BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF INFORMATION TECHNOLOGY
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF INFORMATION SYSTEMS
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

STRATEGIES FOR DISTRIBUTED PASSWORD CRACKING

STRATEGIE DISTRIBUOVANÉHO LÁMÁNÍ HESEL

MASTER'S THESIS
DIPLOMOVÁ PRÁCE

AUTHOR — **Bc. VOJTĚCH VEČEŘA**
AUTOR PRÁCE

SUPERVISOR — **Ing. VLADIMÍR VESELÝ, Ph.D.**
VEDOUCÍ PRÁCE

BRNO 2018

Vysoké učení technické v Brně
Fakulta informačních technologií

Ústav informačních systémů (UIFS) Akademický rok 2018/2019

Zadání diplomové práce

Student: **Večeřa Vojtěch, Bc.**
Program: Informační technologie Obor: Počítačové sítě a komunikace
Název: **Strategie distribuovaného lámání hesel**
Strategies for Distributed Password Cracking
Kategorie: Počítačové sítě
Zadání:

1. Seznamte se s nástroji Hashcat a John the Ripper pro obnovu hesel. Nastudujte si architekturu systémů BOINC a FITcrack.
2. Dle doporučení vedoucího proveďte výkonnostní měření různých typů grafických karet na referenční sestavě pro úlohu lámání hesel.
3. Prostudujte si existující doporučení a strategie k efektivnímu lámání pro různé formáty šifrovaných nosičů.
4. Navrhněte úpravy současného systému FITcrack, které zohlední zjištění z bodů zadání 2 a 3.
5. Navržené řešení implementujte jako rozšíření funkcionality FITcracku.
6. Výsledek otestujte a experimenty porovnejte implementované strategie s již dostupnými způsoby distribuce práce v systému FITcrack.

Literatura:

- D. P. Anderson, "BOINC: a system for public-resource computing and storage," Fifth IEEE/ACM International Workshop on Grid Computing, 2004, pp. 4-10. doi: 10.1109/GRID.2004.14.
- HRANICKÝ Radek, HOLKOVIČ Martin, MATOUŠEK Petr a RYŠAVÝ Ondřej. On Efficiency of Distributed Password Recovery. The Journal of Digital Forensics, Security and Law. 2016, roč. 11, č. 2, s. 79-96. ISSN 1558-7215.

Při obhajobě semestrální části projektu je požadováno:

- Body 1 až 3.

Podrobné závazné pokyny pro vypracování práce viz http://www.fit.vutbr.cz/info/szz/

Vedoucí práce: **Veselý Vladimír, Ing., Ph.D.**
Vedoucí ústavu: Kolář Dušan, doc. Dr. Ing.
Datum zadání: 1. listopadu 2018
Datum odevzdání: 22. května 2019
Datum schválení: 23. října 2018

Abstract
This thesis introduces viable password recovery tools and their categories as well as the technologies and hardware commonly used in this field of informatics. It follows by an overview of the available benchmarking tools for the given hardware. Thesis later contains a description of the custom benchmarking process targeting the aspects of interest. Later, the thesis moves to a distributed system FITcrack as it proposes and experimentally implements new features. The thesis finishes by comparison of the additions against the original state and highlights the areas of improvement.

Abstrakt
Tato práce představuje a kategorizuje nástroje pro obnovu hesel různých formátů. Dále se zabývá obecně používanými technologiemi a typy hardware využitelnými v tomto odvětví informatiky. Práce pokračuje přehledem nástrojů pro měření výkonu hardware, popisem navrhnutého procesu měření a cílů jednotlivých kroků. Později se přesouvá od samostatných uzlů k distrubovanému systému FITcrack. Práce navhuje rozšíření a úpravy onoho systému, které jsou experimentálně implementovány a nakonec srovnány s původní implementací.

Keywords
Hashcat, John the Ripper, BOINC, FITcrack, password recovery, distributed computation

Klíčová slova
Hashcat, John the Ripper, BOINC, FITcrack, obnova hesel, distribuované výpočty

Reference
VEČEŘA, Vojtěch. *Strategies for Distributed Password Cracking.* Brno, 2018. Master's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor Ing. Vladimír Veselý, Ph.D.

Rozšířený abstrakt

Tato diplomová práce se zabývá tématem obnovy hesel za použití grafických karet (GPU) jakožto hardware akcelerátorů. Nejprve práce představuje problematiku obnovy hesel, populární typy útoků a nástroje, které takové útoky umožňují, nehledě na to, zdali jsou komerční nebo volně dostupné. Avšak kvůli dostupnosti a možnosti analýzy, se tato práce do hloubky zabývá pouze volně dostupnými nástroji John the Ripper a hashcat, které práce vzájemně porovnává.

Práce se dále zaobírá distribuovanými nástroji, obzvláště nástrojem FITcrack, který je open-source a je také vyvíjen na Fakultě informačních technologií Vysokého učení technického v Brně. Nástroj FITcrack staví na systému BOINC, který je vyvíjen na Berkeley university v Kalifornii. BOINC je otevřený nástroj pro distribuované výpočty primárně určený pro využití zařízení dobrovolníků, kteří se připojují k různým projektům, na kterých se chtějí podílet.

