [-h] -i INPUT -hs HASHES -hf HASHFILES -hp HASHCATPATH -o OUTPUT
        [-a | --analysis | --no-analysis] [--amd | --no-amd]
```

Figure 6.1: Tool arguments.

Argument	Explanation	Required
–input (-i)	Input folder with configuration files.	True (when –analysis is not present)
–hashes (-hs)	Input file with hashes.	True (when –analysis is not present)
–hashfiles (-hf)	Input folder location of files with hashes.	True (when –analysis is not present)
–hashcatpath (-hp)	Input hashcat path.	True (when –analysis is not present)
–output (-o)	Set output folder path.	True
–analysis (-a)	Creates CSV file for the analysis.	False
–amd	Run with this parameter if you have AMD graphics card.	False

Table 6.3: Explanation of the tool arguments.

6.2 Data Visualisation Scripts

For data visualisation, we used Jupyter Notebook[1], a server-client application which allows us to edit files in our web browser and visualise the output. The language we used for this was again Python because it offers many modules for working with data and their visualisation, like pandas, numpy or matplotlib. These scripts are located in the jupyter-notebook folder. These files are:

- display-data-from-output-folders.ipynb - Displays the data from CSV Files inside the output\results folder or the output\results-together folder.

- display-data-from-output-folders-config-median-mean.ipynb - Displays the median and mean data grouped by configuration from CSV Files inside the output\results folder.

- correlation-of-all-files.ipynb - Calculates correlation between all files from the output\results folder and GPUs hardware information and displays it.

- correlation-of-results-togethe.ipynbr - Calculates correlation between all files from the output\results-together folder and GPUs hardware information and displays it.

---

[1]https://jupyter.org/

- correlation-only-median-values.ipynb - Calculates correlation between median data from all files from the output\results folder and GPUs hardware information, calculate median values of these correlations and displays it and does the same thing for each configuration separately.

- correlations-of-single-files-for-configs.ipynb - Calculates correlations grouped by configuration for a single file from the output\results folder and GPUs hardware information and displays it.

- graph-memory-over-time.ipynb - Display the used memory over time for a file from GPUS\GPU_Name\memory_data folder.

- individual-file-correlation.ipynb - Calculate correlation between data from all files from the output\results folder and GPUs hardware information, calculate median values of these correlations and display it and does the same thing for just median data separately.

- find-memory-hard-hashes.ipynb - Finds the hashes that use the most memory.

- median-correlations-of-single-files-for-configs - Calculate correlation grouped by configuration between median data from all files from the output\results folder and GPUs hardware information, calculate median values of these correlations and display it and does the same thing for just median data separately.

- median-from-correlations.ipynb - Calculated correlation for a single file. Also calculates correlation between data from all files from the output\results folder and GPUs hardware information, calculate median values of these correlations and displays it and does the same thing for each configuration separately.

Chapter 7

Experiments Implementation

We need to have many tests to analyse the data and decide what hardware specs of GPUs are the most important. Because of this, there are 478 hashes tested with different configurations. These hashes come from the hashcat tool's example hashes, and each hash is unique. To further understand with what configurations which GPU specs work the best, we need different configurations. The tests consist of 7 different configurations.

7.1 Configurations

The tests are divided into four different attack strategies: combinator, dictionary, force and hybrid attack. Furthermore, there are multiple dictionary tests with the use of rules.

7.1.1 Combinator attack

The combinator attack is a dictionary attack that combines two or more wordlists. These wordlists can be, for example, leaked password databases, custom wordlists or actual language dictionaries. Figure 7.1 shows us the settings the tool uses for combinator attack. It uses an English dictionary and its copy. This dictionary was chosen because the general public tend to choose weak passwords which are easy to remember [3] which contain primarily common words in their passwords.

```
1    attack-mode: "combinator"
2    dictionaries: "..\\bachelors-thesis-main\\dictionaries\\*"
```

Figure 7.1: Combinator attack settings.

7.1.2 Dictionary attacks

A dictionary attack uses a wordlist and goes through each word and checks if its hash is the same as the password hash. Because we already use wordlists without any modifications in the combinator attack, we decided to use rules to edit our wordlists. The first wordlist has a hundred passwords, and the second has ten passwords. Similarly, the first ruleset has ten rules, and the second ruleset has a hundred rules. With this, we can compare if there is any difference in the use of bigger or smaller wordlists with different numbers of rules. We also use the Rockyou wordlist, on which we will apply both of the rulesets. We decided to

41

use the Rockyou wordlist because it is the largest wordlist of leaked passwords. The figures 7.2, 7.3 7.4 and 7.5 show us the settings for each dictionary attack.

```
1   attack-mode: "dictionary"
2   dictionaries: "..\\bachelors-thesis-main\\dictionaries-dict\\test-dic1.dic"
3   ruleset: "..\\bachelors-thesis-main\\rules\\ruleset1.rule"
```

Figure 7.2: First dictionary attack settings.

```
1   attack-mode: "dictionary"
2   dictionaries: "..\\bachelors-thesis-main\\dictionaries-dict\\test-dic2.dic"
3   ruleset: "..\\bachelors-thesis-main\\rules\\ruleset2.rule"
```

Figure 7.3: Second dictionary attack settings.

```
1   attack-mode: "dictionary"
2   dictionaries: "..\\bachelors-thesis-main\\dictionaries-dict\\rockyou.txt"
3   ruleset: "..\\bachelors-thesis-main\\rules\\ruleset1.rule"
```

Figure 7.4: Third dictionary attack settings.

```
1   attack-mode: "dictionary"
2   dictionaries: "..\\bachelors-thesis-main\\dictionaries-dict\\rockyou.txt"
3   ruleset: "..\\bachelors-thesis-main\\rules\\ruleset2.rule"cd
```

Figure 7.5: Fourth dictionary attack settings.

7.1.3 Brute-force (mask) attack

A force attack, also known as a mask attack, is an attack where the attacker tries every possible combination of characters in a charset. We chose this type of attack because it finds the correct password if given enough time and the right keyspace. In Figure 7.6, we can see the settings of the attack. The increment is set to true, meaning that the mask will go from single-character passwords and progress to two-character passwords and three-character passwords and so on, until it finds the correct password or runs out of keyspace. The mask we are using for this attack is the hashcat tool's default mask taken from hashcats wiki[1], which is: -1 ?l?d?u -2 ?l?d -3 ?l?d*!$@_ ?1?2?2?2?2?2?2?3?3?3?3?d?d?d?d. We use this mask because most of the hashcat's inexperienced users will probably use the default mask and because it is pretty complex and has a large keyspace.

---

[1]https://hashcat.net/wiki/doku.php?id=oclhashcat#default_values

```
1    attack-mode: "brute-force"
2    increment: true
3    hashcatPath: /home/richard/hashcat-6.2.6/hashcat.bin
```

Figure 7.6: Brute-force (mask) attack settings.

7.1.4   Hybrid attack

A hybrid attack is a combination of a dictionary and brute-force attack. Combinations like these are mainly used when looking for a password because they combine the words a password might contain and some extra characters. People often put one special character like * or % at the end of the password, and this type of attack can easily counter that. The Figure 7.7 shows us the settings of this attack. We used the English dictionary as people usually put casual words in their passwords, and then we used the Hashcats default mask with increment since we want to have a big keyspace but want to try the short variations first.

```
1    attack-mode: "hybrid-a"
2    increment: true
3    charset: "?1?2?2?2?2?2?2?2?3?3?3?3?3?d?d?d?d"
4    custom-charsets:
5      charset1: "?l?d?u"
6      charset2: "?l?d"
7      charset3: "?l?d*!$@_"
8    dictionaries: "..\\bachelors-thesis-main\\dictionaries\\English.dic"
```

Figure 7.7: Hybrid attack settings.

7.2   Used Graphics Cards

The graphics cards that these experiments were tested on are NVIDIA GTX 1080 Ti, NVIDIA RTX 2080 Super, NVIDIA RTX 3070, NVIDIA RTX 3090, NVIDIA RTX A4000 and AMD Radeon RX Vega 64. The further comparison of hardware specifications of these GPUs is shown in the Tables 7.1 and 7.2.

	Geforce GTX 1080Ti	Geforce RTX 2080 Super	Geforce RTX 3070	Geforce RTX 3090
Architecture	Pascal	Turing	Ampere	Ampere
Processors	3584	3072	5888	10496
Base Clock (GHz)	1.48	1.65	1.50	1.40
Boost Clock (GHz)	1.58	1.82	1.73	1.70
Standard Memory Config	11GB GDDR5X	8GB GDDR6	8GB GDDR6/	24GB GDDR6X
Memory Interface Width	352-bit	256-bit	256-bit	384-bit

Table 7.1: Comparison of tested GPUs

	Geforce RTX A40000	Radeon RX Vega 64
Architecture	Ampere	Vega
Processors	6144	4096
Base Clock (GHz)	0.74	1.25
Boost Clock (GHz)	1.56	1.55
Standard Memory Config	16GB GDDR6	8GB HBM2
Memory Interface Width	256-bit	2048-bit

Table 7.2: Comparison of NVIDIA RTX 30 Series GPUs [6]

Chapter 8

Analysis of Experimental Results

This Chapter is the core of this thesis, where we look at the collected data from the tests and then analyse them. We will use the data files that already have the modified data prepared for analysis. In these files, there are min, max, mean and median values calculated from the collected test data and then arranged so that it is easy to go through them. To analyse these data, we use Jupyter Notebook. Jupyter Notebook uses Python scripts to better visualise collected data, generate correlations from collected data, display the correlation matrix data and the correlation matrix as a graph and so forth. The analysis focuses, in particular, on how the specific hardware parameters of GPUs affect password cracking performance.

8.1 Correlations

Correlation shows us what is the relation between each column of data. For analysis, where we are looking for the most crucial hardware parameters of graphic cards for cracking passwords and looking at relationships between data and hardware parameters, the use of correlation is ideal. After correlation is calculated from the inputted data, we get a number between -1 and 1 on the output. If the output is -1, that means a perfect negative correlation, meaning that as one variable rises, the other decreases. If the output is 1, that means a perfect positive correlation, meaning that as one variable rises, the other rises. If the correlation output is 0, the variables have no relationship. An example would be if we have a correlation value between speed and the number of processors being 1, then when we increase the processor count, the speed also increases.

8.1.1 Correlations of Individual Files

First, we calculated a correlation separately for each file. The Python script for this is saved inside the individual-file-correlation.ipynb file. Then we show the correlation matrix on the output. Here we can see a correlation of the data for each hash. Figure 8.1 shows an example of such a correlation matrix. We could use this data for the analysis, but if we only used it and analysed every hash mode separately, it would take months to complete. That is why we need to create correlations from more data. In sections 8.1.2 and 8.1.4 we look at the correlations from all data. Creating correlations only for each hash type would also not let us see which hardware parameters are more important in which attack mode, so we must create data correlations for each attack mode separately. We look at this procedure in sections 8.1.3 and 8.1.5.
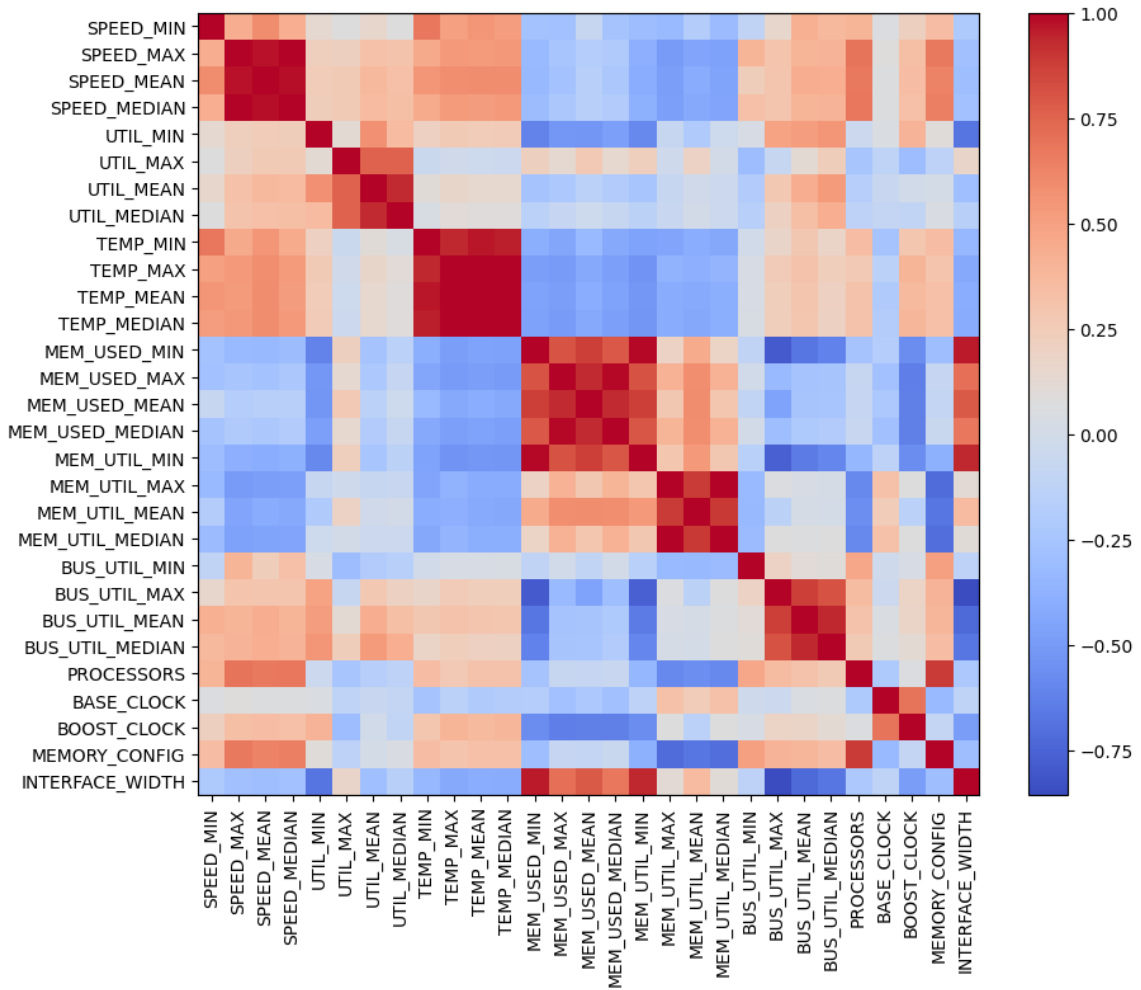
Figure 8.1: Correlation matrix of all data for single test.

8.1.2 Correlation of All Data

We calculated the correlation between all the collected data. This calculation is done in the correlation-of-all-files jupyter notebook file, and to calculate them, we used the Pandas Python module. As shown in Figure 8.2, the correlation of these data would suggest that the speed at which passwords are cracked should be higher if most of the other collected data values are and the hardware specifications values are lower. Pure logic already indicated that this correlation must be wrong, and it is. We can see the opposite when looking at example of randomly picked correlation of file, where the correlation was calculated separately. Figure 8.1 shows us correlation of data from this file. There can be multiple factors in the play why the correlation of all data is misleading. First, we will create a separate correlation of data from every attack mode.
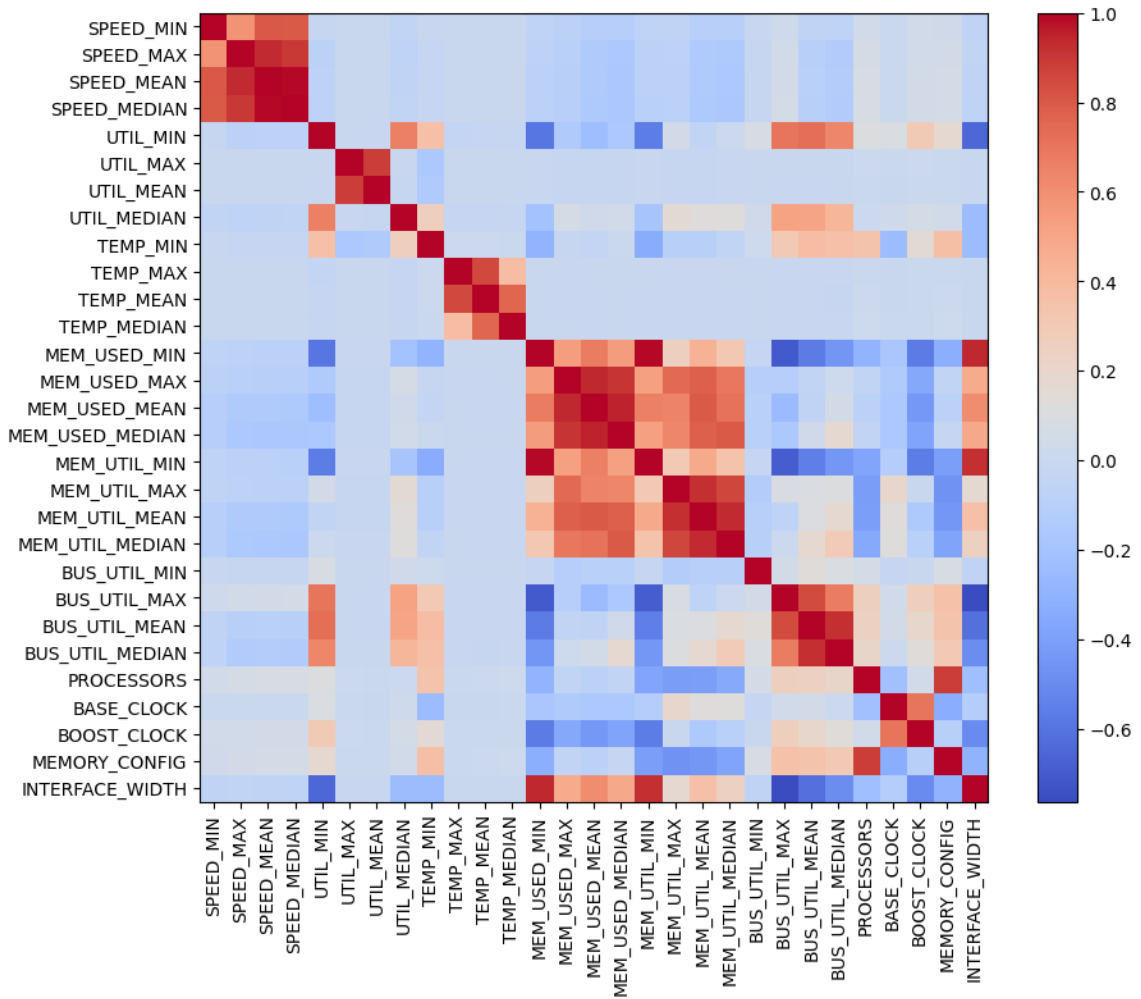
Figure 8.2: Correlation matrix of all data for single test.

8.1.3 Correlation of Data Grouped by Attack Mode

File correlation-of-attack-modes shows separate correlations created for each file inside the results-together folder. These files each have data divided by the attack mode, and they are also named after that attack mode. We can see the correlations of these data in the Figures 8.3, 8.4, 8.5, 8.6, 8.7, 8.8 and 8.9. Like the correlation of all the data, these correlations also seem misleading. These correlations of data that are grouped together are misleading because the values of hash mode vary. In a hash mode like SHA, we can easily get up to the speed of 533552554 hashes/second, but when using a hash mode like VeraCrypt, we can get as low as 30 hashes/second. These speeds vary this much while the utilisation or the memory used values stay the same. That is why we must come up with another solution. We must calculate the correlation of each file separately, and then we can calculate median values from those correlations to get the proper values. This procedure is described in sections 8.1.4 and 8.1.5.
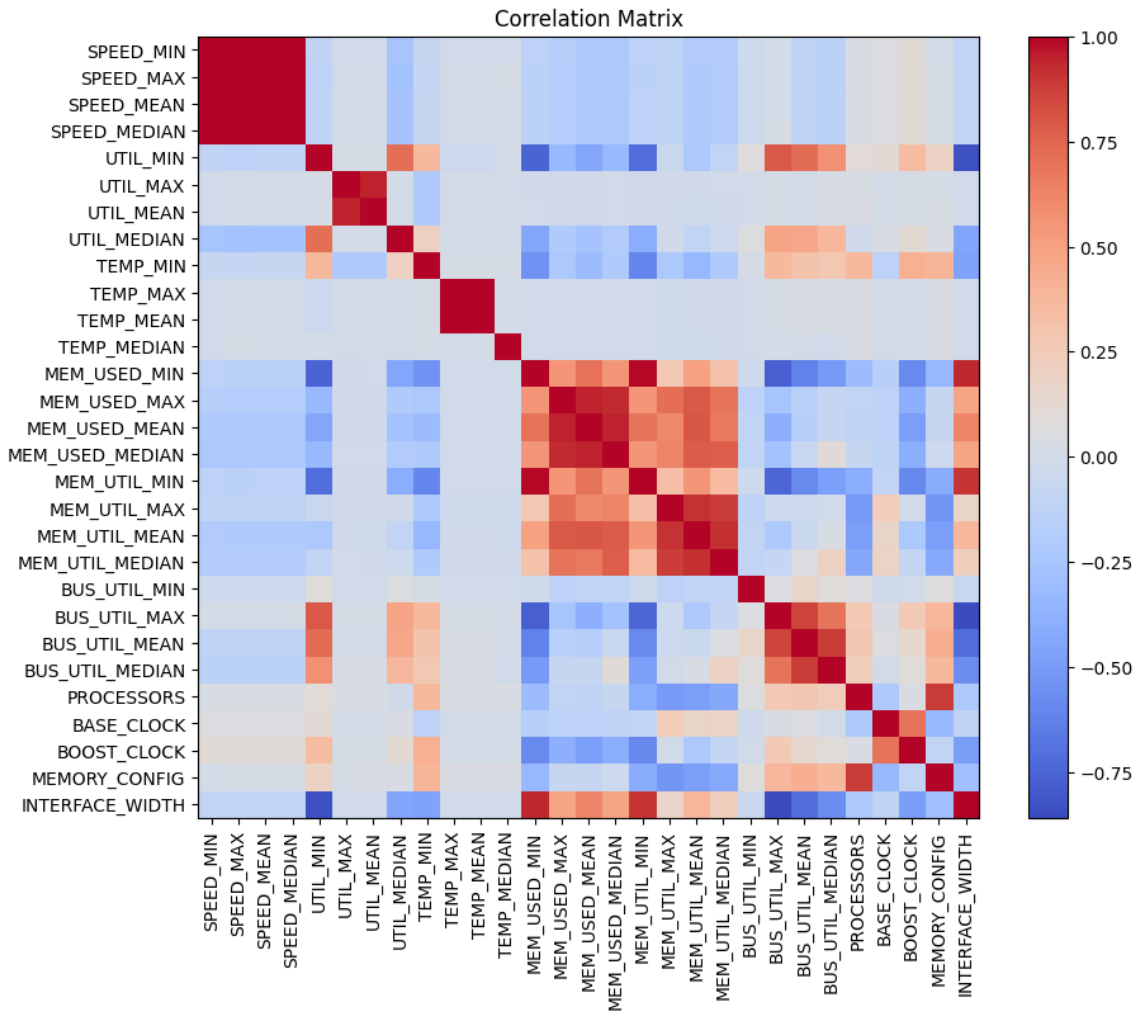
Figure 8.3: Correlation matrix of all data grouped by combinator configuration.