Poté se práce vrací od distribuovaných systémů k samostatným výpočetním uzlům. Práce zde vysvětluje, jak takový uzel může vypadat, jaké může mít technické parametry a jaké lze využít hardware akcelerátory. Načež obratem opodstatňuje časté využití právě konfigurací s více grafickými kartami. Práce si dává za úkol pomoct čtenáři s výběrem správného modelu grafické karty, protože trh s grafickými kartami poskytuje značné množství různých modelů s různými parametry a typy chladičů. Při výběru modelu byly brány v potaz karty dostupné v období Q3/2018 až Q1/2019. Za tímto účelem představuje práce sadu testů zaměřujících se mimo měření výkonu i na stabilitu výpočetního uzlu, jeho průměrnou spotřebu, dosažené teploty, a i pořizovací a provozní náklady. Důležité hodnoty pak práce zobrazuje pomocí grafů a tabulek, jejichž obsah zároveň v textu komentuje a upozorňuje na zajímavé hodnoty.

Ve své další části se práce zaměřuje na metodologie spojené s obnovou hesel. Konkrétně na analýzu různých typů útoků a jejich zařazení do seznamu v takovém pořadím, ve kterém by měly být ideálně prováděny. Avšak také jsou brány v poraz různá další doporučení jak postupovat a zvýšit efektivitu útoků a tím snížit čas potřebný k nalezení hesla.

Následně práce poukazuje na nedostatky a chyby v implementaci některých útoků v rámci nástroje FITcrack. Konktrétně na problém útoků používajících více po sobě jdoucích masek, pro které systém vykazuje nedostatky při přechodu z masek s malým počtem kombinací na masky s velkým množstvím kombinací. Dále práce popisuje podobný problém týkající se útoků cílících na více solených hešů, pro které, jak práce vysvětluje, je nutné brát při plánování v potaz počet unikátních solí.

Zvýšení rychlosti distribuované obnovy hesel za použití slovníkového útoku je jedním z hlavních cílů této práce. Proto tato práce přichází se značnými změnami právě pro tento útok. Práce identifikuje dvě hlavní změny, které je třeba udělat. První je odstranění opakovaného fragmentování slovníků, které FITcrack provádí pro každý uzel a útok znovu, což zvyšuje jeho režii. Druhou změnou je změna metody zasílání fragmentu. FITcrack, respektive BOINC zasílají soubory přes protokol HTTP(S), zatímco tato práce navrhuje použití některého ze síťových souborových systému (Samba, BeegFS, Gluster, a další). Tato změna je možná, díky změně podoby systému, na který FITcrack cílil. Konkrétně přesun od využití obyčejných osobních počítačů z různých domén k vyhrazeným skupinách počítačů s vysokým výkonem v rámci jedné spravované domény.

Posledním rozšíření systému FITcrack, které tato práce navrhuje, je úprava a rozšíření algoritmu zodpovědného za přidělování práce uzlů. Toto rozšíření se skládá ze dvou částí.

Prvním je úprava jádra algoritmu tak, aby plánoval práci pouze pro aktivní uzly. Druhá úprava spočívá v přidání podpory různých priorit útoků.

Všechny změny a nedostatky byly implementovány, v práci popsány a následně i experimentálně otestovány proti původní implementaci. Z výsledků vyplývá, že všechny definované nedostatky byly správně identifikovány a implementované změny zlepšily použitelnost a výkon systému.

Z výsledků měření grafických karet vyplývá, že nejvýkonnější měřenou grafickou kartou je Nvidia RTX 2080Ti. Tato karta zároveň dosahuje nejlepší spotřeby elektrického proudu vůči poměru k výkonu. Nevýhodou této karty jsou však nejvyšší počáteční náklady a v případě menšího rozpočtu má velký potenciál karta Nvidia GTX 1070Ti. AMD RX 580 se ukazuje jako karta, které je sice výhodná při koupi, avšak náklady na její provoz jsou oproti konkurenci vyšší, a tedy časem začíná prodražovat.

Porovnání systému FITcrack před a po implementování změn ukazuje znatelné zrychlení slovníkových útoků až o několik desítek procent na rychlých typech hešů. Stejně tak jako úpravy plánování maskových útoků a útoků proti více solených hešů ukazují zlepšenou použitelnost.

Strategies for Distributed Password Cracking

Declaration

Hereby I declare that this master's thesis was prepared as an original author's work under the supervision of Mr. Ing. Vladimír Veselý, Ph.D. All the relevant information sources, which were used during preparation of this thesis, are properly cited and included in the list of references.

. . . .
Vojtěch Večeřa
May 22, 2019

Acknowledgements

Foremost, I would like to express my sincere gratitude to my supervisor Ing. Vladimír Veselý, Ph.D. as well as to Ing. Jan Pluskal as my research would be nearly impossible without his patient guidance, motivation, support, and feedback.

I am also inexpressibly grateful to all my colleagues from the FITcrack development team and to all students who have also participated in the system throughout the years.

Next, I would like to express my gratitude to Ing. Martin Holkovič for his feedback regarding the text corrections, formatting, and typography of this thesis.

Finally, I would thank all of my family (including people and animals) and my girlfriend for their patience, understanding, and neverending support.

"Mexican" tortilla wraps

- 6-8 tortilla wraps;
- 1/2 can of chopped and peeled tomatoes;
- 1 can of beans in tomato sauce;
- 1/2 can of sterilized Mexican vegetables;
- 400 g of pork meat;
- 2 medium-sized onions;
- 3 spoons of cooking oil;
- 1 spoon of olive oil; and
- seasoning: salt, black pepper, oregano, basil, chili.

At first, clean and cut the onions. Put cooking oil to a larger pan and fry the onions. Cut pork into noodles and add it to the pan when onion becomes soft and add a moderate amount of salt. Wait till the meat is done with occasional mixing. Then, add the tomatoes, beans, and another vegetable with salt, black pepper, oregano, basil, chili, olive oil and mix it. Follow by boiling it all together for a few minutes and till the sauce thickens.

Meanwhile, prepare (heat up) the wraps according to instructions on the cover. When the sauce thickens and wraps are warn, place a big spoon of sauce on every wrap, fold them, and serve. Optionally, add cheese, salad, or spring onion, based on personal preferences and taste.

Contents

1 Introduction **9**

2 Tools for password recovery **11**
2.1 Single machine tools . . . 12
2.1.1 John the Ripper . . . 12
2.1.2 Hashcat . . . 13
2.2 Distributed systems . . . 16

3 Hardware for password recovery **19**
3.1 Computation node configuration . . . 20
3.2 Graphics cards . . . 21
3.2.1 Nvidia . . . 22
3.2.2 AMD . . . 23
3.2.3 GPU summary . . . 23

4 Comparison of computation node configurations **24**
4.1 Designed benchmarks and tests . . . 24
4.1.1 Stability testing . . . 25
4.1.2 Usage of the operation memory . . . 25
4.1.3 Benchmarks of all algorithms . . . 26
4.1.4 Comparison of speeds of dictionary and mask attack . . . 26
4.2 Results of benchmarks and tests . . . 26
4.2.1 Stability of the systems . . . 27
4.2.2 Usage of operation memory . . . 28
4.2.3 Speed benchmarks of all algorithms . . . 28
4.2.4 Speed of dictionary and mask attacks . . . 29
4.2.5 Average power consumption and peak power drain . . . 29
4.2.6 Initial and operation costs . . . 30
4.2.7 GPU selection . . . 31

5 Recommendations and strategies for password recovery **32**
5.1 Recommendations . . . 32
5.2 Strategies . . . 33

6 Proposed system changes **35**
6.1 Salted hashes . . . 35
6.2 Transition between masks . . . 35
6.3 Inclusion of unrecovered hashes . . . 35

6.4 Distribution of dictionaries . . . 36
6.5 Task prioritization . . . 37
6.6 Task scheduling . . . 38

7 Implementation of proposed changes 40
7.1 Salted hashes and mask attack improvements . . . 40
7.2 Inclusion of unrecovered hashes . . . 41
7.3 Distribution of dictionaries . . . 42
7.4 Task prioritization . . . 44
7.5 Scheduling . . . 45

8 Comparison against original state 47
8.1 Mask attack improvements . . . 47
8.2 Handling of salted hashes . . . 48
8.3 Distribution of dictionaries . . . 48
8.4 Task prioritization . . . 49
8.5 Scheduling . . . 50

9 Conclusion 51

Bibliography 53

A Content of CD 57

B Comparison of dictionary and mask attack speeds 58
B.1 SHA-1 . . . 58
B.2 SHA-256 . . . 60
B.3 Bcrypt . . . 62
B.4 Scrypt . . . 64
B.5 RAR5 . . . 66

C Comparison of power consumptions 68

D Long-term cost effectiveness of node configurations 70

List of Figures

2.1 Overview of FITcrack system architecture as an example of extended BOINC architecture. 17

4.1 Comparisons of summarized average speeds each model reached for the five different hash-modes. 30
4.2 GPU selection decision chart visualizing node configuration relevance areas formed by price, power consumption, and performance coefficients. 31

8.1 Workunit duration generated by the original and adjusted system using attack containing three to seven symbols (`?a`) long masks. 47
8.2 Duration of initial workunits using original and adjusted system with various numbers of salted hashes. 48
8.3 Original and adjusted scheduling algorithm workunit duration throughout task duration comparison without considering communication overheads. . . . 50