Figure 8.4: Correlation matrix of all data grouped by dict1 configuration.
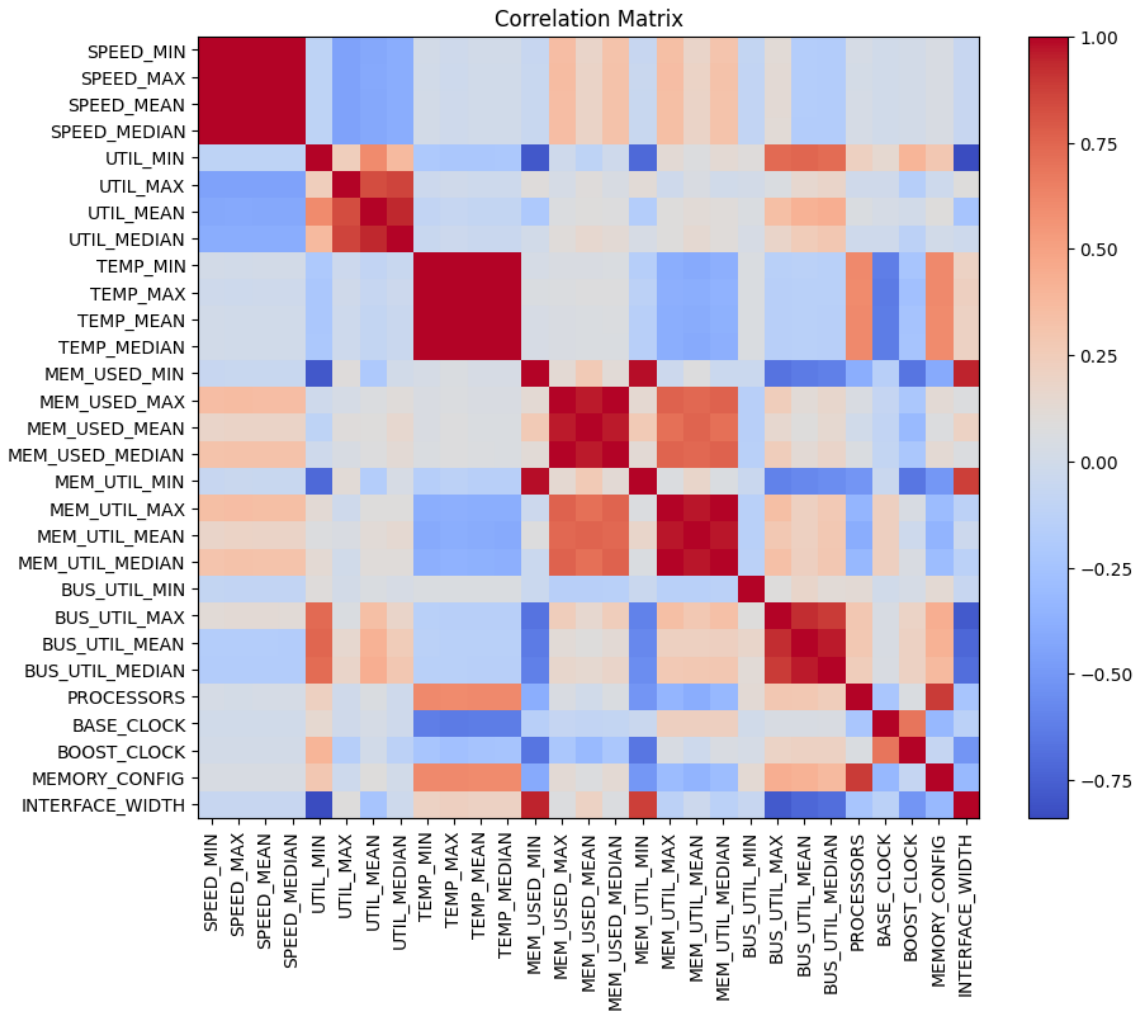
Figure 8.5: Correlation matrix of all data grouped by dict2 configuration.
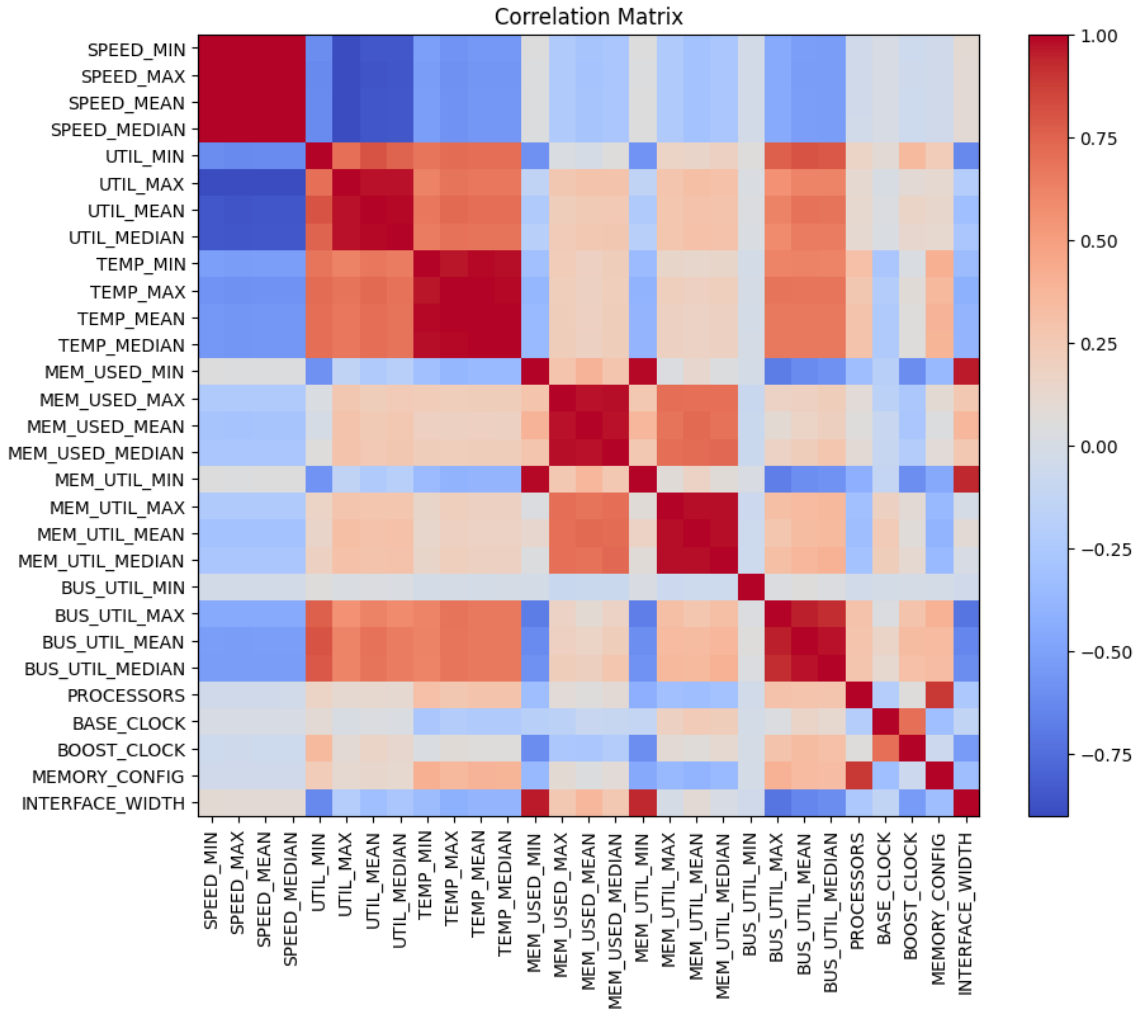
Figure 8.6: Correlation matrix of all data grouped by dict3 configuration.
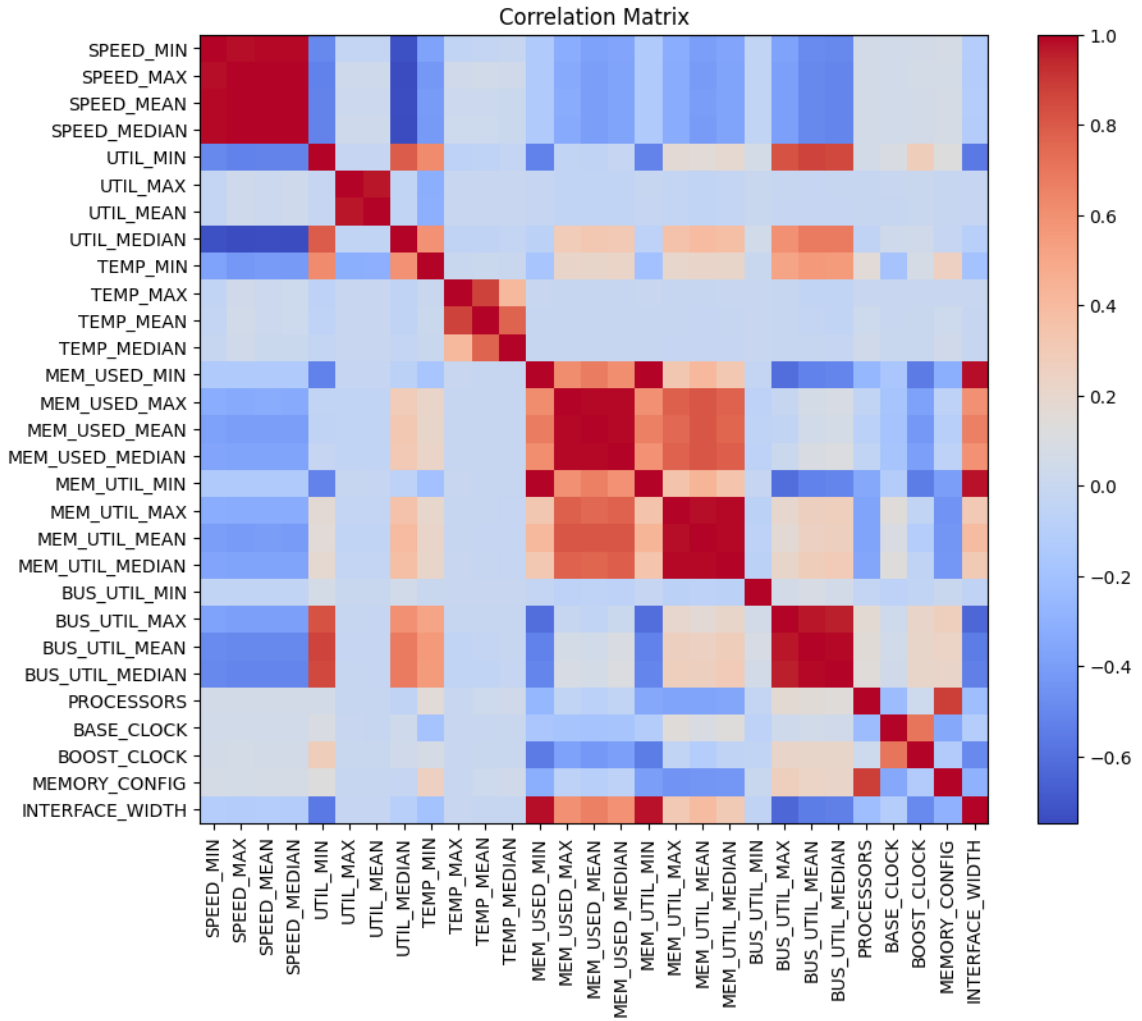
Figure 8.7: Correlation matrix of all data grouped by dict4 configuration.
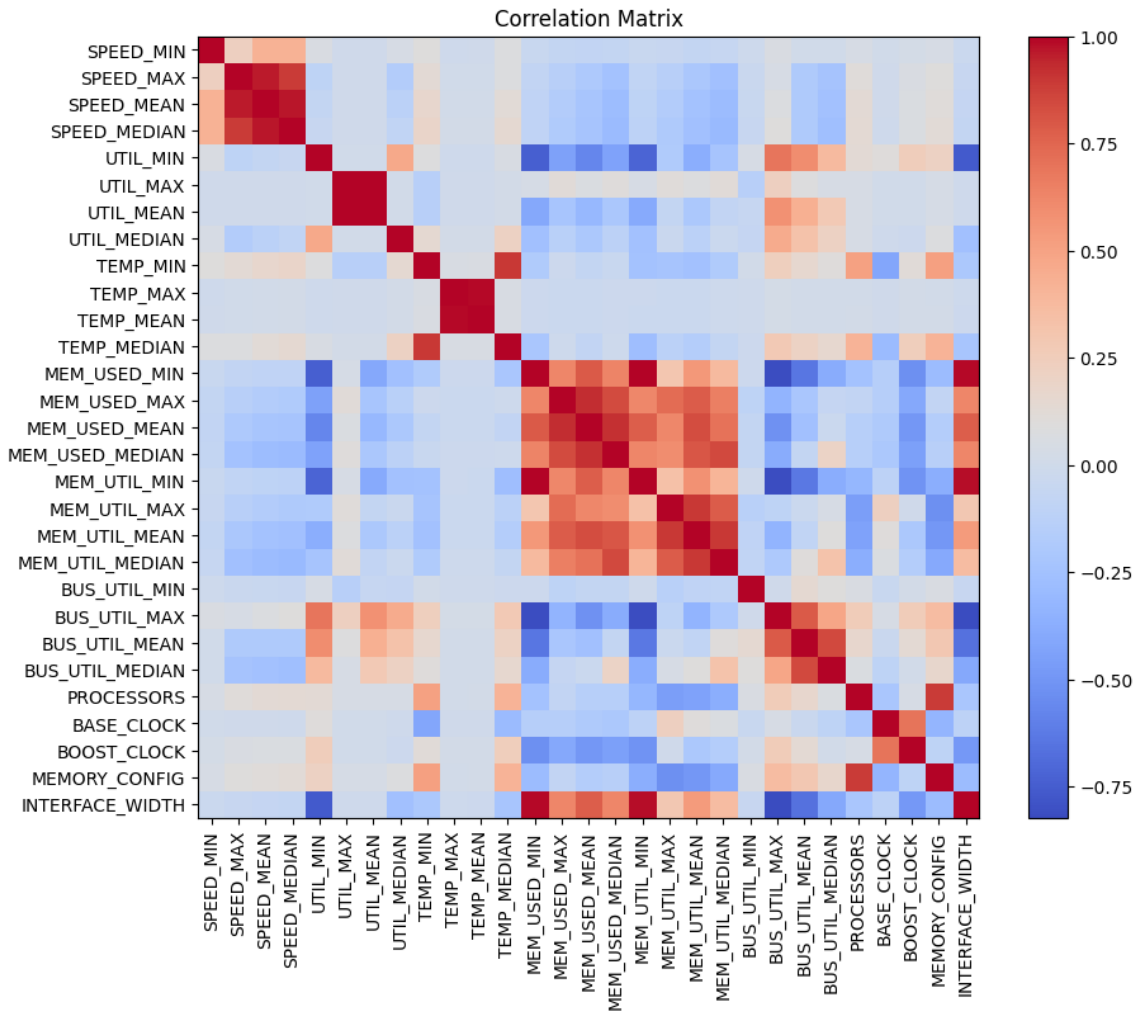
Figure 8.8: Correlation matrix of all grouped by force configuration.
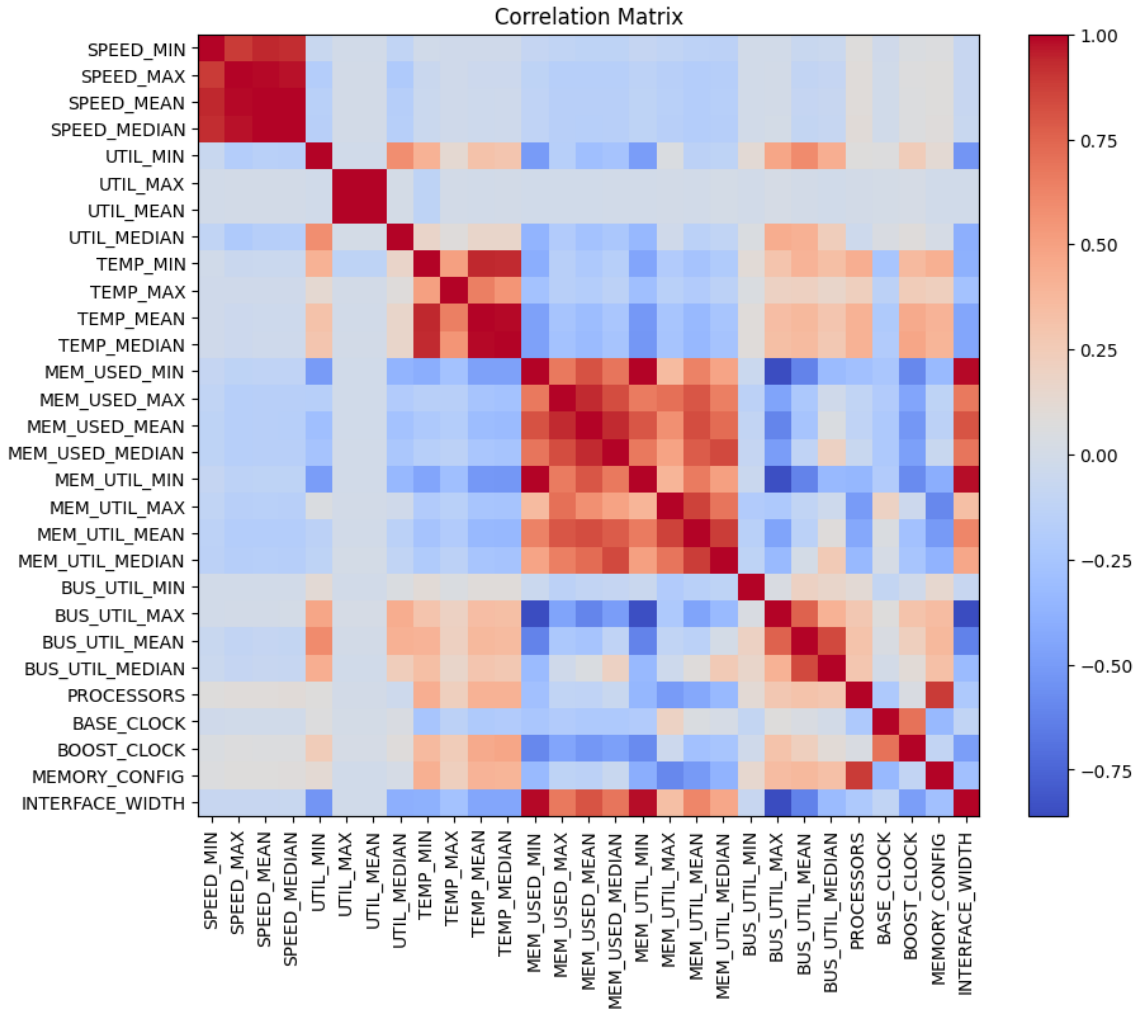
Figure 8.9: Correlation matrix of all data grouped by hybrid configuration.

8.1.4 Median Calculated From Correlations of Individual Files

To calculate the correlation values that are not misleading, we must count them for each file separately and then find the median values of these calculations. The code for these calculations is saved in the median-from-correlations.ipynb file. When we look at the correlation matrix, we can see that the data is finally starting to make sense and that we can properly analyse them. We can see this because if we compare the matrix shown in Figure 8.10 to the correlation matrix calculated from single hash file shown in Figure 8.1, we can see the similarities.
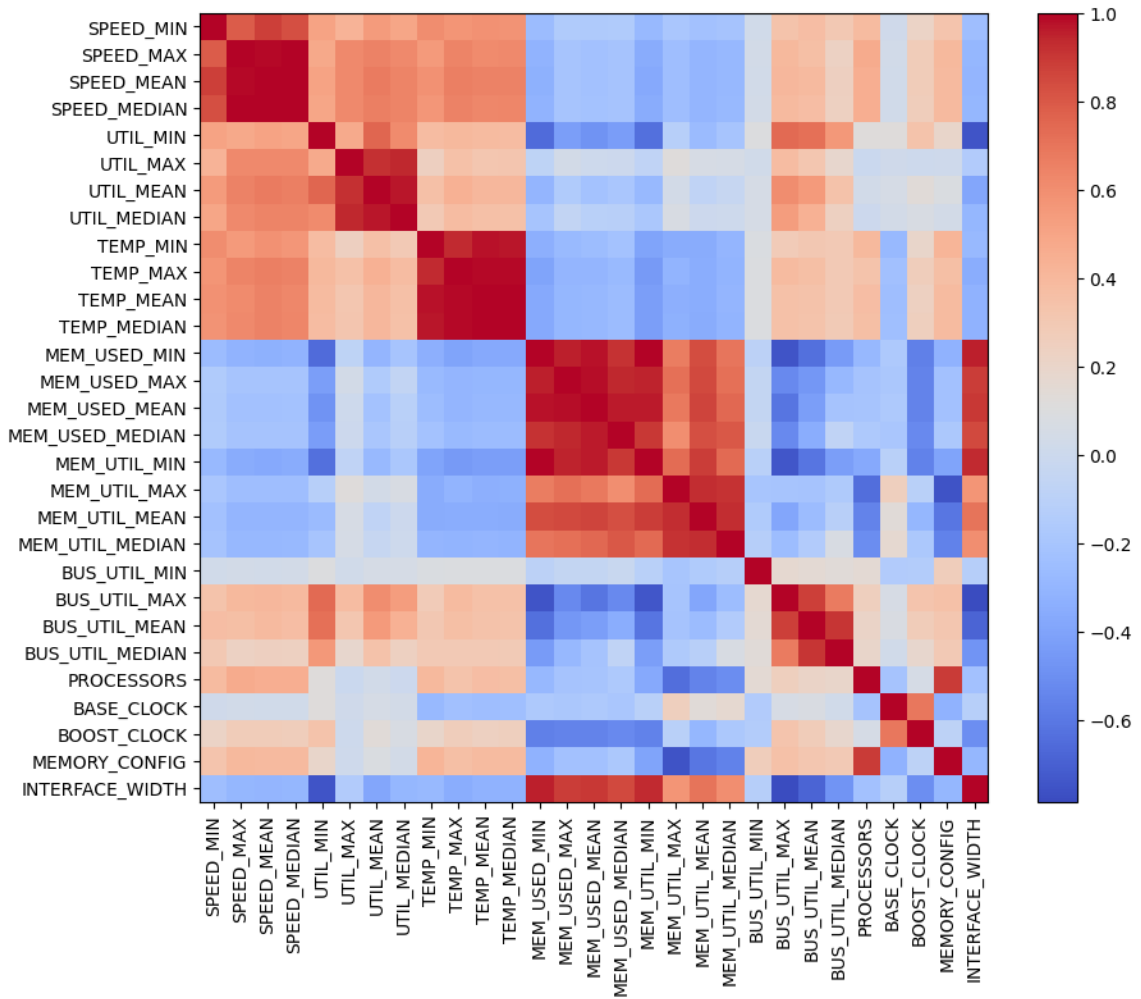
Figure 8.10: Median from correlation matrix of all data.

The most important values for the analysis are the median values because they show how the cases will look most of the time. For example, there can be a minimum utilisation value of 20% when its median is 80%, and that 20% was just some error, which is why we will use mostly median values for analysis. Looking at the correlation matrix shown in Figure 8.11, we can see that the more processors the GPU has, the higher its speed is. We can also see that the speed is higher when memory_config, boost_clock, utilisation, temperature and the bus utilisation values are higher. The processors are the most critical GPU hardware specification when cracking passwords overall because the number of processors has the highest correlation with speed.