B.1 Speed gain of mask attack compared to dictionary attack targeting *SHA1*. . 58
B.2 Average speeds of mask and dictionary attacks on eight GPUs targeting *SHA1*. 58
B.3 Average speeds of mask and dictionary attacks on single GPU targeting *SHA1*. 59
B.4 Detail of dictionary attack average speeds on single GPU targeting *SHA1*. . 59
B.5 Detail of mask attack average speeds on single GPU targeting *SHA1*. 59
B.6 Speed gain of mask attack compared to dictionary attack targeting *SHA256*. 60
B.7 Average speeds of mask and dictionary attacks on eight GPUs targeting *SHA256*. 60
B.8 Average speeds of mask and dictionary attacks on single GPU targeting *SHA256*. 61
B.9 Detail of dictionary attack average speeds on single GPU targeting *SHA256*. 61
B.10 Detail of mask attack average speeds on single GPU targeting *SHA256*. . . 61
B.11 Speed gain of mask attack compared to dictionary attack targeting *bcrypt*. . 62
B.12 Average speeds of mask and dictionary attacks on eight GPUs targeting *bcrypt*. 62
B.13 Average speeds of mask and dictionary attacks on single GPU targeting *bcrypt*. 63
B.14 Detail of dictionary attack average speeds on single GPU targeting *bcrypt*. . 63
B.15 Detail of mask attack average speeds on single GPU targeting *bcrypt*. 63
B.16 Speed gain of mask attack compared to dictionary attack targeting *scrypt*. . 64
B.17 Average speeds of mask and dictionary attacks on eight GPUs targeting *scrypt*. 64
B.18 Average speeds of mask and dictionary attacks on single GPU targeting *scrypt*. 65
B.19 Detail of dictionary attack average speeds on single GPU targeting *scrypt*. . 65
B.20 Detail of mask attack average speeds on single GPU targeting *scrypt*. 65
B.21 Speed gain of mask attack compared to dictionary attack targeting *RAR5*. . 66
B.22 Average speeds of mask and dictionary attacks on eight GPUs targeting *RAR5*. 66

B.23 Average speeds of mask and dictionary attacks on single GPU targeting *RAR5*. 67
B.24 Detail of dictionary attack average speeds on single GPU targeting *RAR5*. . 67
B.25 Detail of mask attack average speeds on single GPU targeting *RAR5*. . . . 67

C.1 Average power drain per one hour of measurements with effect of the length of measurement's duration. 68
C.2 Minimal power drain during measurements. 69
C.3 Maximal power drain during measurements. 69

D.1 Long-term cost effectiveness chart showing computed hashes per one euro of the total costs (purchase costs + power consumption costs) when targeting hash-mode *SHA1*. 70
D.2 Long-term cost effectiveness chart showing computed hashes per one euro of the total costs (purchase costs + power consumption costs) when targeting hash-mode *SHA256*. 70
D.3 Long-term cost effectiveness chart showing computed hashes per one euro of the total costs (purchase costs + power consumption costs) when targeting hash-mode *bcrypt*. 71
D.4 Long-term cost effectiveness chart showing computed hashes per one euro of the total costs (purchase costs + power consumption costs) when targeting hash-mode *scrypt*. 71
D.5 Long-term cost effectiveness chart showing computed hashes per one euro of the total costs (purchase costs + power consumption costs) when targeting hash-mode *RAR5*. 71

List of Tables

2.1 *Hashcat* workload profiles. 14
2.2 Overview of attacks implemented in *hashcat*. 14

3.1 Summary of GPU parameters stated by vendors and manufacturer. 23

4.1 Minimal, average and maximal temperatures of GPUs during two 20 minutes long attacks against MD5 and *bcrypt* hashes compared to maximal safe temperatures. 28
4.2 Usage of operation memory by *hashcat* process on CentOS with different number of GPUs. 28
4.3 Usage of operation memory by *hashcat* process on Windows with different number of GPUs. 29
4.4 The average performance gain/loss of GPU models across all hash-modes compared to non-optimized kernels on Nvidia GTX 1080Ti on CentOS. . . 29
4.5 The average performance gain/loss of GPU models across all hash-modes compared to non-optimized kernels on Nvidia GTX 1080Ti on Windows. . . 29

7.1 Benchmarks of the different implementations reading passwords from a file. 43

8.1 Initial workunit duration/size difference using single/multiple salted hashes comparing the original and adjusted implementations. 48
8.2 Task duration measurements comparing file transfer method using SHA-1 hash-mode. 49
8.3 Task duration measurements comparing file transfer method using Whirlpool hash-mode. 49

List of Acronyms

AMD Advanced Micro Devices, Inc..

ARM Advanced RISC Machine.

BOINC Berkeley Open Infrastructure for Network Computing.

CFM Cubic Feet per Minute.

CIFS Common Internet File System.

CLI Command Line Interface.

CPU Central Processing Unit.

CR Carriage Return.

CUDA Compute Unified Device Architecture.

DDR4 Double Data Rate 4.

FLOPS Floating-Point Operations Per Second.

FPGA Field-Programmable Gate Array.

GPU Graphics Processing Unit.

GUI Graphical User Interface.

HPC High-Performance Computing.

HTTP Hypertext Transfer Protocol.

HTTPS Hypertext Transfer Protocol Secure.

I/O Input/Output.

JSON JavaScript Object Notation.

JtR John the Ripper.

LAN Local Area Network.