The second most critical GPU hardware specification overall is memory. As we can see in the matrix, memory size has the second-highest correlation value with speed. Interestingly, when we look at the used memory and the memory utilisation correlation values with speed, they are both about -0.20. However, when we look at the memory and GPU utilisation, we can see that they are positively correlated. This phenomenon happens because faster GPUs try more candidate passwords in a shorter time, which means they can remove them from their memory faster and do not clog it up. Table 8.1 shows an example of this. As we can see in the correlation table, this would be mostly the case. However, there are cases

and attack modes where more memory is an advantage. For example, we can see this in the Table 8.2 . We look at this phenomenon more in Section 8.1.6. Even if more memory does not always give us the upper hand, it certainly is no downfall and overall, the more memory, the better.

GPU	SPEED MEDIAN	MEM USED MEDIAN	MEM UTIL MEDIAN
NVIDIA GeForce GTX 1080 Ti	1407.50	3989.37	35.42
NVIDIA GeForce RTX 2080 SUPER	2587.00	328.73	4.01
NVIDIA GeForce RTX 3070	2751.00	2278.07	27.81
NVIDIA GeForce RTX 3090	4617.00	499.66	2.03
NVIDIA RTX A4000	2427.00	2673.39	16.33
Radeon RX Vega	1558.00	5915.95	36.30

Table 8.1: Data from combinator VeraCrypt PBKDF2-HMAC-RIPEMD160 + boot-mode + AES (legacy) test.

GPU	SPEED MEDIAN	MEM USED MEDIAN	MEM UTIL MEDIAN
NVIDIA GeForce GTX 1080 Ti	49168.50	10842.38	96.26
NVIDIA GeForce RTX 2080 SUPER	80987.00	7115.68	86.86
NVIDIA GeForce RTX 3070	93567.00	6976.07	85.16
NVIDIA GeForce RTX 3090	189263.00	23245.83	94.59
NVIDIA RTX A4000	89304.00	15716.01	95.97
Radeon RX Vega	59100.00	6525.10	40.00

Table 8.2: Data from combinator SNMPv3HMAC-SHA224-1288 test.

The last hardware aspect positively correlated with the speed is the boost_clock. This correlation implies that GPUs were running primarily at the boost clock speeds, which means that the boost clock speed is more important than the base clock speed. The boost clock speed is more important than the base clock speed because cracking the passwords puts a heavy load on the GPUs, which means they go into their boost speeds most of the time while cracking the passwords.

The last hardware parameter in this correlation matrix is the interface width. The correlation between the interface width and the speed would suggest that the larger interface means less speed, but it could not be further from the truth. This value is wrong because the Radeon RX Vega 64 does not reach the highest speeds, but it has the largest interface, with its width being five times larger than the interface of the most powerful GPU used in this analysis, the RTX 3090. When we remove the Radeon RX Vega from the data comparison, we can see that the correlation between interface width and speed is positive. However, because Radeon RX Vega has such a large interface, it can easily use the system RAM alongside its built-in memory, and that is why a large interface is essential when dealing with attacks that use large amounts of memory. Overall, the interface width is not the most crucial aspect of the GPU for password cracking, but larger is better.

Figure 8.11: Median from correlation matrix of all median data.

8.1.5 Median Calculated from Correlations of Individual Files Grouped by Attack Mode

Each of the attack types uses the GPU differently. They may depend more on different hardware aspects, so we examine each attack type separately.

Combinator Attack

Combinator attack uses the combination of 2 dictionaries. Figure 8.13 shows us the correlation of collected data with the hardware characteristics of GPUs. As shown in this

correlation matrix, the hardware columns that positively correlate with speed are processors, base_clock, boost_clock and memory_config. When analysing the data, we will mainly use the median data for the same reasons as in Section 8.1.4. The correlation matrix made only from median data and GPU hardware specifications is shown in Figure 8.13.
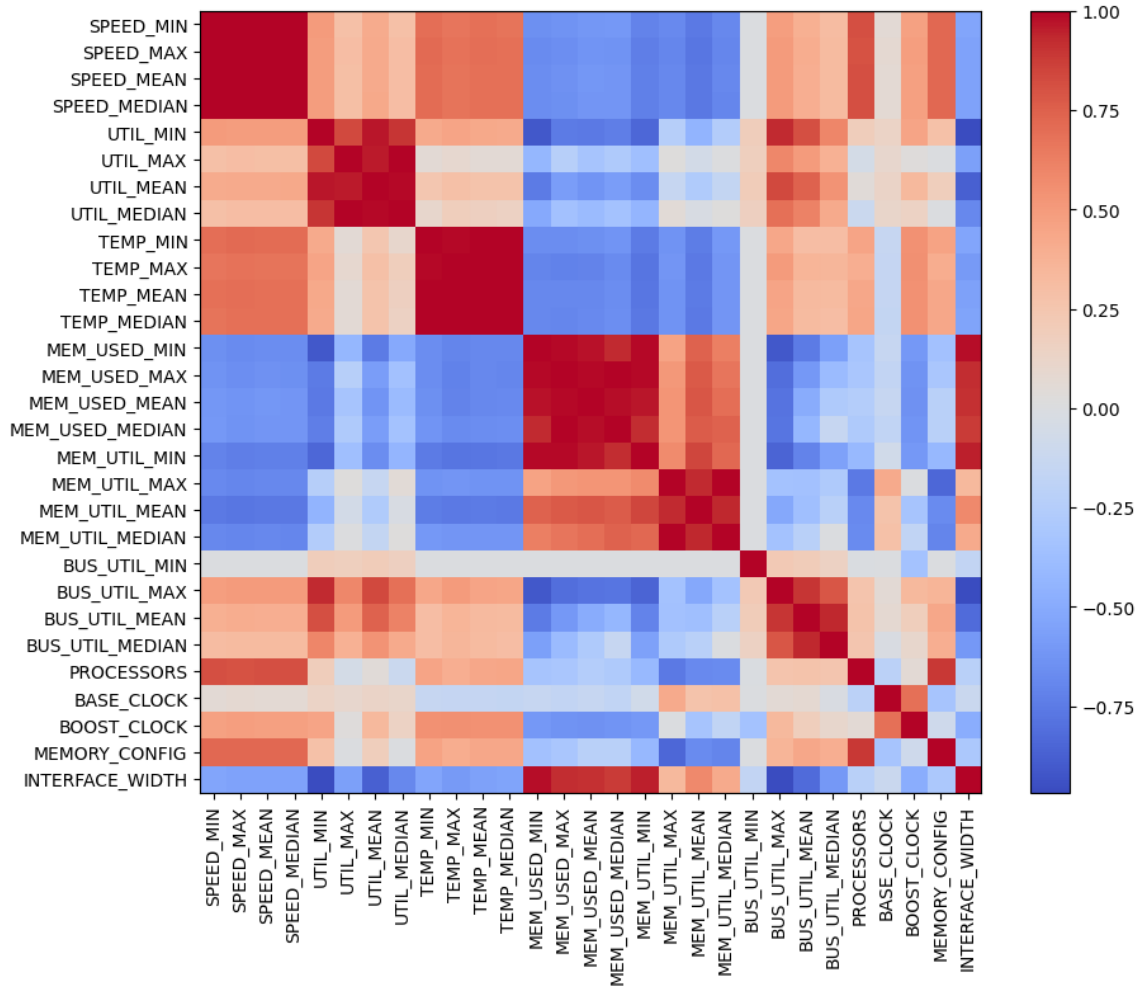
Figure 8.12: Median from correlation matrix of combinator configuration data.

	SPEED_MEDIAN	MEM_USED_MEDIAN	MEM_UTIL_MEDIAN	PROCESSORS	BASE_CLOCK	BOOST_CLOCK	MEMORY_CONFIG	INTERFACE_WIDTH
SPEED_MEDIAN	1.000000	-0.630916	-0.707168	0.808753	0.065268	0.470681	0.720805	-0.545684
MEM_USED_MEDIAN	-0.630916	1.000000	0.746451	-0.276823	-0.165027	-0.623740	-0.242180	0.883902
MEM_UTIL_MEDIAN	-0.707168	0.746451	1.000000	-0.688385	0.234496	-0.209036	-0.718013	0.394164
PROCESSORS	0.808753	-0.276823	-0.688385	1.000000	-0.210624	0.055271	0.890509	-0.221247
BASE_CLOCK	0.065268	-0.165027	0.234496	-0.210624	1.000000	0.691414	-0.322779	-0.116461
BOOST_CLOCK	0.470681	-0.623740	-0.209036	0.055271	0.691414	1.000000	-0.087469	-0.490223
MEMORY_CONFIG	0.720805	-0.242180	-0.718013	0.890509	-0.322779	-0.087469	1.000000	-0.292292
INTERFACE_WIDTH	-0.545684	0.883902	0.394164	-0.221247	-0.116461	-0.490223	-0.292292	1.000000

Figure 8.13: Median from correlation matrix of combinator configuration median data.

The processor count is the most important hardware characteristic for cracking passwords when combining two dictionaries. Its correlation value with speed is 0.8, which is close to being a linear growth. Overall, the higher the processor count is, the higher the password-cracking performance becomes.

The following hardware characteristic we are going to take a look at is memory. The correlation coefficient, which is 0.72, would suggest that the higher the memory of the GPU is, the higher the password-cracking performance becomes. However, the correlation coefficients of used memory with speed and memory utilisation with speed suggest the exact opposite. As explained in Section 8.1.4, this happens because the graphic cards with higher memory and higher password-cracking performance crack passwords faster and

thus do not need to use their whole memory. In Figure 8.14, we can see that the used memory stays around the same value, which means that even with time, the memory would not get fully used, because total memory of RTX 3090 is 24GB. However, there are cases where memory is essential. These cases are hashes like CISCO-IOS9(scrypt) or SolarWindsOrion, where the memory is used much more, and if it were smaller, the cracking performance would be worse as well. We can see the correlation matrices of these hashes in Figure 8.15 and Figure 8.16, where we can see that the memory and speed correlation coefficients are positive. Overall for the combinator attack, memory is not that important factor. However, there exist hashes that can use the surplus memory, and for these hashes, the cracking performance gets impacted by the memory size.
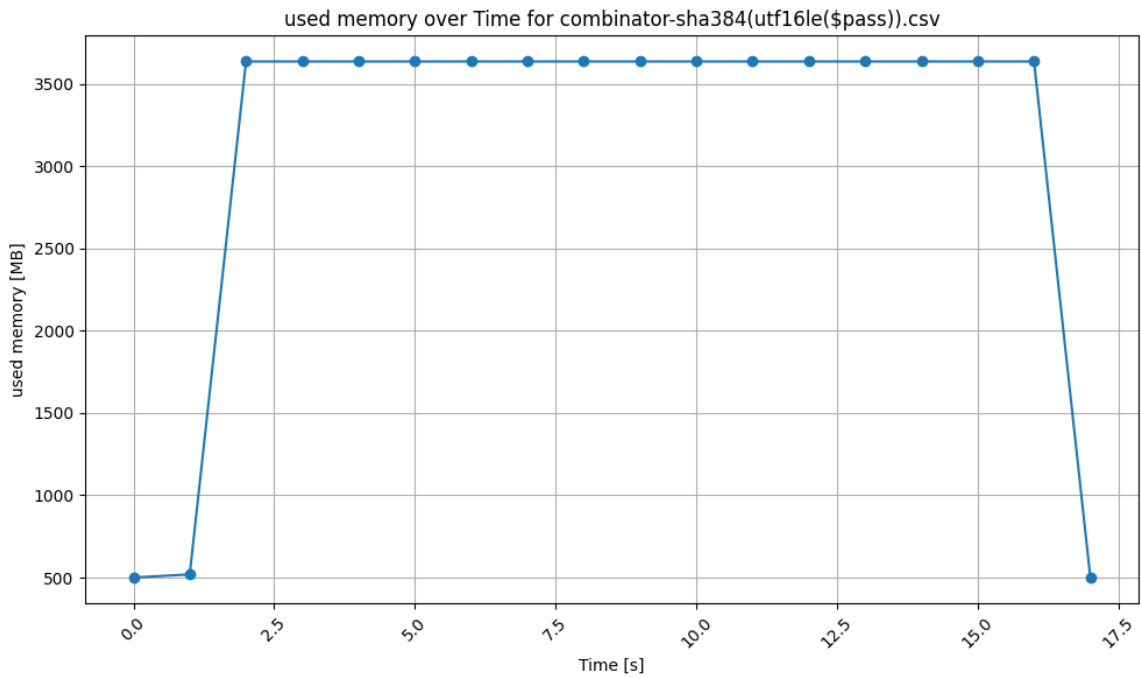
Figure 8.14: Used memory over time for combinator attack with sha384(utf16le($pass)) hash on RTX3090

	SPEED_MEDIAN	MEM_USED_MEDIAN	MEM_UTIL_MEDIAN	PROCESSORS	BASE_CLOCK	BOOST_CLOCK	MEMORY_CONFIG	INTERFACE_WIDTH
SPEED_MEDIAN	1.000000	0.793121	0.733085	0.877242	0.212201	0.428031	0.622544	-0.244829
MEM_USED_MEDIAN	0.793121	1.000000	0.586360	0.819200	0.058649	0.005359	0.688133	0.265676
MEM_UTIL_MEDIAN	0.733085	0.586360	1.000000	0.400630	0.495052	0.383130	-0.013532	0.179883
PROCESSORS	0.877242	0.819200	0.400630	1.000000	-0.210624	0.055271	0.890509	-0.221247
BASE_CLOCK	0.212201	0.058649	0.495052	-0.210624	1.000000	0.691414	-0.322779	-0.116461
BOOST_CLOCK	0.428031	0.005359	0.383130	0.055271	0.691414	1.000000	-0.087469	-0.490223
MEMORY_CONFIG	0.622544	0.688133	-0.013532	0.890509	-0.322779	-0.087469	1.000000	-0.292292
INTERFACE_WIDTH	-0.244829	0.265676	0.179883	-0.221247	-0.116461	-0.490223	-0.292292	1.000000
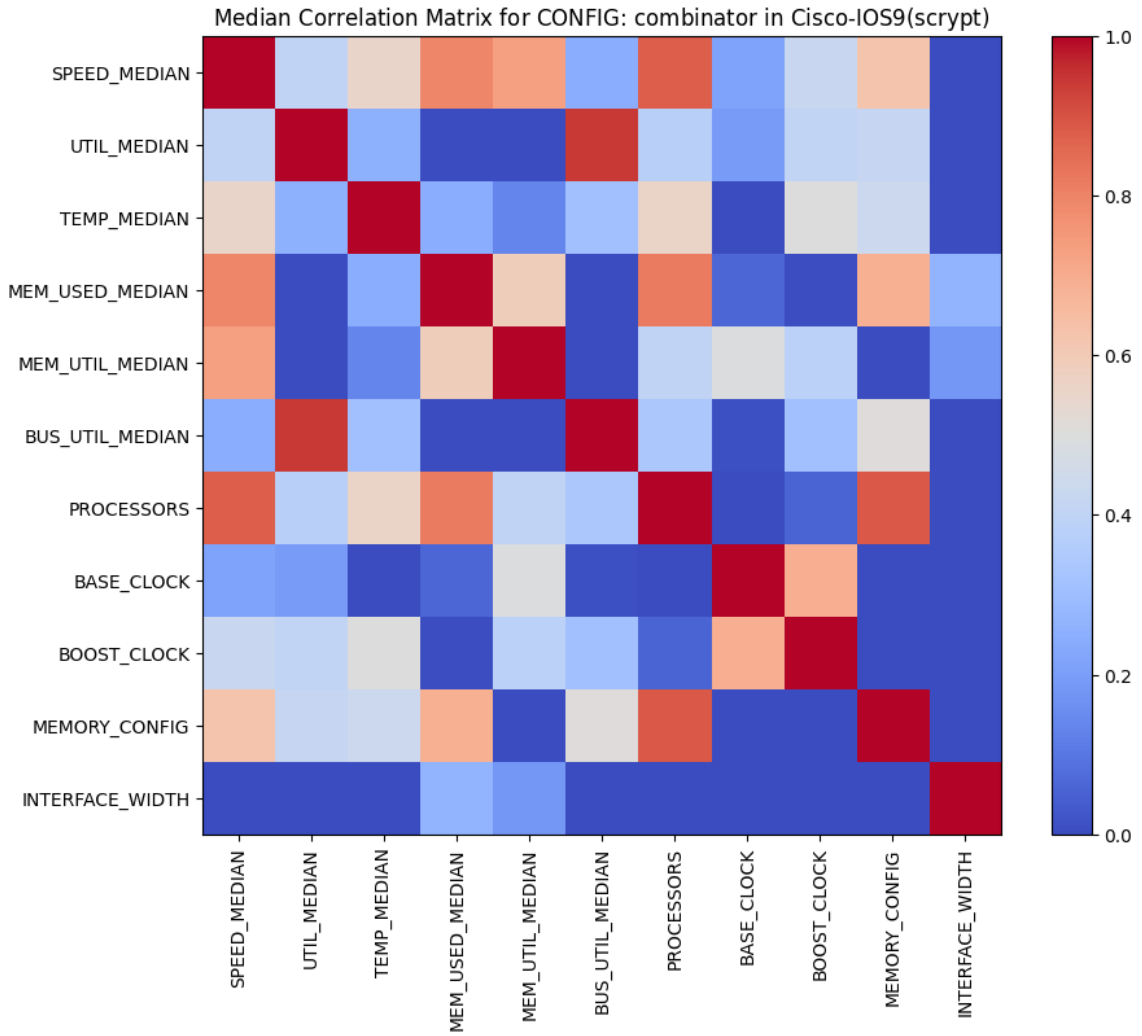
Figure 8.15: Median from correlation matrix of combinator configuration CISCO-IOS9(scrypt) hash.

	SPEED_MEDIAN	MEM_USED_MEDIAN	MEM_UTIL_MEDIAN	PROCESSORS	BASE_CLOCK	BOOST_CLOCK	MEMORY_CONFIG	INTERFACE_WIDTH
SPEED_MEDIAN	1.000000	0.322694	-0.238083	0.726577	-0.403088	-0.091989	0.406826	0.021820
MEM_USED_MEDIAN	0.322694	1.000000	0.208231	0.712181	-0.584524	-0.295729	0.912744	-0.394578
MEM_UTIL_MEDIAN	-0.238083	0.208231	1.000000	-0.200117	-0.040921	0.200332	-0.061516	-0.864929
PROCESSORS	0.726577	0.712181	-0.200117	1.000000	-0.210624	0.055271	0.890509	-0.221247
BASE_CLOCK	-0.403088	-0.584524	-0.040921	-0.210624	1.000000	0.691414	-0.322779	-0.116461
BOOST_CLOCK	-0.091989	-0.295729	0.200332	0.055271	0.691414	1.000000	-0.087469	-0.490223
MEMORY_CONFIG	0.406826	0.912744	-0.061516	0.890509	-0.322779	-0.087469	1.000000	-0.292292
INTERFACE_WIDTH	0.021820	-0.394578	-0.864929	-0.221247	-0.116461	-0.490223	-0.292292	1.000000
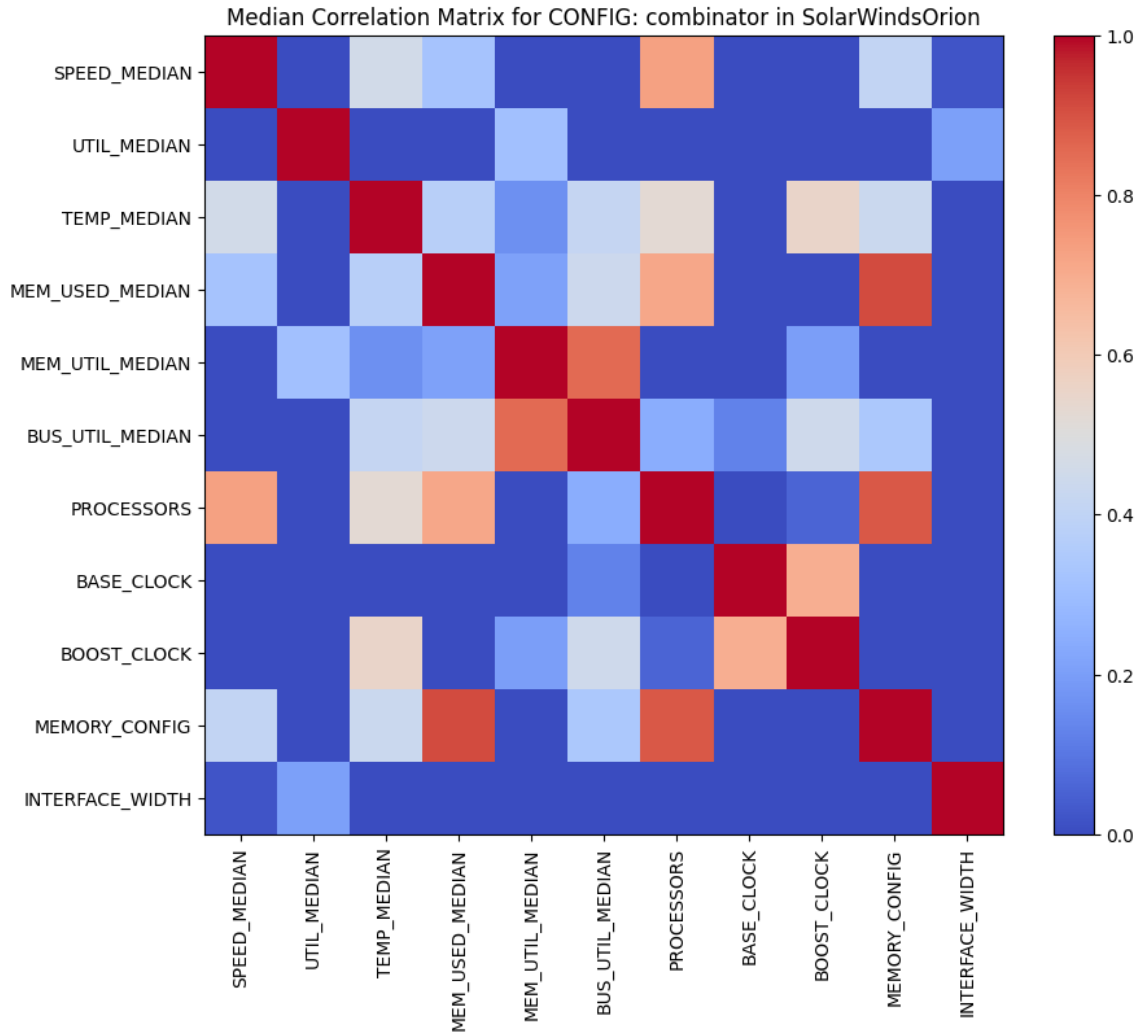
Figure 8.16: Median from correlation matrix of combinator configuration SolarWindsOrion hash.

The following hardware specifications that we will look at are the clock speeds. We have two clock speeds, the base clock speed and the boost clock speed. Clock speed refers to the speed at which the GPU processors operate. The higher, the better. The correlation matrix shows that the correlation coefficient of base clock speed with speed is almost 0. However, the boost clock speed is 0.47 because when the GPUs are under heavy load, they operate at the boost clock speed instead of the base clock speed. Because of this, the boost clock speed is much more important than the base clock speed.

The last hardware characteristic we will examine is the bus interface width. The correlation matrix would suggest that the smaller the bus interface is, the better. However, this is because the Radeon RX Vega 64 has a much larger bus interface than the rest of the GPUs, and it is not the most powerful GPU. When we delete the Radeon RX Vega 64 from counting the correlation matrix, we can see that the interface width positively correlates with speed. Overall having the interface width bigger is an advantage, but other hardware specifications are more crucial regarding the password-cracking performance.

Dict1 and Dict3 configurations

First, we will look at the attack strategies that each use ten rules. The dictionary used in the Dict1 strategy has only got a hundred words, while the dictionary used in Dict3 is Rockyou wordlist. Unfortunately, because the dict1 only creates 1000 possible hash candidates, some hashes were calculated so quickly that their data are unusable in analysis, which means that to analyse Dict1, we can only use more complex hashes.

In the Dict1 correlation matrix shown in Figure 8.17, we can see that the most crucial hardware specification is again the number of processors. We can see the same thing in the Dict3 correlation matrix shown in Figure 8.18.

The second most crucial hardware characteristic for dict1 is the boost clock speed, which is almost as crucial as the number of processors. The GPUs operate on the boost clock speeds most of the time when cracking passwords, so the base clock speed is not that important. We can see that for dict3, the boost speed is also essential, but it falls far from the number of processors. The keyspace difference between dict1 and dict3 can explain this phenomenon. While dict3 has enormous keyspace and can utilise the number of cores, dict1 cannot because it can fit into the number of processors, and more than half of processors on our GPUs are unused. Thus the speed of each processor becomes more critical in the dict1 configuration.

Memory is a special case, and like in other cases, it is not that important overall. Sections 8.1.4 and 8.1.5 explain why the memory size is not that important in most cases, and section 8.1.6 explains in which cases the memory size is essential.

	SPEED_MEDIAN	MEM_USED_MEDIAN	MEM_UTIL_MEDIAN	PROCESSORS	BASE_CLOCK	BOOST_CLOCK	MEMORY_CONFIG	INTERFACE_WIDTH
SPEED_MEDIAN	1.000000	-0.441628	-0.513873	0.451343	-0.038073	0.425273	0.360722	-0.571570
MEM_USED_MEDIAN	-0.441628	1.000000	0.837463	-0.244661	-0.135074	-0.593737	-0.032485	0.460146
MEM_UTIL_MEDIAN	-0.513873	0.837463	1.000000	-0.576430	0.079337	-0.310234	-0.322193	0.075034
PROCESSORS	0.451343	-0.244661	-0.576430	1.000000	-0.210624	0.027636	0.890509	-0.221247
BASE_CLOCK	-0.038073	-0.135074	0.079337	-0.210624	1.000000	0.691414	-0.322779	-0.116461
BOOST_CLOCK	0.425273	-0.593737	-0.310234	0.027636	0.691414	1.000000	-0.087469	-0.490223
MEMORY_CONFIG	0.360722	-0.032485	-0.322193	0.890509	-0.322779	-0.087469	1.000000	-0.292292
INTERFACE_WIDTH	-0.571570	0.460146	0.075034	-0.221247	-0.116461	-0.490223	-0.292292	1.000000

Figure 8.17: Median from correlation matrix of dict1 configuration median data.

	SPEED_MEDIAN	MEM_USED_MEDIAN	MEM_UTIL_MEDIAN	PROCESSORS	BASE_CLOCK	BOOST_CLOCK	MEMORY_CONFIG	INTERFACE_WIDTH
SPEED_MEDIAN	1.000000	-0.240014	-0.400077	0.810588	0.046161	0.404787	0.741466	-0.434488
MEM_USED_MEDIAN	-0.240014	1.000000	0.264924	0.000000	-0.130090	-0.344881	0.000000	0.545433
MEM_UTIL_MEDIAN	-0.400077	0.264924	1.000000	-0.535304	0.194088	0.000000	-0.678036	0.000000
PROCESSORS	0.810588	0.000000	-0.535304	1.000000	-0.210624	0.055271	0.890509	-0.221247
BASE_CLOCK	0.046161	-0.130090	0.194088	-0.210624	1.000000	0.691414	-0.322779	-0.116461
BOOST_CLOCK	0.404787	-0.344881	0.000000	0.055271	0.691414	1.000000	-0.087469	-0.490223
MEMORY_CONFIG	0.741466	0.000000	-0.678036	0.890509	-0.322779	-0.087469	1.000000	-0.292292
INTERFACE_WIDTH	-0.434488	0.545433	0.000000	-0.221247	-0.116461	-0.490223	-0.292292	1.000000
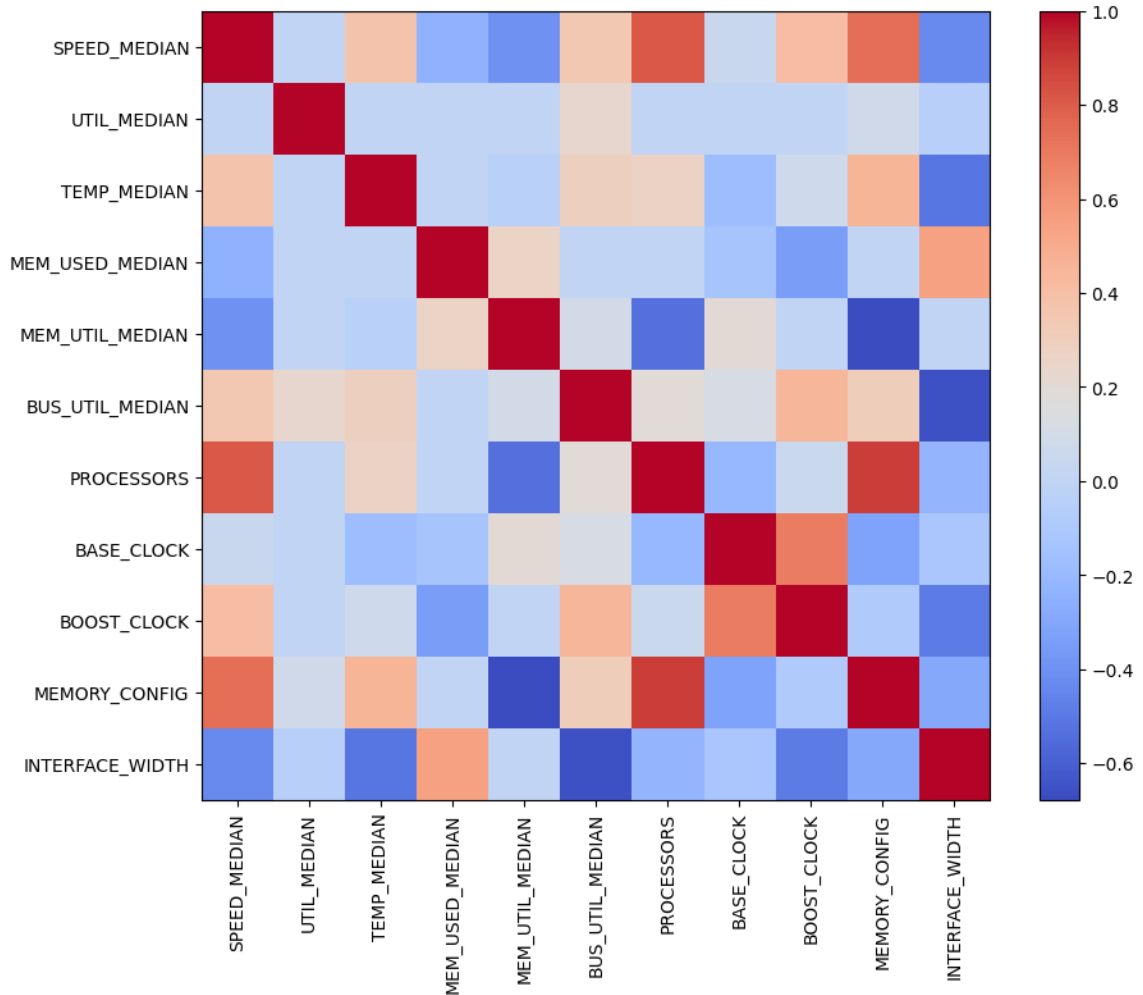
Figure 8.18: Median from correlation matrix of dict3 configuration median data.

Dict2 and Dict4

In these configurations, we used the same dictionaries as in dict1 and dict3, where dict4 has the RockYou wordlist. However, we used a rule file with 100 rules, meaning the hashes will calculate slower, impacting the analysis results. Figure 8.19 shows the correlation matrix of dict2 data, and Figure 8.20 shows the correlation matrix of dict4 data. As these matrixes show, the clock speed is the more critical hardware parameter for the dict2 configuration, but the number of processors is the most critical hardware parameter for the dict4 configuration. These essential hardware parameters differ because we can fit the dict2 namespace into the provided number of processors, and then their speed is more

important. The takeaway from this is if we have only a few candidate passwords, the clock speed at which the processors are running is more important than their number because we cannot utilise the high number of processors in this case. However, in most cases of password cracking, the number of candidate passwords is enormous, and we can utilise the processors, so overall, the number of processors is more important.

Just like in other cases, memory is not that important overall, and after reaching a certain value of used memory, the tests do not use any more. Of course, some hashes need more memory, and having more memory is beneficial in those cases. Nevertheless, overall, it is not that important. Sections 8.1.4, 8.1.5 and 8.1.6 better describe these cases.

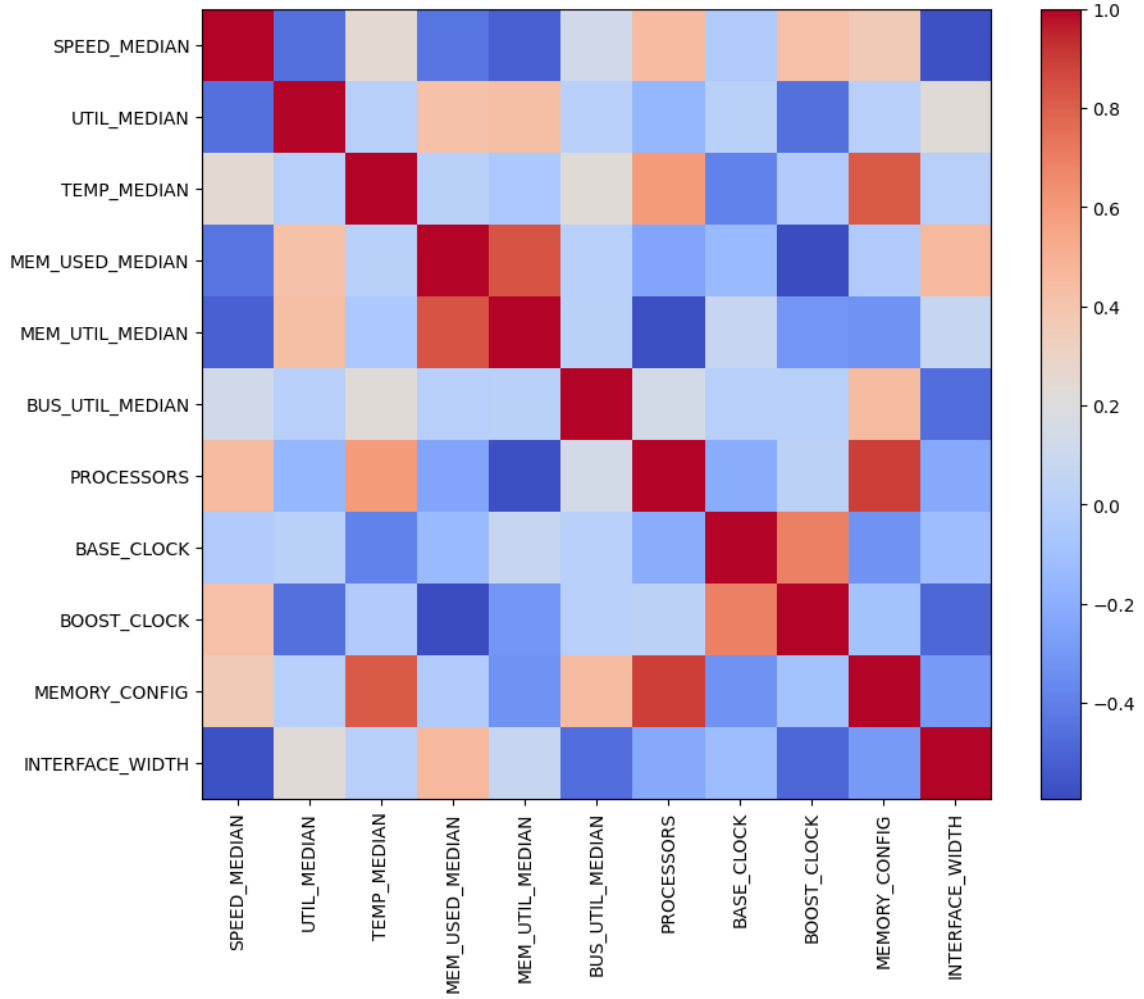|	SPEED_MEDIAN	MEM_USED_MEDIAN	MEM_UTIL_MEDIAN	PROCESSORS	BASE_CLOCK	BOOST_CLOCK	MEMORY_CONFIG	INTERFACE_WIDTH
SPEED_MEDIAN	1.000000	-0.035499	0.000000	0.396864	0.000000	0.490010	0.360722	-0.595925
MEM_USED_MEDIAN	-0.035499	1.000000	0.556741	-0.142238	-0.091216	-0.380686	0.001072	0.049558
MEM_UTIL_MEDIAN	0.000000	0.556741	1.000000	-0.634394	0.309460	0.000000	-0.558541	-0.044152
PROCESSORS	0.396864	-0.142238	-0.634394	1.000000	-0.210624	0.055271	0.890509	-0.221247
BASE_CLOCK	0.000000	-0.091216	0.309460	-0.210624	1.000000	0.691414	-0.322779	-0.116461
BOOST_CLOCK	0.490010	-0.380686	0.000000	0.055271	0.691414	1.000000	-0.087469	-0.490223
MEMORY_CONFIG	0.360722	0.001072	-0.558541	0.890509	-0.322779	-0.087469	1.000000	-0.292292
INTERFACE_WIDTH	-0.595925	0.049558	-0.044152	-0.221247	-0.116461	-0.490223	-0.292292	1.000000

Figure 8.19: Median from correlation matrix of dict2 configuration median data.

66

	SPEED_MEDIAN	MEM_USED_MEDIAN	MEM_UTIL_MEDIAN	PROCESSORS	BASE_CLOCK	BOOST_CLOCK	MEMORY_CONFIG	INTERFACE_WIDTH
SPEED_MEDIAN	1.000000	-0.494877	-0.599770	0.787236	0.167532	0.437342	0.725934	-0.543731
MEM_USED_MEDIAN	-0.494877	1.000000	0.808603	-0.086518	-0.196832	-0.557605	-0.127309	0.977544
MEM_UTIL_MEDIAN	-0.599770	0.808603	1.000000	-0.474928	0.105087	-0.230157	-0.620187	0.775369
PROCESSORS	0.787236	-0.086518	-0.474928	1.000000	-0.210624	0.055271	0.890509	-0.221247
BASE_CLOCK	0.167532	-0.196832	0.105087	-0.210624	1.000000	0.691414	-0.322779	-0.116461
BOOST_CLOCK	0.437342	-0.557605	-0.230157	0.055271	0.691414	1.000000	-0.087469	-0.490223
MEMORY_CONFIG	0.725934	-0.127309	-0.620187	0.890509	-0.322779	-0.087469	1.000000	-0.292292
INTERFACE_WIDTH	-0.543731	0.977544	0.775369	-0.221247	-0.116461	-0.490223	-0.292292	1.000000
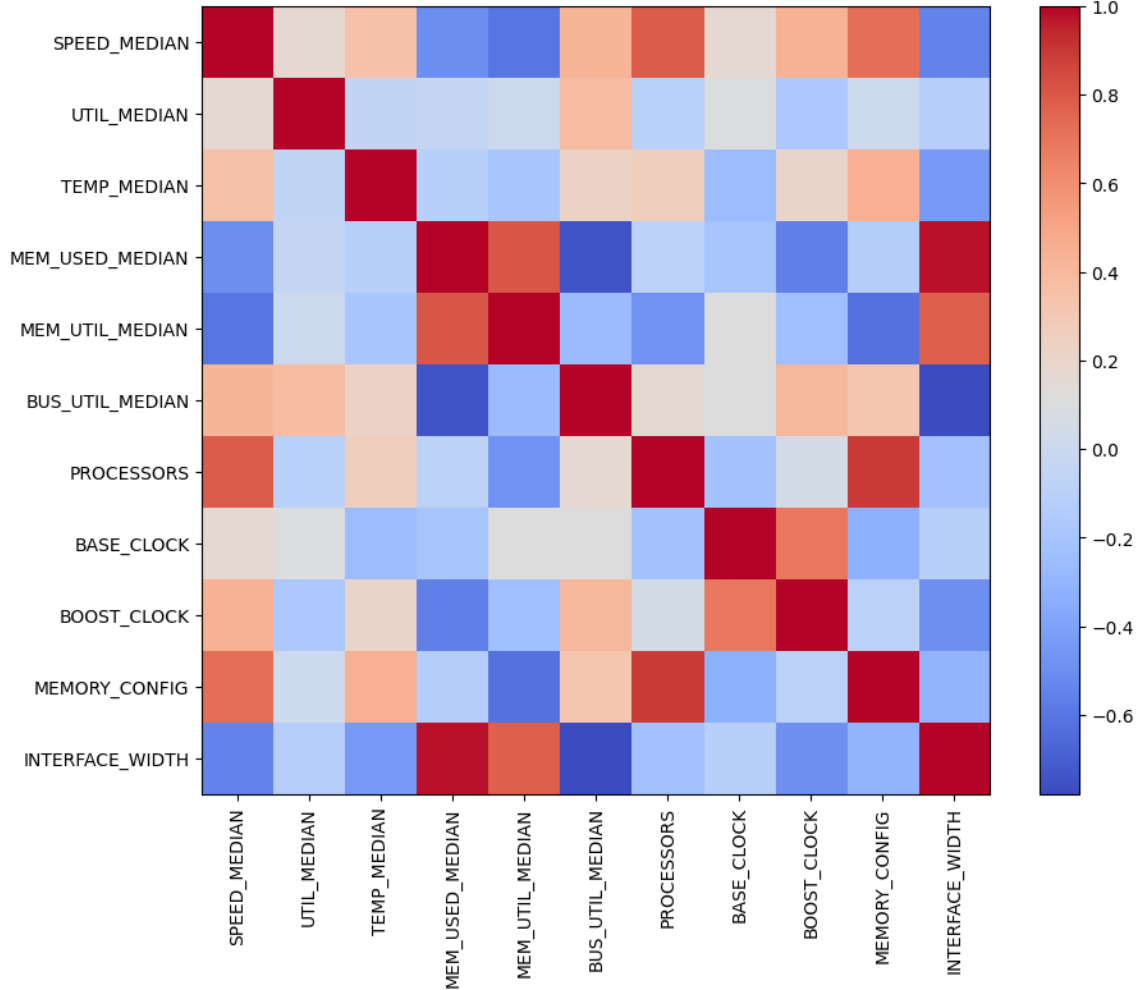
Figure 8.20: Median from correlation matrix of dict4 configuration median data.

Force Attack

Force attack, also known as Mask attack, is an attack that is most reliant on the number of processors on the GPU and their clock speeds, and that is because all of the calculations of the candidate hashes are ongoing on the GPU itself. It does not load any data input from the computer. The correlation matrix shown in Figure 8.21 shows us that the number of processors is the most critical hardware parameter for force type of attack. For memory, the case is the same as in the Combinator attack explained in sections 8.1.4 and 8.1.5 and Section 8.1.6.

	SPEED_MEDIAN	MEM_USED_MEDIAN	MEM_UTIL_MEDIAN	PROCESSORS	BASE_CLOCK	BOOST_CLOCK	MEMORY_CONFIG	INTERFACE_WIDTH
SPEED_MEDIAN	1.000000	-0.261964	-0.277006	0.757145	0.000000	0.448279	0.624412	-0.409669
MEM_USED_MEDIAN	-0.261964	1.000000	0.820087	-0.192377	-0.170896	-0.518242	-0.131255	0.544107
MEM_UTIL_MEDIAN	-0.277006	0.820087	1.000000	-0.302357	0.000000	-0.110923	-0.365814	0.140001
PROCESSORS	0.757145	-0.192377	-0.302357	1.000000	-0.210624	0.055271	0.890509	-0.221247
BASE_CLOCK	0.000000	-0.170896	0.000000	-0.210624	1.000000	0.691414	-0.322779	-0.116461
BOOST_CLOCK	0.448279	-0.518242	-0.110923	0.055271	0.691414	1.000000	-0.087469	-0.490223
MEMORY_CONFIG	0.624412	-0.131255	-0.365814	0.890509	-0.322779	-0.087469	1.000000	-0.292292
INTERFACE_WIDTH	-0.409669	0.544107	0.140001	-0.221247	-0.116461	-0.490223	-0.292292	1.000000

Figure 8.21: Median from correlation matrix of force configuration median data.

Hybrid attack

The Hybrid attack combines the dictionary attack and the force attack into one. It functions like a combinator attack, combining each word from an inputted dictionary with each password created by force attack. The correlation matrix shown in Figure 8.22 shows that the most critical hardware characteristic is the processor count, followed by memory and then the boost clock speed. On the other hand, the memory utilisation and the used memory values suggest that memory is not that important. Whether the memory is important depends on the hash type and is further explained in the sections 8.1.4, 8.1.5 and 8.1.6.