LEA Law Enforcement Agency.

LF Line Feed.

MD5 Message Digest 5 Algorithm.

MIPS Millions of Instructions Per Second.

MPI Message Passing Interface.

NAS Network Attached Storage.

NFS Network File System.

NVMe Non-Volatile Memory Express.

OpenCL Open Computing Language.

OpenMP Open Multi-Processing.

OS Operating System.

PCFG Probabilistic Context-Free Grammar.

PCI-e Peripheral Component Interconnect Express.

POSIX Portable Operating System Interface for Unix.

PSU Power Supply Unit.

RAM Random Access Memory.

RAR5 Roshal Archive, version 5.

RPC Remote Procedure Call.

RPM Rotations Per Minute.

SAN Storage Area Network.

SATA Serial ATA.

SHA-256 Secure Hashing Algorithm, 256-Bits.

SHA-1 Secure Hashing Algorithm 1.

SMI System Management Interface.

SSD Solid-State Drive.

STL Standard Template Library.

USB Universal Serial Bus.

VRAM Video Random Access Memory.

WebUI Web User Interface.

WHQL Windows Hardware Quality Labs.

List of Terms

Markov model A statistical model defining a probability distribution over a sequence of symbols. Commonly used in the speech recognition system. Password recovery uses it as enhancement of brute-force attack as it generates the most probable password candidates first and may also reduce the keyspace [31].

password recovery Process of retrieving plain text passwords from hashes. Usable when a user loses the password to important data, for security audits, or for gaining access to data of interest. Legally done by *Law Enforcement Agency (LEA)* or illegally by attackers.

rainbow attack A type of attack which uses a pre-computed rainbow table to recover passwords by reversing the hashing functions [27].

scalability It is a system's ability to handle the addition of users and resources without suffering a noticeable loss of performance or increase of in administrative complexity [33].

workunit It is a unit of work generated from password recovery task prepared for a single host, which is supposed to last for a time specified by the user.

Chapter 1

Introduction

Every day of our lives, we are surrounded by passwords in various forms. Passwords are the most common way of authentication. We use passwords to unlock the phones, to confirm with the credit card transactions, to log-in to the websites on the Internet, and many other. It has been proven to be insufficient to save passwords in their plain text form. Therefore, the systems use a set of functions (algorithms) to derive value from each password.

The functions (so-called hashing functions) are one-way mathematical functions used in cryptography. The hashing functions produce fixed-length values (hashes) for any given input [35]. The system derives a hash out of the input password and compares it to the user's hash derived on the sign-up to the system. This approach is used to prevent anyone from accessing the original value of the password [37].

The derived hash or data (required by the authentication process for verification of the password) are the inputs of the *password recovery* process. The recovery process itself consists of password candidate generation and the set of functions deriving a hash or any other verification value. It means that the recovery process repeatedly performs the same steps over and over again, and the only variable input is the password candidate. Thus, the process is easily parallelizable, which speeds up the whole process. Employed algorithm and the performance of the computation node affect the speed of the recovery. The speed also highly depends on the used tool.

There are several popular attack types which various tools implement different ways. This thesis considers only the brute-force and dictionary attack types [37, 47]. Other attacks generally combine, modify, or enhance these basic attacks (e.g., attacks proposed by Chou et al. [3] and by Narayanan et al. [31]). The keyspace of attacks has tendencies to grow exponentially especially for brute-force attacks, attacks against multiple salted hashes, or when performing a series of multiple attacks. Therefore, one compute node does not have enough power to search through whole keyspace in an acceptable time. Hence, the motivation to use multiple interconnected nodes and distribute this task over the infrastructure.

The task distribution systems simplify the node control, adding new tasks, task assignment to nodes, and gathering of the results. Such systems also implement various scheduling algorithms maximizing the *password recovery* efficiency, providing task prioritization, dynamic speed adjustments, and others. The more advanced system may add separated user work-spaces and roles.

Chapter 2 overviews the viable tools for *password recovery*. Chapter 3 introduces types of accelerators usable for *password recovery* tasks and discusses the selection of accelerators for experimental testing. Each of the basic attacks generally has different computation char-

acteristics, resource utilization and is limited by different parts of the hardware. Chapter 4 proposes tests to identify the effects of different attacks on hardware as well as speeds of different hashing functions (algorithms). The chapter also outlines the comparison of various measured data on the selected accelerator.

Chapter 5 introduces common recommendations regarding *password recovery* tasks. The same chapter then follows with attack types order recommendations. Chapter 6 then identifies missing features, points several issues regarding task scheduling, and at last presents changes required for integration of prioritized task scheduling. Chapter 7 describes the implementation details and impacts of the proposed changes throughout the systems code base. Chapter 8 then compares the adjusted version of the system to the original with a series of measurements and observations. The final chapter (Chapter 9) summarizes and comments on the information from previous chapters and proposes future tasks and system enhancements.