	SPEED_MEDIAN	MEM_USED_MEDIAN	MEM_UTIL_MEDIAN	PROCESSORS	BASE_CLOCK	BOOST_CLOCK	MEMORY_CONFIG	INTERFACE_WIDTH
SPEED_MEDIAN	1.000000	-0.457953	-0.571464	0.800311	0.005909	0.499362	0.668831	-0.452906
MEM_USED_MEDIAN	-0.457953	1.000000	0.863387	-0.156009	-0.211784	-0.582185	-0.158786	0.920491
MEM_UTIL_MEDIAN	-0.571464	0.863387	1.000000	-0.429367	-0.019922	-0.369002	-0.446813	0.732072
PROCESSORS	0.800311	-0.156009	-0.429367	1.000000	-0.210624	0.055271	0.890509	-0.221247
BASE_CLOCK	0.005909	-0.211784	-0.019922	-0.210624	1.000000	0.691414	-0.322779	-0.116461
BOOST_CLOCK	0.499362	-0.582185	-0.369002	0.055271	0.691414	1.000000	-0.087469	-0.490223
MEMORY_CONFIG	0.668831	-0.158786	-0.446813	0.890509	-0.322779	-0.087469	1.000000	-0.292292
INTERFACE_WIDTH	-0.452906	0.920491	0.732072	-0.221247	-0.116461	-0.490223	-0.292292	1.000000
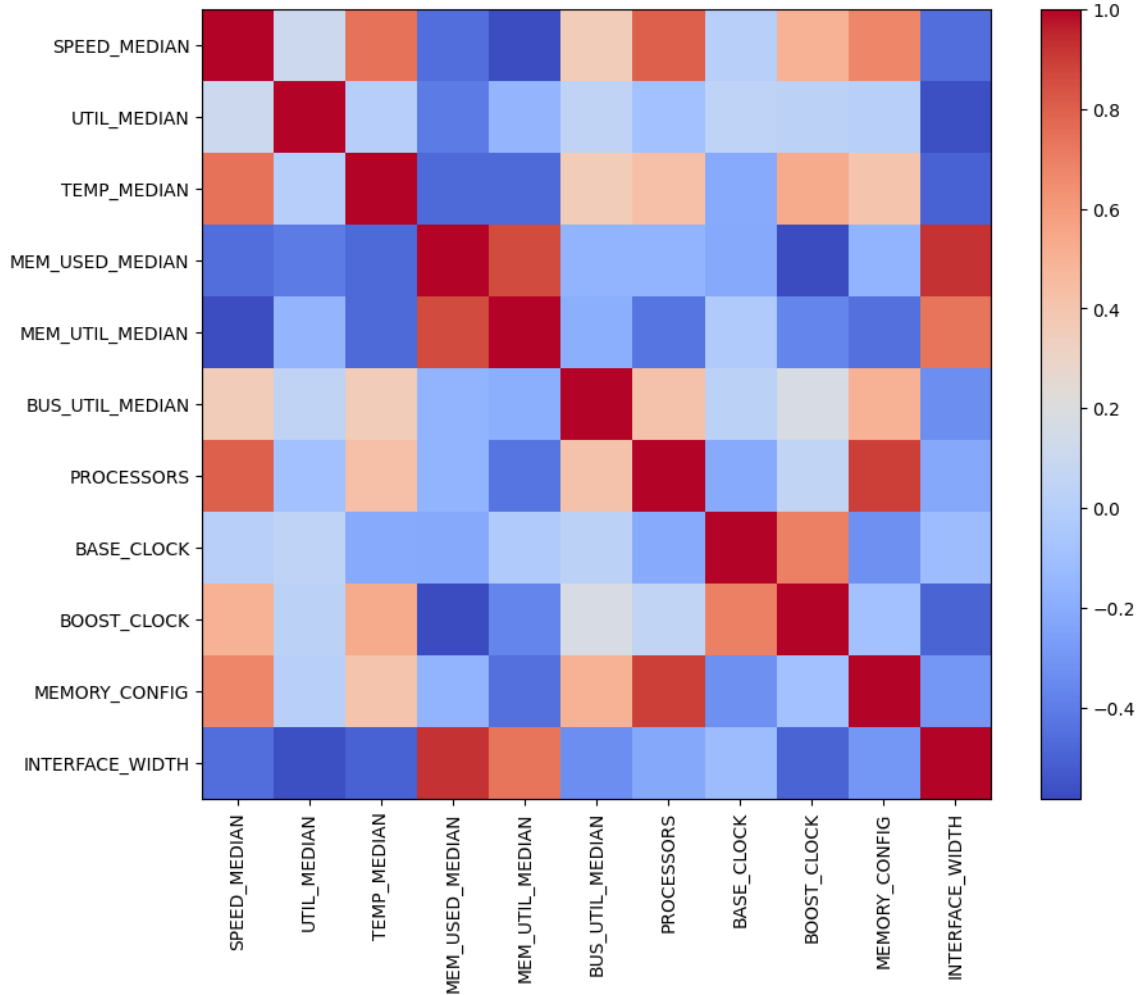
Figure 8.22: Median from correlation matrix of force configuration median data.

8.1.6 Memory-hard Hash Functions

A memory-hard hash function is a function that has to spend a large amount of memory when calculating the hash, or the password-cracking performance will get slower. In the sections 8.1.4 and 8.1.5 we have experienced a disagreement about whether large memory is essential or not. In this section, we look at hashes that can use the extra memory and benefit from it. First, we calculate the correlation matrix from median values for every file with a memory utilisation value higher than 70%. We will further analyse these hashes to see which hardware parameters are most important for these memory-hungry hashes by looking at the correlation between hardware parameters and speed.

The hashes that we filtered from the rest are:

- DiskCryptor SHA512 + XTS 1024 bit (Serpent-AES)

- DiskCryptor SHA512 + XTS 1024 bit (Twofish-Serpent)

- DiskCryptor SHA512 + XTS 512 bit (Serpent)

- ExodusDesktopWallet(scrypt)

- MultiBitClassic.wallet(scrypt)

- MultiBitHD(scrypt)

- SNMPv3HMAC-MD5-96-HMAC-SHA1-968

- SNMPv3HMAC-MD5-968

- SNMPv3HMAC-SHA1-968

- SNMPv3HMAC-SHA224-1288

- SNMPv3HMAC-SHA256-19288

- SNMPv3HMAC-SHA384-2568

- SNMPv3HMAC-SHA512-3848

- SolarWindsOrion

- SolarWindsOrionv2

- TrueCrypt 5.0+ PBKDF2-HMAC-RIPEMD160 + Serpent-AES + boot (legacy)

The correlation matrixes shown in Figures from 8.23 to 8.38 show that the two most crucial hardware characteristics for these hashes are the processor count and the memory size. The correlation matrix of DiskCryptor SHA512 + XTS 512 bit (Serpent) even shows us that memory is more important than processor count for this hash. The hashes that require much memory while being cracked are really complex. These hashes usually belong to either disk encryptions, wallet encryption, network management protocol encryptions or administration platform encryptions.

Overall this means that the bigger size of the memory is, the better. It is not always fully used, because not all hashes are that complex and need the full use of it. However, if we try to crack complex hash, or we use complex settings with the use of lot of rules, the memory can be used up quite quickly.

	SPEED_MEDIAN	MEM_USED_MEDIAN	MEM_UTIL_MEDIAN	PROCESSORS	BASE_CLOCK	BOOST_CLOCK	MEMORY_CONFIG	INTERFACE_WIDTH
SPEED_MEDIAN	1.000000	0.160392	0.025491	0.403763	0.118845	0.357478	0.362246	-0.392990
MEM_USED_MEDIAN	0.160392	1.000000	0.294551	0.127624	-0.131338	-0.179209	0.284965	-0.303419
MEM_UTIL_MEDIAN	0.025491	0.294551	1.000000	-0.577520	0.500322	0.453737	-0.620318	-0.462425
PROCESSORS	0.403763	0.127624	-0.577520	1.000000	-0.210624	0.055271	0.890509	-0.221247
BASE_CLOCK	0.118845	-0.131338	0.500322	-0.210624	1.000000	0.691414	-0.322779	-0.116461
BOOST_CLOCK	0.357478	-0.179209	0.453737	0.055271	0.691414	1.000000	-0.087469	-0.490223
MEMORY_CONFIG	0.362246	0.284965	-0.620318	0.890509	-0.322779	-0.087469	1.000000	-0.292292
INTERFACE_WIDTH	-0.392990	-0.303419	-0.462425	-0.221247	-0.116461	-0.490223	-0.292292	1.000000

Figure 8.23: Median from correlation matrix of data from DiskCryptor SHA512 + XTS 1024 bit (Serpent-AES) hash.

71

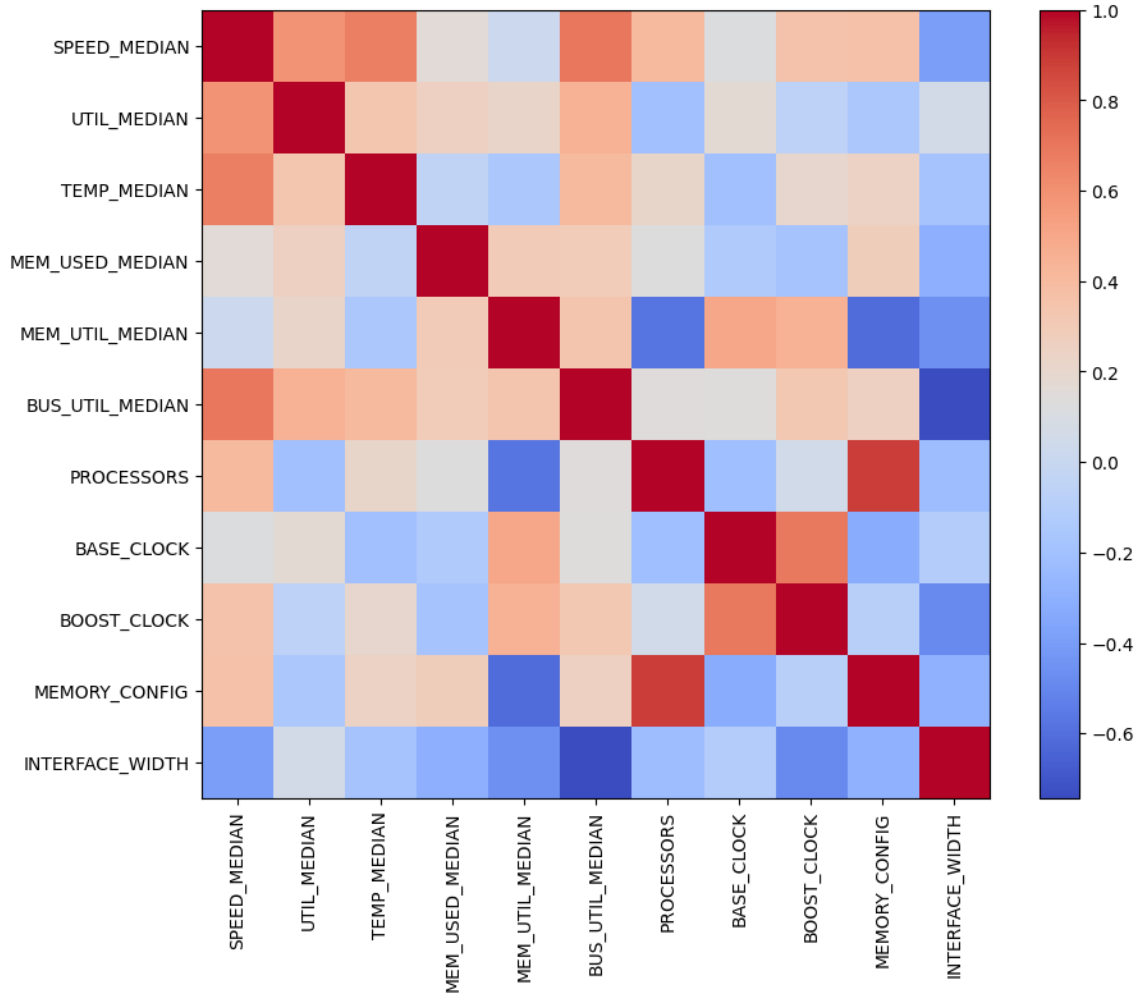|	SPEED_MEDIAN	MEM_USED_MEDIAN	MEM_UTIL_MEDIAN	PROCESSORS	BASE_CLOCK	BOOST_CLOCK	MEMORY_CONFIG	INTERFACE_WIDTH
SPEED_MEDIAN	1.000000	0.171793	-0.015642	0.483569	0.136548	0.365383	0.428176	-0.398010
MEM_USED_MEDIAN	0.171793	1.000000	0.292087	0.126269	-0.131212	-0.181838	0.282925	-0.296810
MEM_UTIL_MEDIAN	-0.015642	0.292087	1.000000	-0.578788	0.500981	0.453301	-0.622109	-0.459819
PROCESSORS	0.483569	0.126269	-0.578788	1.000000	-0.210624	0.055271	0.890509	-0.221247
BASE_CLOCK	0.136548	-0.131212	0.500981	-0.210624	1.000000	0.691414	-0.322779	-0.116461
BOOST_CLOCK	0.365383	-0.181838	0.453301	0.055271	0.691414	1.000000	-0.087469	-0.490223
MEMORY_CONFIG	0.428176	0.282925	-0.622109	0.890509	-0.322779	-0.087469	1.000000	-0.292292
INTERFACE_WIDTH	-0.398010	-0.296810	-0.459819	-0.221247	-0.116461	-0.490223	-0.292292	1.000000

Figure 8.24: Median from correlation matrix of data from DiskCryptor SHA512 + XTS 1024 bit (Twofish-Serpent) hash.

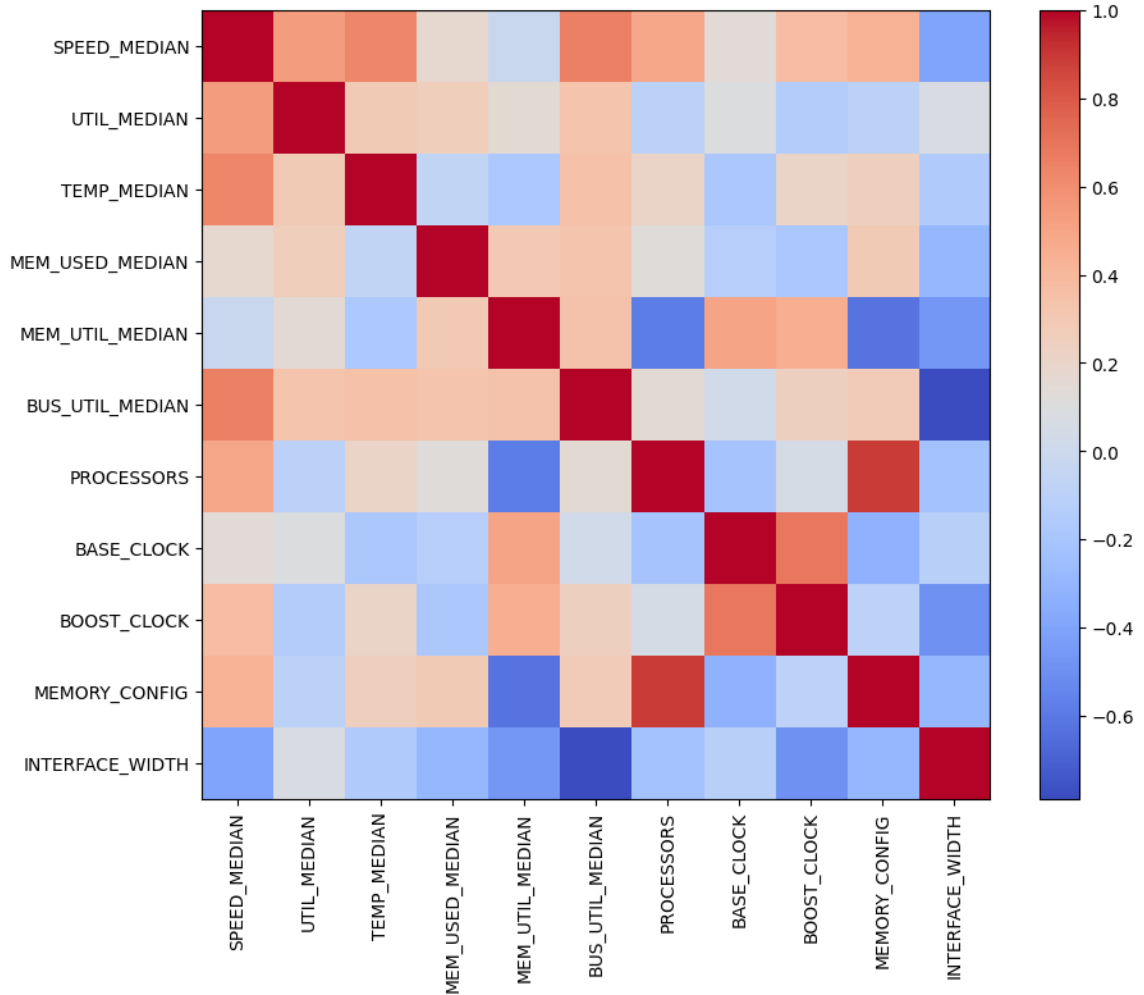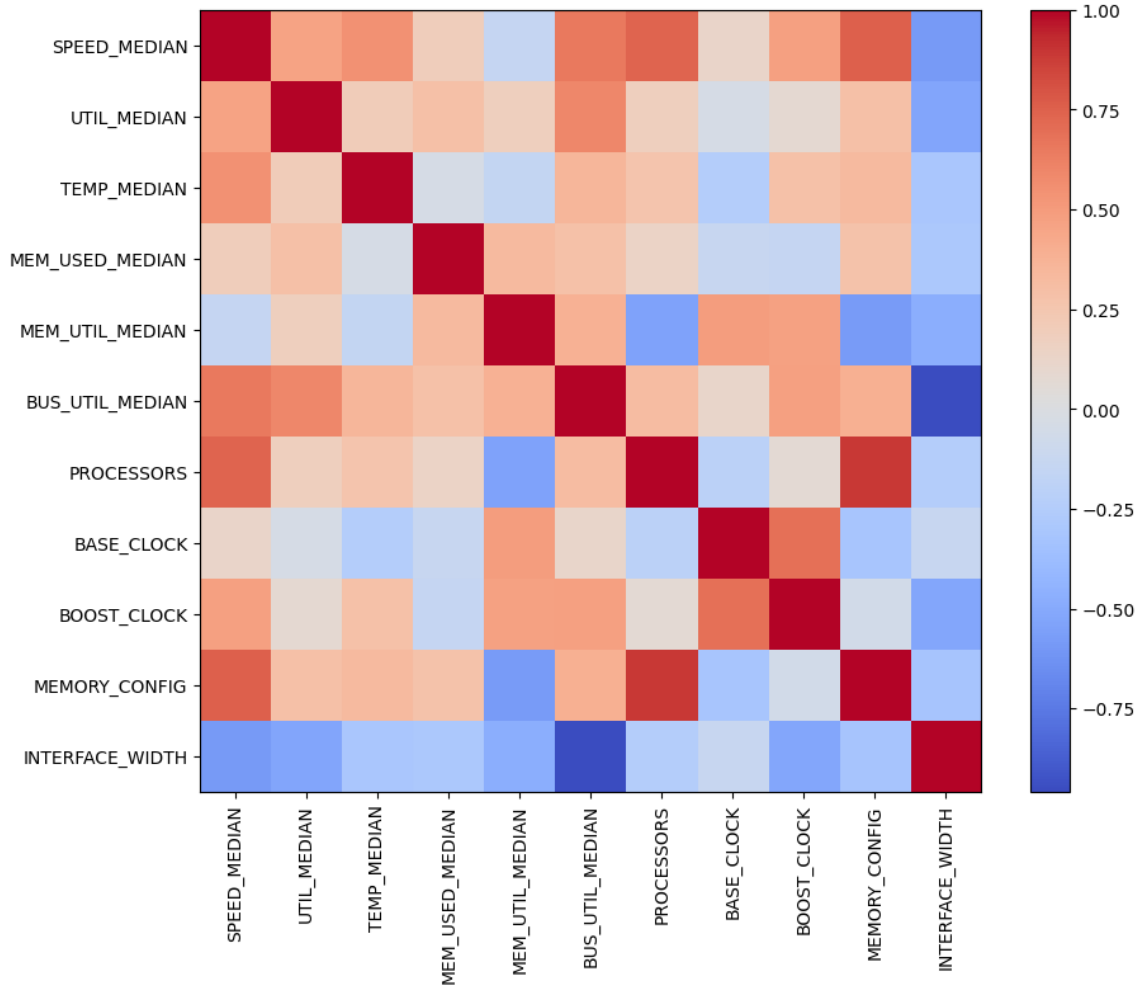|	SPEED_MEDIAN	MEM_USED_MEDIAN	MEM_UTIL_MEDIAN	PROCESSORS	BASE_CLOCK	BOOST_CLOCK	MEMORY_CONFIG	INTERFACE_WIDTH
SPEED_MEDIAN	1.000000	0.194538	-0.144971	0.738935	0.122559	0.472295	0.749381	-0.586002
MEM_USED_MEDIAN	0.194538	1.000000	0.330079	0.136994	-0.125590	-0.142589	0.279687	-0.291141
MEM_UTIL_MEDIAN	-0.144971	0.330079	1.000000	-0.541082	0.492889	0.468447	-0.578454	-0.475755
PROCESSORS	0.738935	0.136994	-0.541082	1.000000	-0.203636	0.073534	0.891605	-0.243036
BASE_CLOCK	0.122559	-0.125590	0.492889	-0.203636	1.000000	0.688171	-0.312082	-0.127724
BOOST_CLOCK	0.472295	-0.142589	0.468447	0.073534	0.688171	1.000000	-0.059611	-0.516681
MEMORY_CONFIG	0.749381	0.279687	-0.578454	0.891605	-0.312082	-0.059611	1.000000	-0.319432
INTERFACE_WIDTH	-0.586002	-0.291141	-0.475755	-0.243036	-0.127724	-0.516681	-0.319432	1.000000

Figure 8.25: Median from correlation matrix of data from DiskCryptor SHA512 + XTS 512 bit (Serpent) hash.

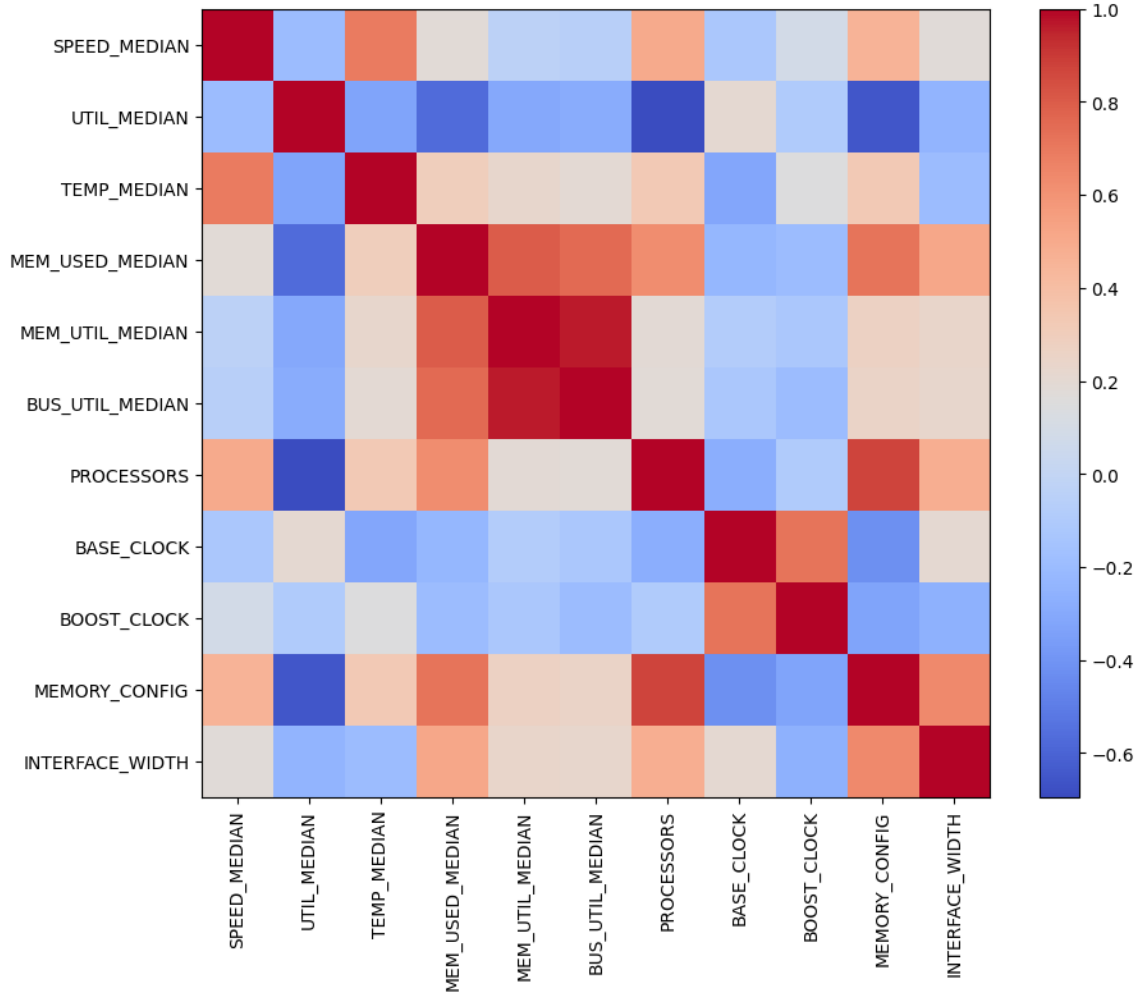|	SPEED_MEDIAN	MEM_USED_MEDIAN	MEM_UTIL_MEDIAN	PROCESSORS	BASE_CLOCK	BOOST_CLOCK	MEMORY_CONFIG	INTERFACE_WIDTH
SPEED_MEDIAN	1.000000	0.181776	-0.034838	0.503041	-0.120274	0.090687	0.456755	0.173898
MEM_USED_MEDIAN	0.181776	1.000000	0.800927	0.622752	-0.230736	-0.197439	0.715899	0.510356
MEM_UTIL_MEDIAN	-0.034838	0.800927	1.000000	0.189888	-0.079804	-0.119308	0.265938	0.237934
PROCESSORS	0.503041	0.622752	0.189888	1.000000	-0.277063	-0.097425	0.871557	0.478588
BASE_CLOCK	-0.120274	-0.230736	-0.079804	-0.277063	1.000000	0.720294	-0.422371	0.203539
BOOST_CLOCK	0.090687	-0.197439	-0.119308	-0.097425	0.720294	1.000000	-0.329477	-0.262936
MEMORY_CONFIG	0.456755	0.715899	0.265938	0.871557	-0.422371	-0.329477	1.000000	0.640097
INTERFACE_WIDTH	0.173898	0.510356	0.237934	0.478588	0.203539	-0.262936	0.640097	1.000000

Figure 8.26: Median from correlation matrix of data from ExodusDesktopWallet(scrypt) hash.

	SPEED_MEDIAN	MEM_USED_MEDIAN	MEM_UTIL_MEDIAN	PROCESSORS	BASE_CLOCK	BOOST_CLOCK	MEMORY_CONFIG	INTERFACE_WIDTH
SPEED_MEDIAN	1.000000	0.486874	0.409981	0.409776	-0.049080	0.191688	0.396181	-0.288339
MEM_USED_MEDIAN	0.486874	1.000000	0.797540	0.628412	-0.196380	-0.087688	0.701242	-0.112542
MEM_UTIL_MEDIAN	0.409981	0.797540	1.000000	0.232992	-0.018674	0.052604	0.295064	-0.244524
PROCESSORS	0.409776	0.628412	0.232992	1.000000	-0.232823	0.036187	0.879033	-0.216785
BASE_CLOCK	-0.049080	-0.196380	-0.018674	-0.232823	1.000000	0.691100	-0.350024	-0.114986
BOOST_CLOCK	0.191688	-0.087688	0.052604	0.036187	0.691100	1.000000	-0.113726	-0.488585
MEMORY_CONFIG	0.396181	0.701242	0.295064	0.879033	-0.350024	-0.113726	1.000000	-0.291366
INTERFACE_WIDTH	-0.288339	-0.112542	-0.244524	-0.216785	-0.114986	-0.488585	-0.291366	1.000000

Figure 8.27: Median from correlation matrix of data from MultiBitClassic.wallet(scrypt) hash.

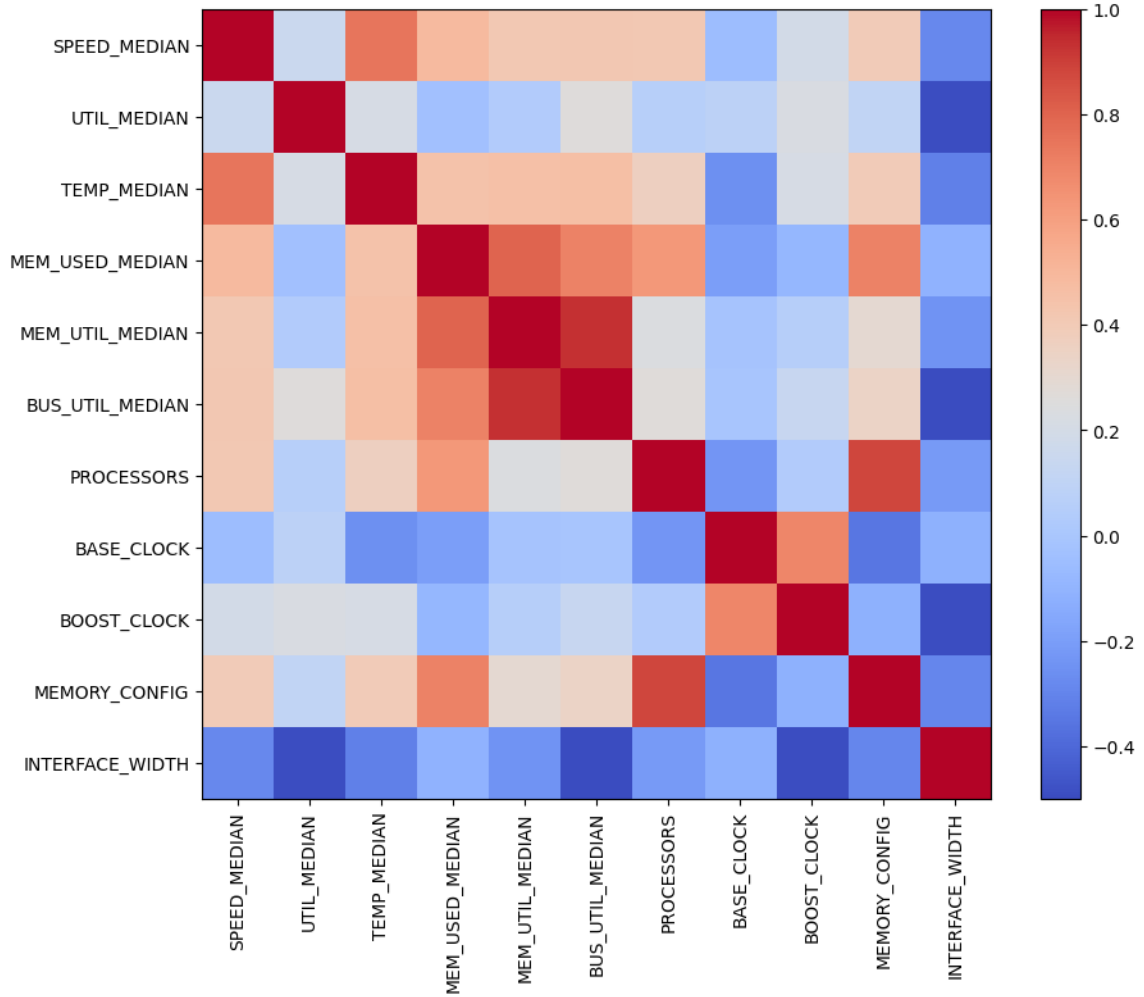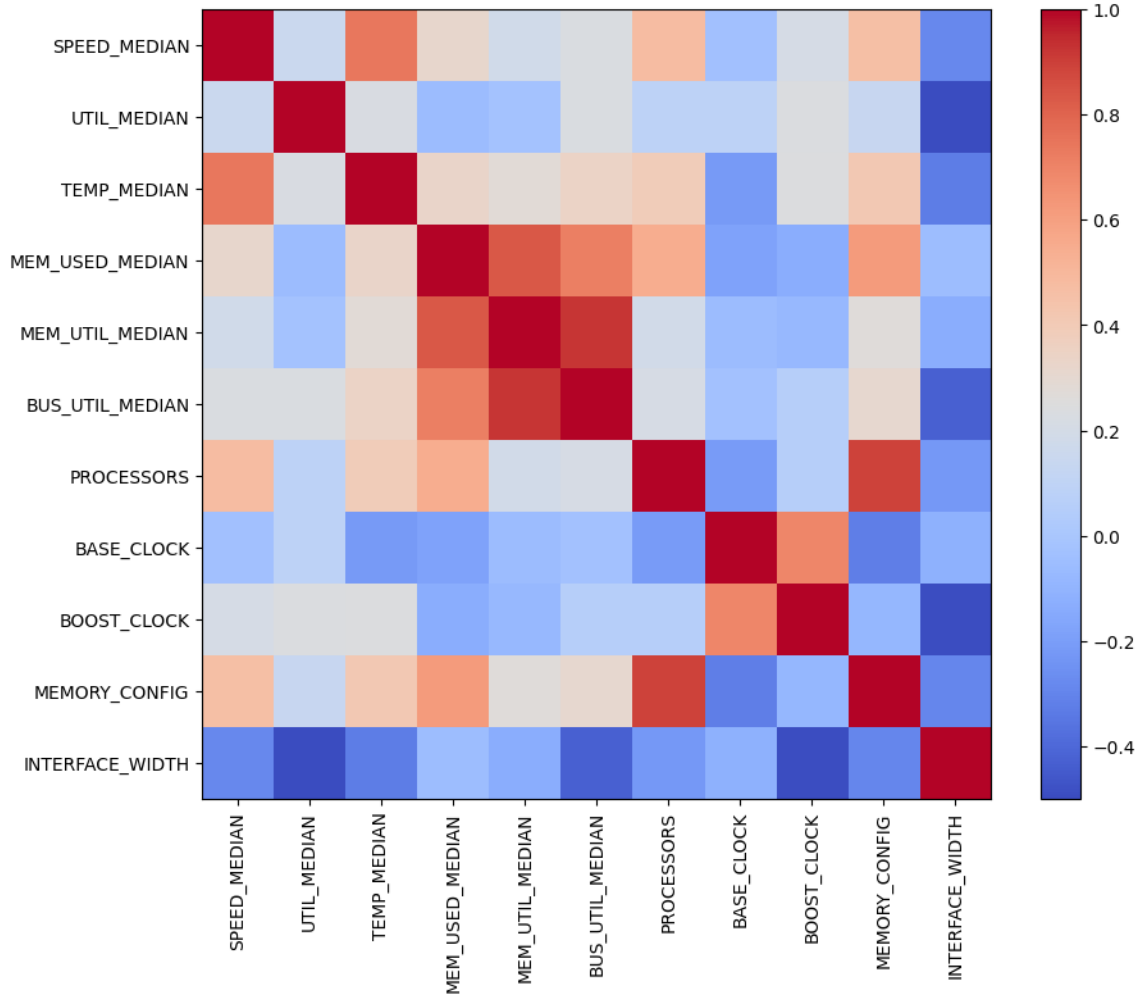|	SPEED_MEDIAN	MEM_USED_MEDIAN	MEM_UTIL_MEDIAN	PROCESSORS	BASE_CLOCK	BOOST_CLOCK	MEMORY_CONFIG	INTERFACE_WIDTH
SPEED_MEDIAN	1.000000	0.317015	0.183983	0.475637	-0.034818	0.205411	0.461332	-0.286567
MEM_USED_MEDIAN	0.317015	1.000000	0.833547	0.543791	-0.179108	-0.131415	0.614005	-0.051956
MEM_UTIL_MEDIAN	0.183983	0.833547	1.000000	0.191355	-0.056263	-0.078608	0.265199	-0.133975
PROCESSORS	0.475637	0.543791	0.191355	1.000000	-0.210624	0.055271	0.890509	-0.221247
BASE_CLOCK	-0.034818	-0.179108	-0.056263	-0.210624	1.000000	0.691414	-0.322779	-0.116461
BOOST_CLOCK	0.205411	-0.131415	-0.078608	0.055271	0.691414	1.000000	-0.087469	-0.490223
MEMORY_CONFIG	0.461332	0.614005	0.265199	0.890509	-0.322779	-0.087469	1.000000	-0.292292
INTERFACE_WIDTH	-0.286567	-0.051956	-0.133975	-0.221247	-0.116461	-0.490223	-0.292292	1.000000

Figure 8.28: Median from correlation matrix of data from MultiBitHD(scrypt) hash.

	SPEED_MEDIAN	MEM_USED_MEDIAN	MEM_UTIL_MEDIAN	PROCESSORS	BASE_CLOCK	BOOST_CLOCK	MEMORY_CONFIG	INTERFACE_WIDTH
SPEED_MEDIAN	1.000000	-0.271812	-0.438191	0.377305	0.050536	0.263990	0.300223	-0.138427
MEM_USED_MEDIAN	-0.271812	1.000000	0.818322	0.520675	-0.227206	-0.224099	0.685370	-0.078442
MEM_UTIL_MEDIAN	-0.438191	0.818322	1.000000	0.066510	-0.083174	-0.214966	0.320089	-0.168482
PROCESSORS	0.377305	0.520675	0.066510	1.000000	-0.217108	0.080604	0.887082	-0.233568
BASE_CLOCK	0.050536	-0.227206	-0.083174	-0.217108	1.000000	0.673779	-0.331388	-0.092098
BOOST_CLOCK	0.263990	-0.224099	-0.214966	0.080604	0.673779	1.000000	-0.070221	-0.478464
MEMORY_CONFIG	0.300223	0.685370	0.320089	0.887082	-0.331388	-0.070221	1.000000	-0.322478
INTERFACE_WIDTH	-0.138427	-0.078442	-0.168482	-0.233568	-0.092098	-0.478464	-0.322478	1.000000

Figure 8.29: Median from correlation matrix of data SNMPv3HMAC-MD5-96-HMAC-SHA1-968 hash.

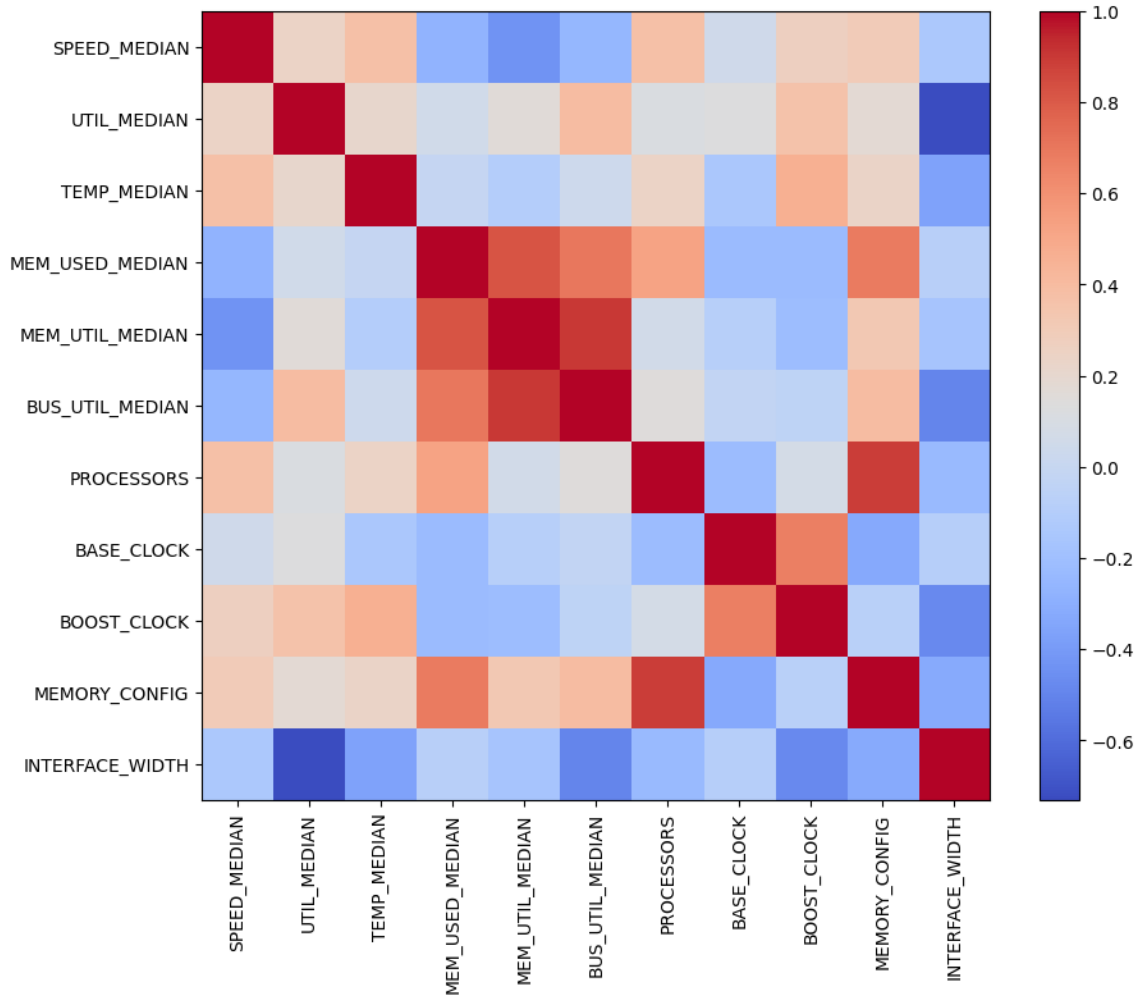|	SPEED_MEDIAN	MEM_USED_MEDIAN	MEM_UTIL_MEDIAN	PROCESSORS	BASE_CLOCK	BOOST_CLOCK	MEMORY_CONFIG	INTERFACE_WIDTH
SPEED_MEDIAN	1.000000	0.173439	-0.156182	0.464290	-0.008272	0.224195	0.381779	-0.100172
MEM_USED_MEDIAN	0.173439	1.000000	0.723057	0.702324	-0.259249	-0.199559	0.841741	-0.074353
MEM_UTIL_MEDIAN	-0.156182	0.723057	1.000000	0.183980	-0.028478	-0.058469	0.369942	-0.246806
PROCESSORS	0.464290	0.702324	0.183980	1.000000	-0.242908	0.027687	0.873549	-0.214967
BASE_CLOCK	-0.008272	-0.259249	-0.028478	-0.242908	1.000000	0.690970	-0.362396	-0.114374
BOOST_CLOCK	0.224195	-0.199559	-0.058469	0.027687	0.690970	1.000000	-0.125504	-0.487904
MEMORY_CONFIG	0.381779	0.841741	0.369942	0.873549	-0.362396	-0.125504	1.000000	-0.291168
INTERFACE_WIDTH	-0.100172	-0.074353	-0.246806	-0.214967	-0.114374	-0.487904	-0.291168	1.000000

Figure 8.30: Median from correlation matrix of data SNMPv3HMAC-MD5-968 hash.

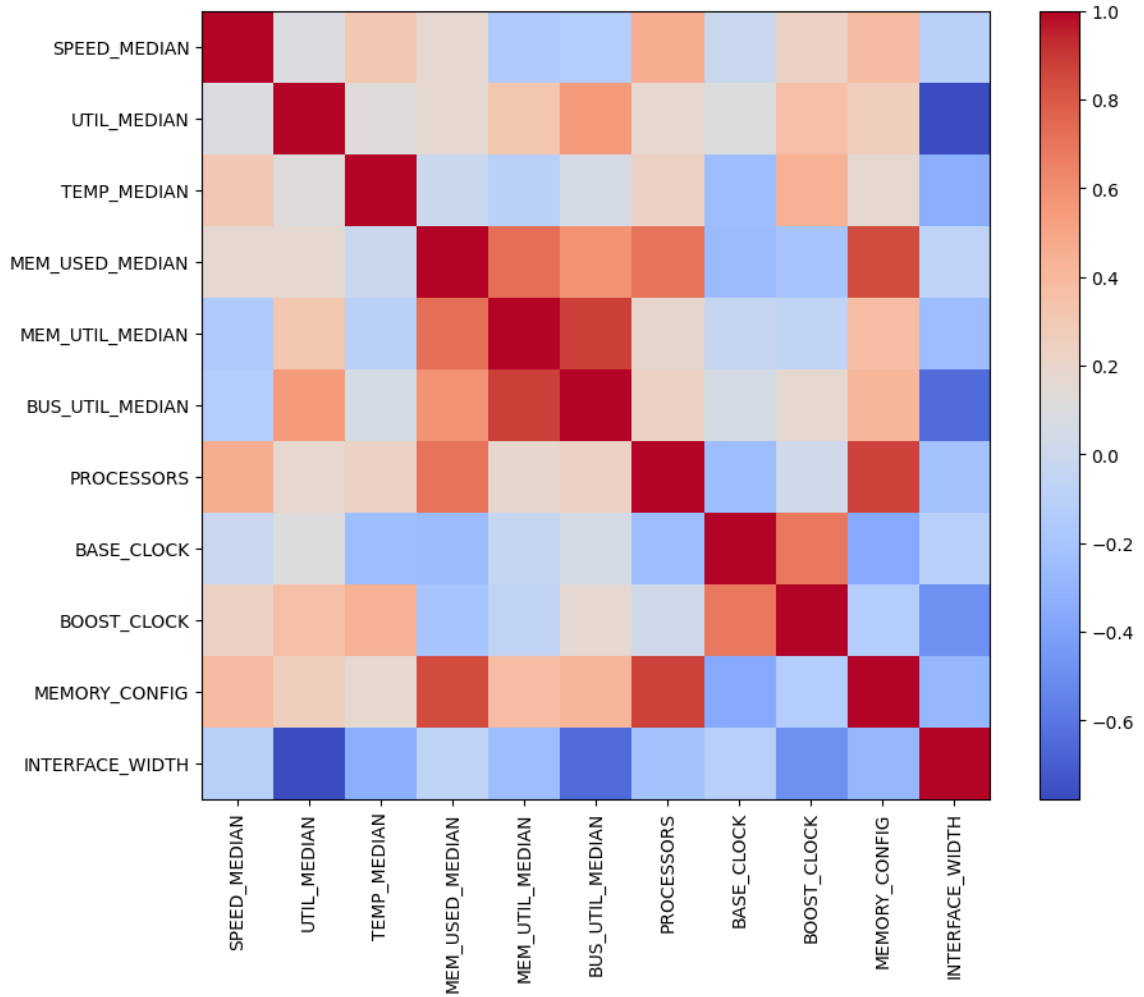|	SPEED_MEDIAN	MEM_USED_MEDIAN	MEM_UTIL_MEDIAN	PROCESSORS	BASE_CLOCK	BOOST_CLOCK	MEMORY_CONFIG	INTERFACE_WIDTH
SPEED_MEDIAN	1.000000	-0.043521	-0.300108	0.531538	0.003468	0.272465	0.454438	-0.184847
MEM_USED_MEDIAN	-0.043521	1.000000	0.719758	0.553051	-0.095073	-0.111263	0.647889	-0.089911
MEM_UTIL_MEDIAN	-0.300108	0.719758	1.000000	0.040166	0.211109	0.108907	0.127304	-0.290064
PROCESSORS	0.531538	0.553051	0.040166	1.000000	-0.236256	0.045424	0.874700	-0.235378
BASE_CLOCK	0.003468	-0.095073	0.211109	-0.236256	1.000000	0.687697	-0.351776	-0.125298
BOOST_CLOCK	0.272465	-0.111263	0.108907	0.045424	0.687697	1.000000	-0.097543	-0.513986
MEMORY_CONFIG	0.454438	0.647889	0.127304	0.874700	-0.351776	-0.097543	1.000000	-0.317398
INTERFACE_WIDTH	-0.184847	-0.089911	-0.290064	-0.235378	-0.125298	-0.513986	-0.317398	1.000000

Figure 8.31: Median from correlation matrix of data SNMPv3HMAC-SHA1-968 hash.