Chapter 2

Tools for password recovery

Systems, applications, and even data containers such as archives, disk partitions, generally use different steps and functions to authenticate owners and decrypt the content, so-called authentication method. Therefore, this leads to a vast list of such methods and their variants. Authors of password recovery tools can then decide whether they want to target only some specific set of authentication methods or whether they want to make a tool targeting as many as possible processes.

The authors have to also choose which attacks type they want to support in their tool. Some tools are limited to simple brute-force and dictionary attacks. More advanced tools add the *rainbow attack*, brute-force attack enhanced by *Markov model* or dictionary attack enhanced by rules (so-called rule-based attack), or any other combinations of above-mentioned types [11, 36, 44].

Apart from the implemented attack types and supported authentication methods i.e., hashing algorithms, the tools can be categories based on the following criteria:

1. licensing of their use (commercial vs. free);
2. type of the computation unit (e.g., *Central Processing Unit (CPU)*, *Graphics Processing Unit (GPU)*, *Field-Programmable Gate Array (FPGA)*, or other less common types);
3. whether they support multi-node or just single-node computation;
    a) without the possibility to add node during computation;
    b) with the possibility to add new node during computation;
4. the type of the user interface (*Command Line Interface (CLI)*, *Graphical User Interface (GUI)* or *Web User Interface (WebUI)*);
5. control mode of the tool, which involves the following approaches:
    a) a local control system (interface module and the *password recovery* module are part of one application);
    b) a remote control system (interface module and the *password recovery* module are two standalone applications);
6. keyspace dividing mechanism;
7. support for skipping and limiting portion keyspace to compute;

8. how the tool obtains the data of interest;

    a) itself by extraction from the operation system, communication device, and other; and

    b) by a user who saves specially formatted data to a specific file.

In the context of this thesis, items 1, 2, and 7 are the main criteria. The selected tools based on the criteria are *hashcat* [44] and *John the Ripper (JtR)* [11]. Sections 2.1.1 and 2.1.2 further describe the tools, attacks they provide, and uncover some of their implementation details. Also, the systems mentioned in Section 2.2 take advantage of *hashcat* and could potentially use *JtR* as well. Therefore, Sections 2.1 and 2.2 separate the single machine tools for the distributed systems, which mostly build on top of them.

2.1 Single machine tools

Tools in this category primarily focus on the utilization of all available resources in one machine. That means that the tools often try to exploit various accelerators and other high-performance hardware, such as *GPUs*. The best tools in general use programming languages embracing parallel processing, *Open Computing Language (OpenCL)* and *Compute Unified Device Architecture (CUDA)* in most of the cases. While *CUDA* is language specific for programming Nvidia *GPU* only, the *OpenCL* is language widely supported by various vendors of *CPU*, *GPU*, *Advanced RISC Machine (ARM)*, and *FPGA* chips [24].

Both JtR[1] and *hashcat*[2] are open-source licensed with free to use and to redistribute (in source and binary forms) terms at the moment. Also, both tools support a large number of hash-modes and provide some mechanism for skipping and limiting the keyspace. Therefore, they are easy to use and control.

2.1.1 John the Ripper

JtR is a popular free open source password recovery tool. It comes in three major versions, the *official* version, the community enhanced version (*jumbo*), and the **Pro** version. The version 1.8.0-jumbo supports 194[3] hash formats. It is a well known and often recommended tool among penetration testers and cryptoanalysts [9]. Also, academic researches use this tool for their experiments; e.g., Lim et al. [30].

The official version of *JtR* supports only *Open Multi-Processing (OpenMP)* parallelization. However, the jumbo version contains some *OpenCL* and *CUDA* implementations for additional support of *GPUs*. Although, just for a limited amount of formats [11].

As for indexing and skipping of the portions of the keyspace, *JtR* uses chunk-based approach. Meaning user sets desired chunks of the keyspace and the tool then computes the number of password candidates accordingly. Thus, the individual nodes can be assigned sequences of the chunks of various magnitudes according to their power. The parameter assigned for this feature is `--nodes=MIN(-MAX)/TOTAL`, where MIN-MAX represents start and end numbers of a chunk of a wanted sequence. The only exception to this approach is for the Markov mode contained in the jumbo version. This mode offers a configuration of the start and end indexes.

[1] https://openwall.info/wiki/john/licensing

[2] https://github.com/hashcat/hashcat/blob/master/docs/license.txt

[3] Based on formats reported by `./john.exe -h`

Wordlist mode

Wordlist — also called dictionary — mode is the most straightforward mode present in the *JtR*. The only requirement is to supply the wordlist via parameters [12]. The supplied dictionary can be modified/extended via rules. It allows conversion of word to new candidates. Common modification rules are[4]:

- substitution of letters by numbers and special symbols – `hello → h3!!0` or `password → p4$$w0rd`;
- capitalization of letters words – `hello → Hello` or `password → paSsWorD`; and
- prepending and appending of numbers and special symbols – `hello → 1hello` or `password → password#`.

Incremental mode

Due to its keyspace, the incremental mode is the most powerful one. By default, it runs from the length of 1 to as many as possible characters with the possibility to define the starting and final lengths of the passwords. This mode takes advantage of the statistics of character occurrences, as it uses trigrams. Anyone can create such statistics from any text, e.g., already recovered passwords, books, or any other texts [12].