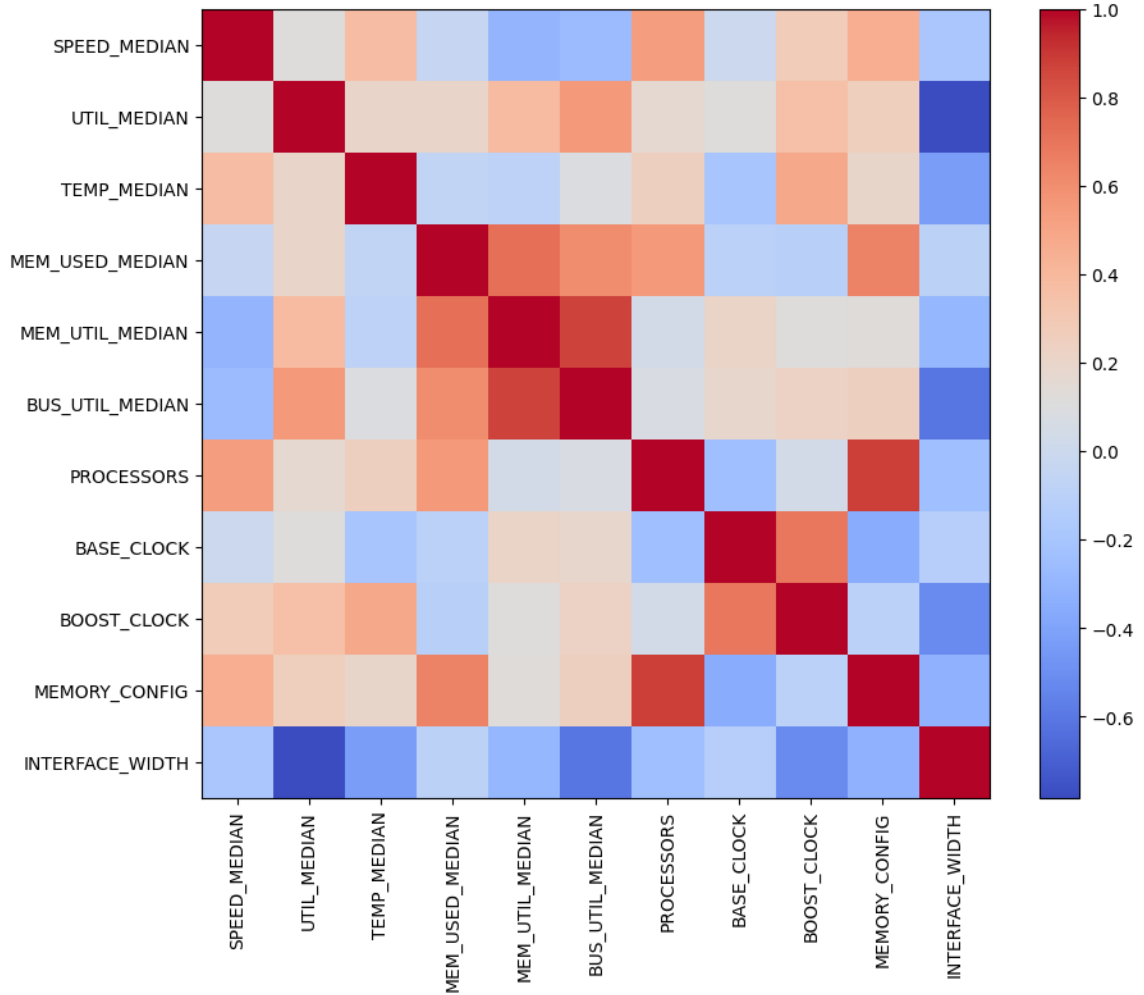|	SPEED_MEDIAN	MEM_USED_MEDIAN	MEM_UTIL_MEDIAN	PROCESSORS	BASE_CLOCK	BOOST_CLOCK	MEMORY_CONFIG	INTERFACE_WIDTH
SPEED_MEDIAN	1.000000	0.035763	-0.139670	0.461532	0.001429	0.238560	0.392240	-0.178811
MEM_USED_MEDIAN	0.035763	1.000000	0.671406	0.569673	-0.234221	-0.091568	0.687276	-0.264075
MEM_UTIL_MEDIAN	-0.139670	0.671406	1.000000	0.016083	0.135371	0.253222	0.108941	-0.603122
PROCESSORS	0.461532	0.569673	0.016083	1.000000	-0.236931	0.032712	0.876820	-0.216030
BASE_CLOCK	0.001429	-0.234221	0.135371	-0.236931	1.000000	0.691046	-0.355064	-0.114733
BOOST_CLOCK	0.238560	-0.091568	0.253222	0.032712	0.691046	1.000000	-0.118534	-0.488303
MEMORY_CONFIG	0.392240	0.687276	0.108941	0.876820	-0.355064	-0.118534	1.000000	-0.291269
INTERFACE_WIDTH	-0.178811	-0.264075	-0.603122	-0.216030	-0.114733	-0.488303	-0.291269	1.000000

Figure 8.32: Median from correlation matrix of data SNMPv3HMAC-SHA224-1288 hash.

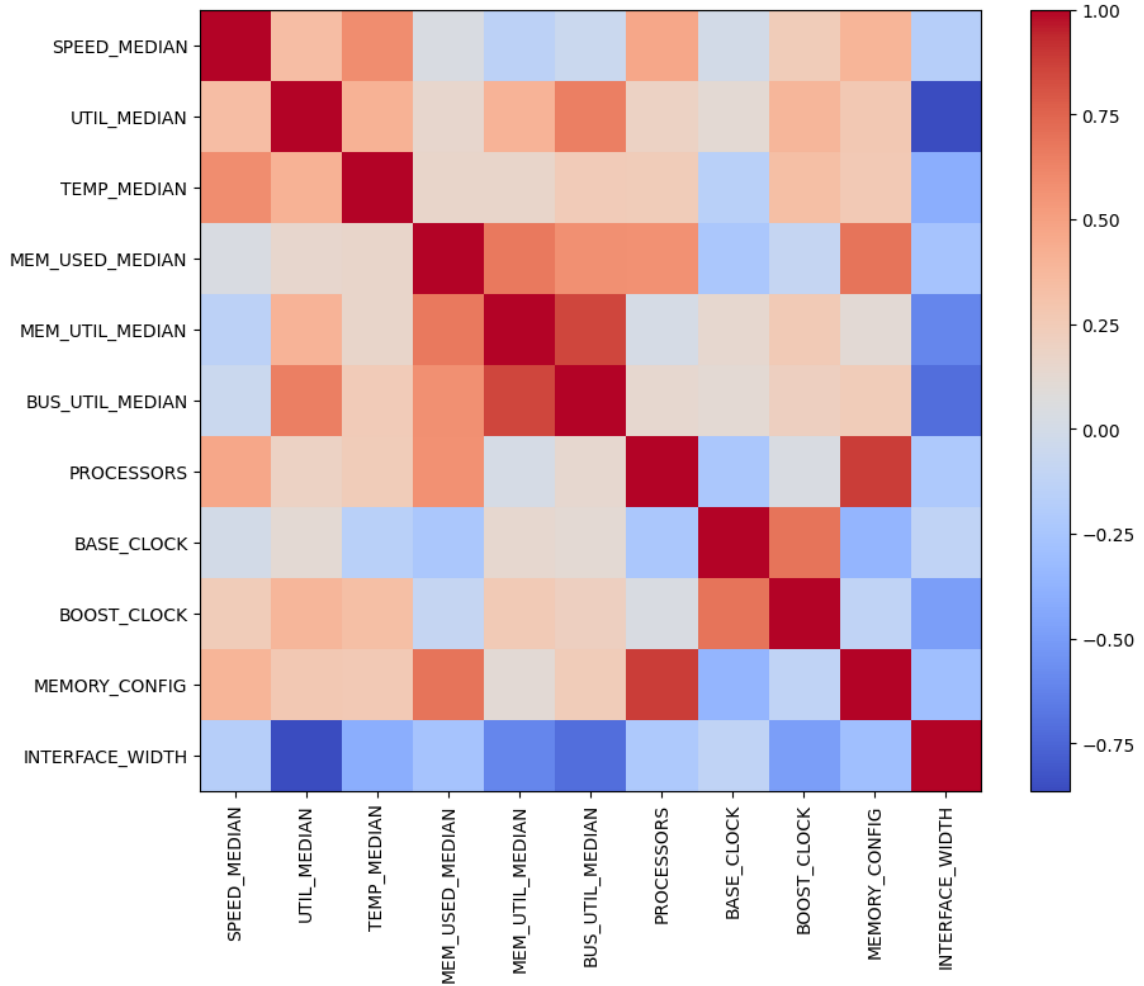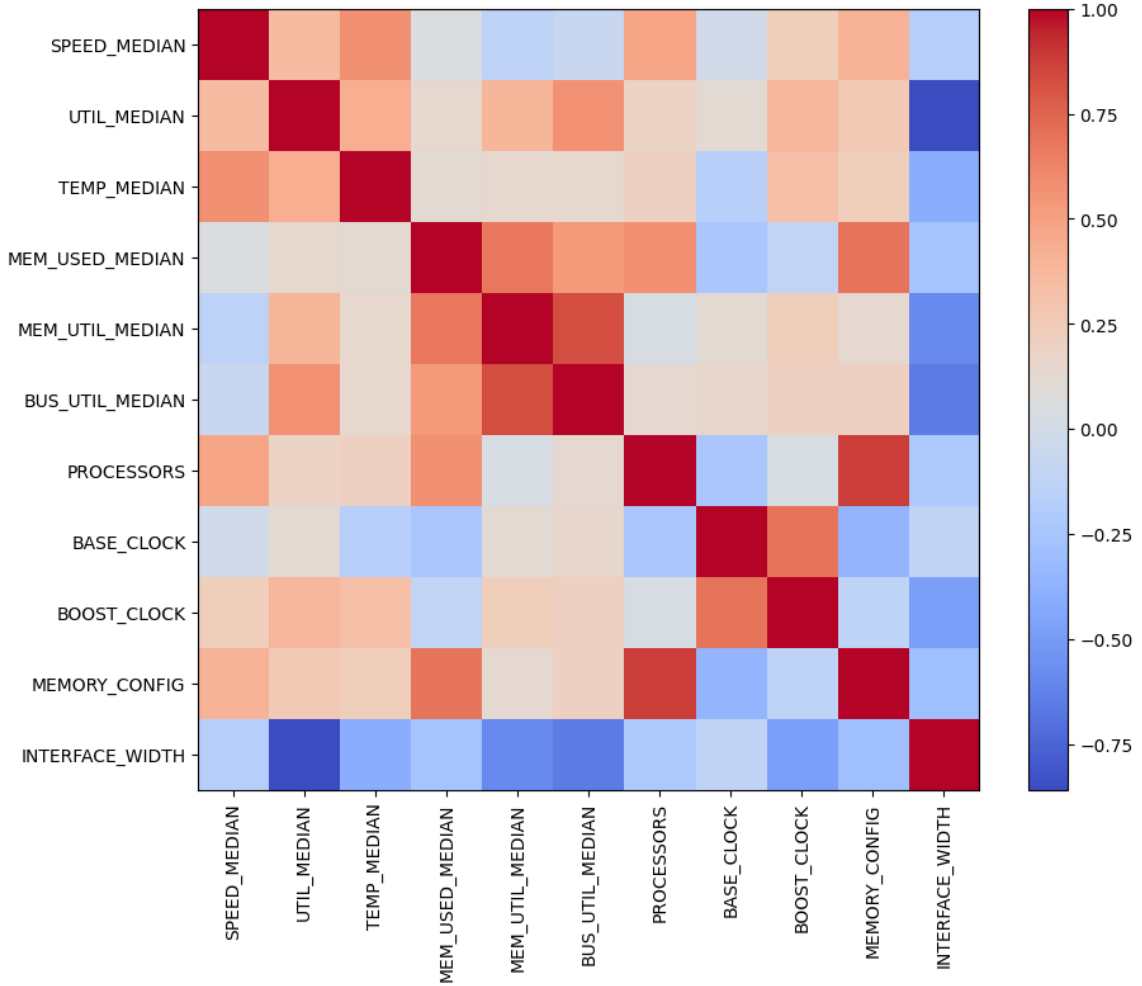|	SPEED_MEDIAN	MEM_USED_MEDIAN	MEM_UTIL_MEDIAN	PROCESSORS	BASE_CLOCK	BOOST_CLOCK	MEMORY_CONFIG	INTERFACE_WIDTH
SPEED_MEDIAN	1.000000	0.049136	-0.131992	0.470770	-0.010677	0.223612	0.399088	-0.175074
MEM_USED_MEDIAN	0.049136	1.000000	0.679772	0.571964	-0.239839	-0.101514	0.688775	-0.258510
MEM_UTIL_MEDIAN	-0.131992	0.679772	1.000000	0.029841	0.116777	0.224835	0.121788	-0.589665
PROCESSORS	0.470770	0.571964	0.029841	1.000000	-0.236931	0.032712	0.876820	-0.216030
BASE_CLOCK	-0.010677	-0.239839	0.116777	-0.236931	1.000000	0.691046	-0.355064	-0.114733
BOOST_CLOCK	0.223612	-0.101514	0.224835	0.032712	0.691046	1.000000	-0.118534	-0.488303
MEMORY_CONFIG	0.399088	0.688775	0.121788	0.876820	-0.355064	-0.118534	1.000000	-0.291269
INTERFACE_WIDTH	-0.175074	-0.258510	-0.589665	-0.216030	-0.114733	-0.488303	-0.291269	1.000000

Figure 8.33: Median from correlation matrix of data SNMPv3HMAC-SHA256-19288 hash.

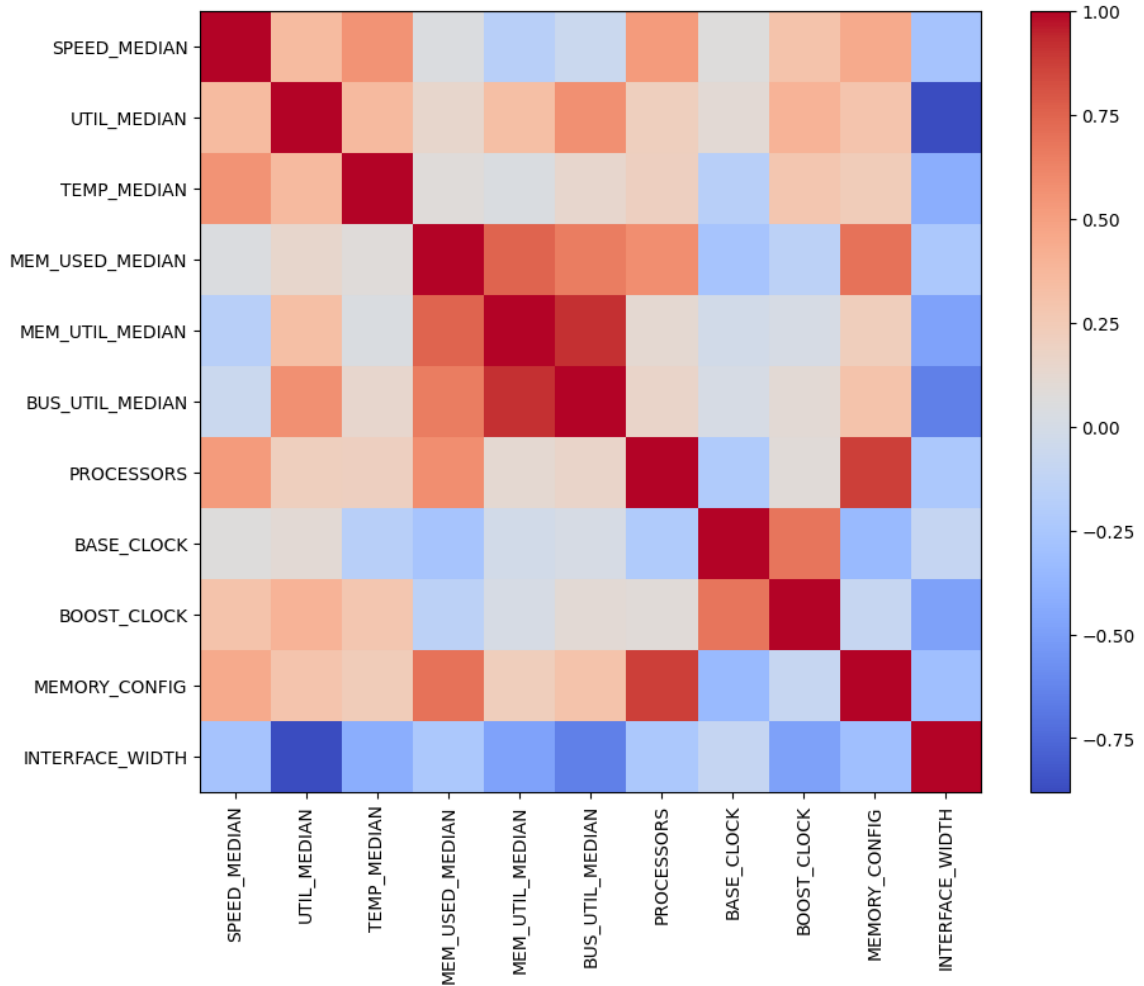|	SPEED_MEDIAN	MEM_USED_MEDIAN	MEM_UTIL_MEDIAN	PROCESSORS	BASE_CLOCK	BOOST_CLOCK	MEMORY_CONFIG	INTERFACE_WIDTH
SPEED_MEDIAN	1.000000	0.042539	-0.169589	0.516993	0.060636	0.301383	0.443348	-0.270039
MEM_USED_MEDIAN	0.042539	1.000000	0.745113	0.574594	-0.263735	-0.151485	0.695300	-0.239890
MEM_UTIL_MEDIAN	-0.169589	0.745113	1.000000	0.119096	-0.018218	0.009943	0.218067	-0.481753
PROCESSORS	0.516993	0.574594	0.119096	1.000000	-0.212741	0.084018	0.874911	-0.235261
BASE_CLOCK	0.060636	-0.263735	-0.018218	-0.212741	1.000000	0.677991	-0.340305	-0.100640
BOOST_CLOCK	0.301383	-0.151485	0.009943	0.084018	0.677991	1.000000	-0.087346	-0.484581
MEMORY_CONFIG	0.443348	0.695300	0.218067	0.874911	-0.340305	-0.087346	1.000000	-0.306216
INTERFACE_WIDTH	-0.270039	-0.239890	-0.481753	-0.235261	-0.100640	-0.484581	-0.306216	1.000000

Figure 8.34: Median from correlation matrix of data SNMPv3HMAC-SHA384-2568 hash.

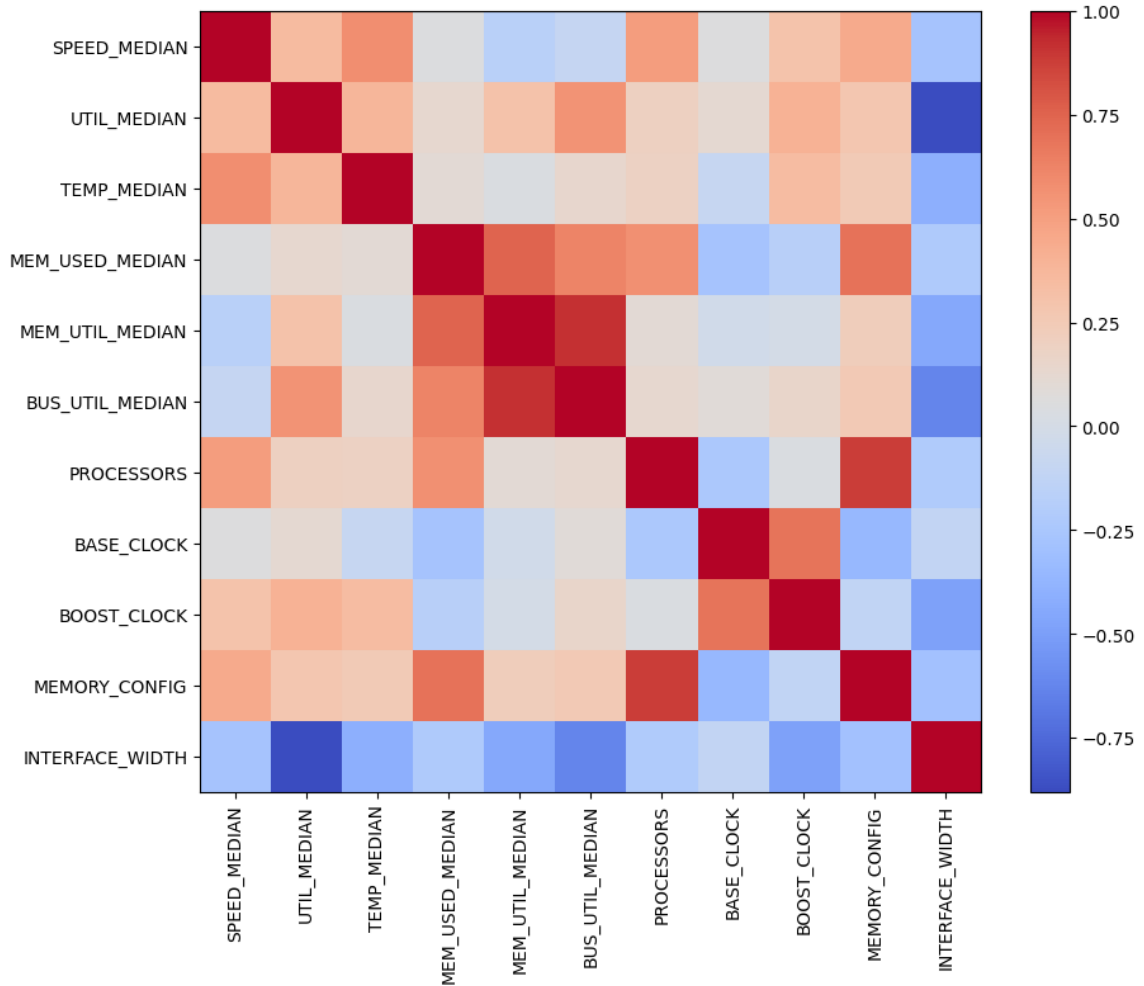|	SPEED_MEDIAN	MEM_USED_MEDIAN	MEM_UTIL_MEDIAN	PROCESSORS	BASE_CLOCK	BOOST_CLOCK	MEMORY_CONFIG	INTERFACE_WIDTH
SPEED_MEDIAN	1.000000	0.049220	-0.161365	0.510979	0.059773	0.296302	0.445413	-0.272946
MEM_USED_MEDIAN	0.049220	1.000000	0.747465	0.567237	-0.276501	-0.174577	0.696726	-0.223656
MEM_UTIL_MEDIAN	-0.161365	0.747465	1.000000	0.097963	-0.025046	-0.000115	0.221950	-0.449121
PROCESSORS	0.510979	0.567237	0.097963	1.000000	-0.236931	0.032712	0.876820	-0.216030
BASE_CLOCK	0.059773	-0.276501	-0.025046	-0.236931	1.000000	0.691046	-0.355064	-0.114733
BOOST_CLOCK	0.296302	-0.174577	-0.000115	0.032712	0.691046	1.000000	-0.118534	-0.488303
MEMORY_CONFIG	0.445413	0.696726	0.221950	0.876820	-0.355064	-0.118534	1.000000	-0.291269
INTERFACE_WIDTH	-0.272946	-0.223656	-0.449121	-0.216030	-0.114733	-0.488303	-0.291269	1.000000

Figure 8.35: Median from correlation matrix of data SNMPv3HMAC-SHA512-3848 hash.

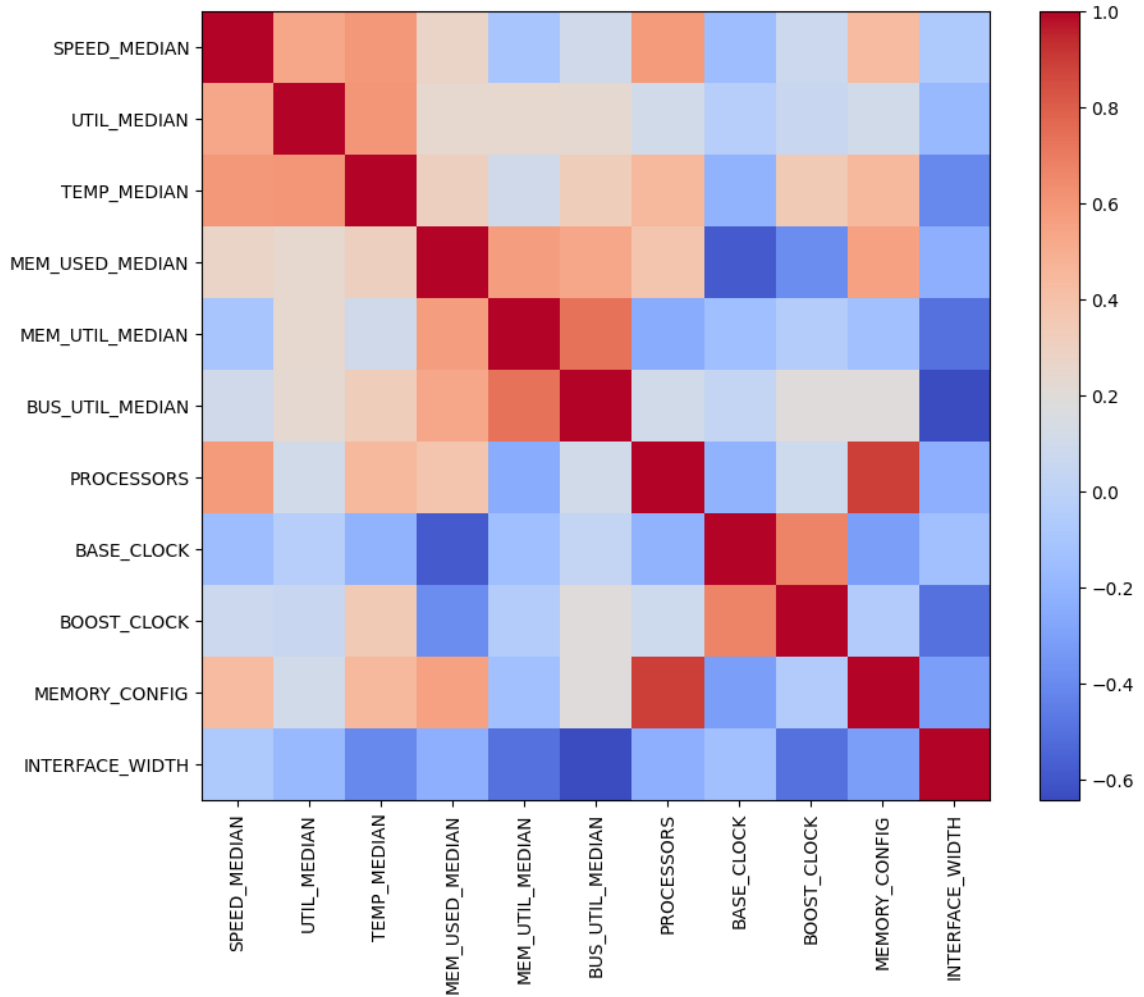|	SPEED_MEDIAN	MEM_USED_MEDIAN	MEM_UTIL_MEDIAN	PROCESSORS	BASE_CLOCK	BOOST_CLOCK	MEMORY_CONFIG	INTERFACE_WIDTH
SPEED_MEDIAN	1.000000	0.272691	-0.103224	0.579355	-0.153300	0.078270	0.433874	-0.073790
MEM_USED_MEDIAN	0.272691	1.000000	0.568667	0.375667	-0.583985	-0.388507	0.554253	-0.228324
MEM_UTIL_MEDIAN	-0.103224	0.568667	1.000000	-0.247050	-0.137774	-0.046504	-0.135291	-0.499162
PROCESSORS	0.579355	0.375667	-0.247050	1.000000	-0.212715	0.084434	0.888515	-0.228956
BASE_CLOCK	-0.153300	-0.583985	-0.137774	-0.212715	1.000000	0.671932	-0.315643	-0.130650
BOOST_CLOCK	0.078270	-0.388507	-0.046504	0.084434	0.671932	1.000000	-0.059014	-0.500121
MEMORY_CONFIG	0.433874	0.554253	-0.135291	0.888515	-0.315643	-0.059014	1.000000	-0.311812
INTERFACE_WIDTH	-0.073790	-0.228324	-0.499162	-0.228956	-0.130650	-0.500121	-0.311812	1.000000

Figure 8.36: Median from correlation matrix of data SolarWindsOrion hash.

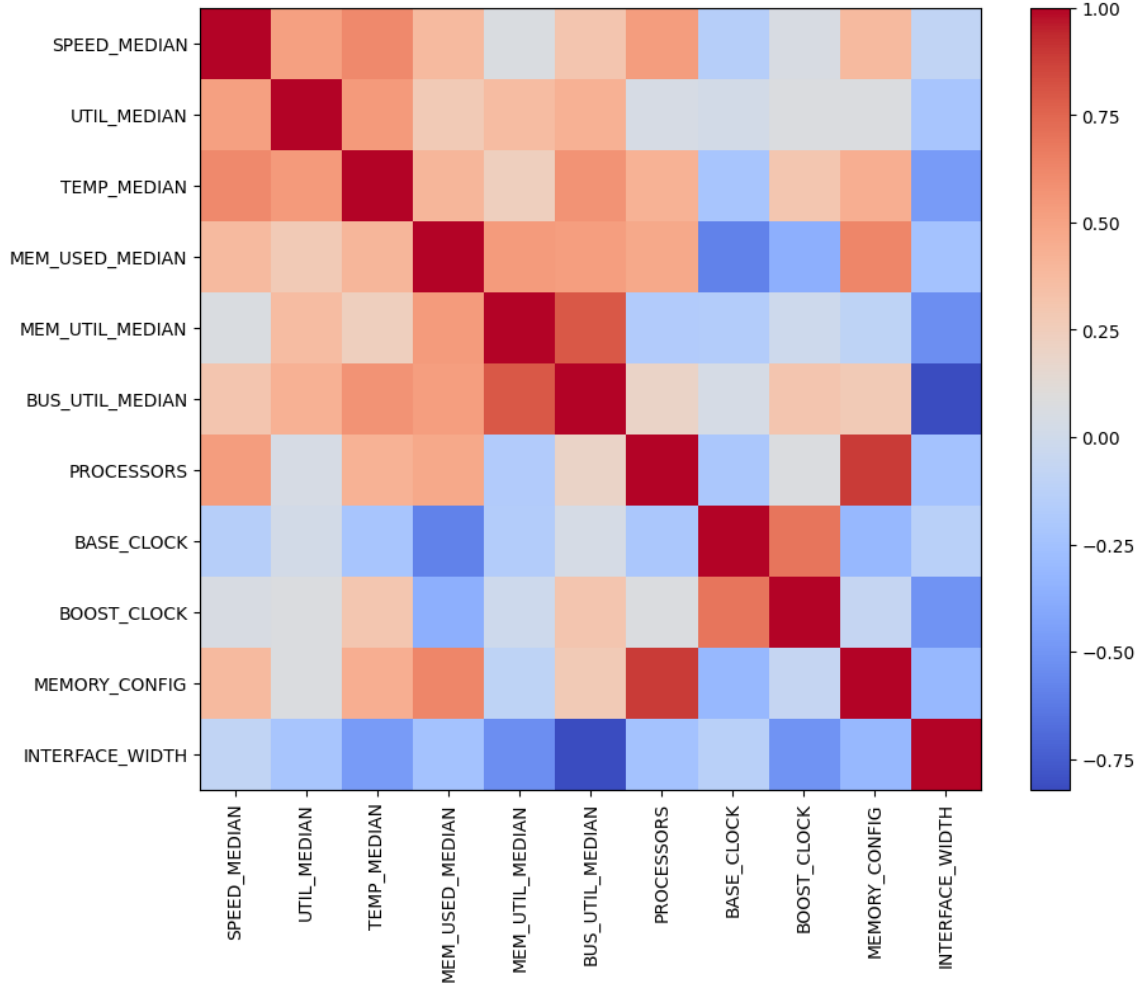|	SPEED_MEDIAN	MEM_USED_MEDIAN	MEM_UTIL_MEDIAN	PROCESSORS	BASE_CLOCK	BOOST_CLOCK	MEMORY_CONFIG	INTERFACE_WIDTH
SPEED_MEDIAN	1.000000	0.379085	0.061888	0.525317	-0.150306	0.046763	0.378414	-0.084236
MEM_USED_MEDIAN	0.379085	1.000000	0.533303	0.469343	-0.587656	-0.365617	0.623132	-0.241150
MEM_UTIL_MEDIAN	0.061888	0.533303	1.000000	-0.169203	-0.160764	-0.013398	-0.096849	-0.535057
PROCESSORS	0.525317	0.469343	-0.169203	1.000000	-0.204762	0.070629	0.891429	-0.239659
BASE_CLOCK	-0.150306	-0.587656	-0.160764	-0.204762	1.000000	0.688676	-0.313800	-0.125967
BOOST_CLOCK	0.046763	-0.365617	-0.013398	0.070629	0.688676	1.000000	-0.064026	-0.512607
MEMORY_CONFIG	0.378414	0.623132	-0.096849	0.891429	-0.313800	-0.064026	1.000000	-0.315245
INTERFACE_WIDTH	-0.084236	-0.241150	-0.535057	-0.239659	-0.125967	-0.512607	-0.315245	1.000000

Figure 8.37: Median from correlation matrix of data SolarWindsOrionv2 hash.

|	SPEED_MEDIAN	MEM_USED_MEDIAN	MEM_UTIL_MEDIAN	PROCESSORS	BASE_CLOCK	BOOST_CLOCK	MEMORY_CONFIG	INTERFACE_WIDTH
SPEED_MEDIAN	1.000000	0.157657	-0.236463	0.705561	-0.070874	-0.011854	0.539451	-0.074614
MEM_USED_MEDIAN	0.157657	1.000000	0.516639	0.233625	-0.189074	-0.233117	0.341947	-0.293678
MEM_UTIL_MEDIAN	-0.236463	0.516639	1.000000	-0.400181	0.371340	0.301223	-0.440770	-0.448182
PROCESSORS	0.705561	0.233625	-0.400181	1.000000	-0.237584	0.037744	0.889898	-0.210111
BASE_CLOCK	-0.070874	-0.189074	0.371340	-0.237584	1.000000	0.682730	-0.330307	-0.148215
BOOST_CLOCK	-0.011854	-0.233117	0.301223	0.037744	0.682730	1.000000	-0.087806	-0.505936
MEMORY_CONFIG	0.539451	0.341947	-0.440770	0.889898	-0.330307	-0.087806	1.000000	-0.298973
INTERFACE_WIDTH	-0.074614	-0.293678	-0.448182	-0.210111	-0.148215	-0.505936	-0.298973	1.000000
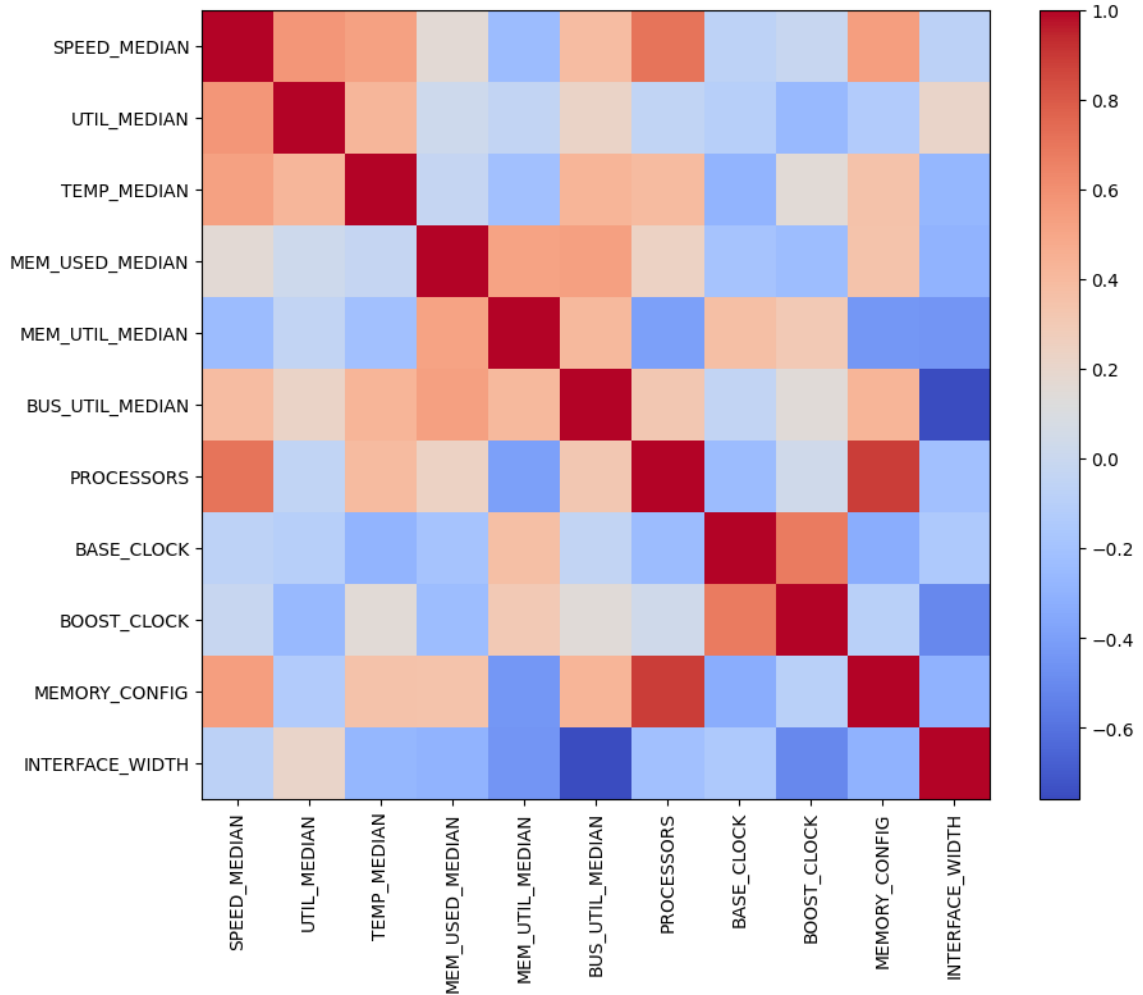
Figure 8.38: Median from correlation matrix of data TrueCrypt 5.0+ PBKDF2-HMAC-RIPEMD160 + Serpent-AES + boot (legacy) hash.

8.1.7 Sumary of the Analysis

Architecture is the most important aspect of GPU when discussing its performance, and it is no different in password cracking. The newer GPUs will outperform the old ones because their architecture is better. We analysed other aspects of GPUs to see what difference they make.

After reviewing the collected data, calculating the correlation matrixes and analysing them, the conclusion is that other than the architecture, the number of processors is the most critical hardware parameter for password-cracking performance overall. The main

reason is that the user usually uses a large keyspace when cracking passwords, which means he needs to calculate a hash for every single one. The faster the hashes are calculated, the faster we can go through a keyspace and find the correct password. The only exception, when the number of processors would not be significant, would be when the keyspace is tiny, with just a few hashes. Then the speed at which the cores run and how powerful these cores are would be more critical. For such a use case, using a CPU with few cores that are much more powerful than GPU cores would be the better option.

The second most crucial hardware characteristic is the boost clock speed of the GPU. The clock speed is the speed at which the processors run, directly affecting the password-cracking performance. The faster the processors run, the faster the hash is calculated. Thus we can try new passwords from the keyspace more often. The data we went through directly support this claim, where in almost every scenario, the higher the clock speed, the better the speed. The boost clock speed is much more important than the base clock speed because when GPUs are under much load, they go into the boost speed. Cracking passwords creates much load for GPUs, so their processors almost always run on the boost clock speeds. So the base clock speed is not that important.

The third most critical hardware characteristic is memory size, even though it was not used to its full potential in our experiments most of the time. However, this is the case only because, in the tests, we only used a few complicated password candidates. In reality, the hash configuration is much more complicated if we want to crack the password. It uses many more complicated rules, which may use multiple dictionaries and complex rules. Using such configurations would cause the memory to clutter, and it would be used to the fullest. We saw this when testing the hashes that are complicated to calculate. Even if we used simple configurations, it still used all the memory.

Memory bus size does not directly affect password-cracking performance that much. As we saw in the data, the GPUs with bigger memory buses were not performing better. Of course, having the bigger memory bus size is not an issue, and if having to choose between a smaller and a bigger memory bus, and all the other parameters are the same, choose the bigger one. However, choosing between a bigger memory bus or a higher number of processors, higher boost clock speed and bigger memory size, it is better to choose the latter. It is only better to choose the bigger bus size if we compare it to the better base clock speed, at which the GPU cores will rarely run.

Chapter 9

Conclusion

The goal of the thesis was to test the password-cracking performance of available graphic cards, analyse the data collected from these tests and deduce which hardware parameters of graphic cards affect their password-cracking performance the most.

Firstly, we studied password-cracking techniques and examined the different approaches to password-cracking. Then we surveyed currently available graphic cards. In this survey, we could see the broad spectrum of available GPUs with varying hardware parameters. In the third part, we studied the hashcat password cracking tool and familiarised ourselves with the different attack modes the hashcat provided.

We designed and implemented a tool for testing password-cracking performance from these findings. This tool controls the hashcat and launches it with different configurations while simultaneously collecting the data hashcat provides alongside the other hardware GPU data.

We tested each GPU with different configurations to simulate different password-cracking tactics. We used dictionary, force, hybrid and combinator attack modes in these tests. The dictionary attack had further configurations where we used small and big dictionaries alongside 10 and 100 rules. From these tests, we collected data and used them for the analysis. Then we developed scripts to help us visualise and analyse collected data.

After analysing the data, we have figured that the most important hardware specifications, other than the architecture of the GPU, are the processor count, the clock speed at which the processors are running and the size of GPU RAM.

Bibliography

[1] ADVANCED MICRO DEVICES, I. *AMD graphics cards information*
   [https://www.amd.com/en/graphics/]. Accessed: 30.12.2022.

[2] ALFRED, M., SCOTT, V. et al. *Handbook of applied cryptography.* CRC press
   ISBN-13:978-0-84-938523-0, 1997.

[3] BISHOP, M. and V. KLEIN, D. Improving system security via proactive password
   checking. *Computers Security.* 1995, vol. 14, no. 3, p. 233–249. DOI:
   https://doi.org/10.1016/0167-4048(95)00003-Q. ISSN 0167-4048. Available at:
   https://www.sciencedirect.com/science/article/pii/016740489500003Q.

[4] BLOCKI, J., HARSHA, B. and ZHOU, S. On the Economics of Offline Password
   Cracking. In: *2018 IEEE Symposium on Security and Privacy (SP).* 2018,
   p. 853–871. DOI: 10.1109/SP.2018.00009.

[5] CORPORATION, I. *Intel graphics cards information*
   [https://www.intel.com/content/www/us/en/products/docs/arc-discrete-
   graphics/a-series/desktop.html]. Accessed: 30.12.2022.

[6] CORPORATION, N. *NVIDIA graphics cards information*
   [https://www.nvidia.com/en-eu/geforce/graphics-cards/]. Accessed: 30.12.2022.

[7] ESTÉBANEZ, C., SAEZ, Y., RECIO, G. and ISASI, P. Performance of the most
   common non-cryptographic hashfunctions. *Software: Practice and Experience.*
   vol. 44, no. 6, p. 681–698. DOI: https://doi.org/10.1002/spe.2179. Available at:
   https://onlinelibrary.wiley.com/doi/abs/10.1002/spe.2179.

[8] GAUTAM, T. and JAIN, A. Analysis of brute force attack using TG — Dataset.
   In: *2015 SAI Intelligent Systems Conference (IntelliSys).* 2015, p. 984–988. DOI:
   10.1109/IntelliSys.2015.7361263.

[9] HRANICKÝ RADEK. *Digital Forensics: The Acceleration of Password Cracking.* Brno,
   2021. PhD thesis. Brno University of Technology, Faculty of Information Technology.
   Supervisor DOC. ING. ONDŘEJ RYŠAVÝ, PH.D.

[10] KARIMI, K., DICKSON, N. G. and HAMZE, F. A Performance Comparison of CUDA
   and OpenCL. *ArXiv.* 2010, abs/1005.2581.

[11] LI, Y., WANG, H. and SUN, K. A study of personal information in human-chosen
   passwords and its security implications. In: *IEEE INFOCOM 2016 - The 35th
   Annual IEEE International Conference on Computer Communications.* 2016, p. 1–9.
   DOI: 10.1109/INFOCOM.2016.7524583.

[12] Marks, M. and Niewiadomska Szynkiewicz, E. Hybrid CPU/GPU Platform For High Performance Computing. In: *ECMS*. 2014, p. 508–514.

[13] Munshi, A. The OpenCL specification. In: *2009 IEEE Hot Chips 21 Symposium (HCS)*. 2009. DOI: 10.1109/HOTCHIPS.2009.7478342.

[14] Murakami, T., Kasahara, R. and Saito, T. An implementation and its evaluation of password cracking tool parallelized on GPGPU. In: *2010 10th International Symposium on Communications and Information Technologies*. 2010, p. 534–538. DOI: 10.1109/ISCIT.2010.5665047.

[15] Scarfone, K. and Souppaya, M. *Guide to enterprise password management* [National Institute of Standards and Technology]. 2009. NIST Special Publication (SP) 800-118.

[16] Sprengers, M. *GPU-based Password Cracking: On the Security of Password Hashing Schemes regarding Advances in Graphics Processing Units*. 2011. Master's thesis. Radboud University Nijmegen, Faculty of Science, Kerckhoffs Institute.

[17] Tasevski, P. Password attacks and generation strategies. *Tartu University: Faculty of Mathematics and Computer Sciences*. 2011.

[18] Weir, M., Aggarwal, S., Medeiros, B. d. and Glodek, B. Password Cracking Using Probabilistic Context-Free Grammars. In: *2009 30th IEEE Symposium on Security and Privacy*. 2009, p. 391–405. DOI: 10.1109/SP.2009.8.

Appendix A

The contents of the attached storage medium

The attached SD card contains the following files:

- `dictionaries/` - dictionaries used in password-cracking tests,
- `dictionaries-dict/` - other dictionaries used in password-cracking tests,
- `hash-files/` - files containing hashes used in password-cracking tests,
- `input/` - configuration files used in password-cracking tests,
- `jupyter-notebook/` - jupyter notebook files used for analysis,
- `output/` - output data from tests
- `input-hash.csv` - file containing hashes used in password-cracking tests,
- `rules` - folder containing rule files,
- `src` - folder containing source files,
- `latex-files.zip` - compressed archive containing latex source documents,
- `thesis.pdf` - thesis report file.
- `README.md` - README manual for the project

The collected data for each GPU under output/GPUs are put into compressed archives because there were too many files. To use these data, you need to uncompress them.