Without further configuration, *JtR* uses all 95 ASCII printable characters. However, it contains few pre-configured subsets of ASCII (lower-case letters, alpha-numerical character, and other) and supports custom character set. Those can be used to specify non-English alphabets and other non-ASCII symbols [12].

Single crack mode

The single crack mode is a wordlist mode which uses a small dictionary with a significant number of complex rules. This attack assumes that wordlist contains strings such as user credentials, full names as well as home directory names. Therefore, it is ideal to perform the attack on the computer of the person who created the hashes. This is the best attack type to use at the start of the *password recovery* process [12].

2.1.2 Hashcat

As its authors claim, it is the world's fastest password recovery tool and the first and only tool with the in-kernel rule engine. The tool is available under MIT license since April 2015. The change of the license makes the tool even more popular as it allows the integration of *hashcat* into some distributed systems (Section 2.2). Since the *hashcat*'s version 3.0, there is only a single tool supporting different types and vendors of chips via *OpenCL* kernels. Therefore, it is mandatory to have *OpenCL* installed on the system when running *hashcat* even for computations on *CPU*. It is known for its speed which is provided by these kernels. Since version 4.0, some hash-modes have two implementations of the kernels [44]:

- non-optimized kernels – new kernels supporting input password and salts up to the length of 256 bytes which results in lower speeds; and

[4] The list is not exhaustive nor complete.

- hand-optimized kernels – the only kernels present until version 4.0 with the support of passwords and salts lengths varying based on the used hash-mode.

Since version 3.0 *hashcat* contains the auto-tune engine allowing the user to limit the impact on the system responsiveness and computation efficiency on execution [44]. The engine is configurable either via parameter `--workload-profile` with values from Table 2.1, parameters `-n`, `-u` and `--opencl-vector-width`, or by adding/altering records in auto-tune database file (named `hashcat.hcstat`).

Profile	Performance	Runtime	Power Consumption	Desktop Impact
1	Low	2 ms	Low	Minimal
2	Default	12 ms	Economic	Noticeable
3	High	96 ms	High	Unresponsive
4	Nightmare	480 ms	Insane	Headless

Table 2.1: *Hashcat* workload profiles [44].

Hashcat implements five attacks, and three out of them are further modifiable by use of rules (see Table 2.2). Moreover, *hashcat* rules are designed based on *JtR* rules, which allows creating use-cases applicable for both rules without creating conflicts. All of the attacks are implemented to work on any *OpenCL* device theoretically. However, it should first undergo compatibility tests on exotic/untested chips. The dictionary and mask are the only two attacks used in the rest of the text. Therefore, Sections 2.1.2 and 2.1.2 describe them in more detail.

Id	Name	Input type	Rules
0	Dictionary	File(s) with passwords	Yes
1	Combinator	Two files with passwords	No
3	Mask	Mask as a string or in file	No
6	Dictionary + mask	File with passwords + mask as a string or in file	Yes
7	Mask + dictionary	Mask as a string or in file + file with passwords	Yes

Table 2.2: Overview of attacks implemented in *hashcat* [44].

Dictionary attack

A dictionary attack is — same as wordlist attack in *JtR* — one of the most straightforward attacks. The tool reads a given file line by line. Meaning a user has to obtain or generate some dictionary and supply it to the tool. Techniques for obtaining a dictionary may vary from taking a single book, gather some personal information about the person — who supplied the password — to unethical techniques like phishing and malware. Also, quite popular are dictionaries of the already leaked passwords like those from Rockyou, PhpBB, MySpace, and other sources[5].

When generating their own dictionaries, generators can take *Markov models*, *Probabilistic Context-Free Grammar (PCFG)* models, or use other techniques based on statistic/probabilistic models. Although, *hashcat*'s candidate generator can use *Markov model* directly (Section 2.1.2). Therefore, supplying the candidates in pre-computed form is unnecessary or even unwise.

[5] `https://wiki.skullsecurity.org/Passwords`

Hashcat can use rules with the dictionary attack in the same fashion as *JtR* does. Therefore, the examples from Section 2.1.1 generate the same password candidates for both tools.

The keyspace for a dictionary attack is just the number of usable password candidates saved in the supplied file. A portion of keyspace is skippable by the use of the parameter `-skip <N>` where the N is the number of candidates to skip. For limitation of the number of candidates to use (M), the tool contains parameter `--limit <M>`. Keyspace of the dictionary attack is also the same as the *hashcat*'s keyspace units.

Mask attack

The second attack type is taking advantage of password masks. It is the fastest attack type the tool provides as generates all of the password candidates right on the *GPU*. The mask specifies potential symbols for each position of the password candidates. The symbols can be specified either as a single symbol or as a specific set of symbols called charsets. Each set of symbols has its shortcut represented as a question mark followed by one character. Allowed characters are `a, b, d, h, H, l, u, s` and 1, 2, 3, 4. *Hashcat* comes with a few basic predefined charsets [44]:

- `?l` – lower-case letters (26 symbols): `a, b, c, d, ..., z`;
- `?u` – upper-case letters (26 symbols): `A, B, C, D, ..., Z`;
- `?d` – decimal symbols (10 symbols): `0, 1, ..., 9`;
- `?s` – special symbols (33 symbols): all printable symbols;
- `?a` – all ASCII printable symbols (95 symbols): `?l?u?d?s`;
- `?b` – all byte values (256 „symbols"): `0x00, 0x01, ..., 0xff`;
- `?h` – hexadecimal lower-case symbols (16 symbols): `a, b, ..., f, 0, 1, ...,9`; and
- `?H` – hexadecimal upper-case symbols (16 symbols): `A, B, ..., F, 0, 1, ...,9`.

Moreover, *hashcat* offers four user-defined charsets configurable by either:

(a) defining charsets as a part of *hashcat* command via values of parameters `-1`, `-2`, `-3`, `-4`;

(b) defining them in separated files passed via their names to the parameters `-1`, `-2`, `-3`, and `-4`; or

(c) defining up to a four charsets along with the mask in the so-called mask file (`.hcmask` extension), which *hashcat* then uses instead of the string with a mask.

Therefore, the mask corresponding to the string `hello` is `?l?l?l?l?l` and for string `1aPple` is `?d?l?u?l?l?l`. The same masks also generate string `brown` and `2pEars`. Computation of the mask keyspace is not as simple as for dictionary attacks. It generally involves magnitudes multiplication of all sets used in the mask. Thus, keyspace of a mask increases exponentially as the mask grows in length. Keyspace of the stated masks is then calculatable as follows:

- `?l?l?l?l?l` has got keyspace $26 * 26 * 26 * 26 * 26 \Rightarrow 11\,881\,376$; and
- `?d?l?u?l?l?l` $10 * 26 * 26 * 26 * 26 * 26 \Rightarrow 118\,813\,760$.

However, *hashcat* processes the masks to achieve the best possible recovery performance, which alters the resulting mask's keyspace. *Hashcat* splits masks into two parts called *base* and *mod* loops. The *base* loop is adjustable by a user, but the *mod* loop works as performance amplifier running directly from *GPU*. Both parts are dynamic and depend on the mask length and hash-mode. Therefore, it is not recommended to compute keyspace manually, but to run *hashcat* with `--keyspace` parameter instead. Because of this, the skipping and limiting of keyspace is a little bit trickier and more complex task. Despite that, the process is the same as for dictionary attack (parameters `--skip <N>` and `--limit <M>`) [44].

2.2 Distributed systems

The distributed system means a password recovery system operating on multiple individual computation nodes. The system shall not be restricted to operate only on nodes connected to *Local Area Network (LAN)*, and it shall support any combination of Internet/intranet deployment scenario.

Systems in this category come either as commercial or as free open-source. The representatives of the commercial tools are Hashstack[6], ElcomSoft Distributed Password Recovery[7], Passware Kit Forensic[8], and more. As for free to use open-source tools, the representatives are Hashtopolis (formerly Hashtopussy)[9], CrackLord[10], and FITcrack[11]. Also, there are several experimental implementations like Lim's *Message Passing Interface (MPI)* extension of *JtR* [30], and other.

While most of the commercial system — except Hashstack — use their proprietary cracker. Most of the open-source tools and Hashstack use either JtR, *hashcat* or both. Therefore, they are comparable regarding the maximal speeds. What differs from the system to system are the server side tools, communication protocols, and limitations. The main differences are generally in approaches to distribution of dictionary attack, computation schedulers (work generators), and even pre-processing of the data. Both of these aspects significantly affect the utilization of resources.

One of the essential attributes for evaluation of a distributed system — except for maximal achieved *password recovery* speed — is the *scalability* and the computation efficiency. As Hranický et al. show in [20], the *scalability* and computation efficiency of FITcrack system seems to be promising.

FITcrack

FITcrack is the distributed password recovery system developed at Brno University of Technology Faculty of Information Technology. The system originated from the single machine password recovery tool. FITcrack at first used its single machine tool as a cracker to be later

[6] https://sagitta.pw/software/
[7] https://www.elcomsoft.com/eprb.html
[8] https://www.passware.com/kit-forensic/
[9] https://github.com/s3inlc/hashtopolis/wiki
[10] http://jmmcatee.github.io/cracklord/
[11] https://fitcrack.fit.vutbr.cz/

replaced by *hashcat*. Transition to *hashcat* greatly extended base of the supported hash-modes as well as uplifted the recovery speeds, hardware compatibility, portability of the system, thus, significantly increasing the potential of FITcrack [21, 48].

FITcrack uses *Berkeley Open Infrastructure for Network Computing (BOINC)* (described in Section 2.2) as the underlying platform for distribution of work to nodes. The system consists of a dedicated server node and multiple computation nodes employing the client-server communication scheme. However, FITcrack slightly extends BOINC's architectures by adding new application-specific modules as shows Figure 2.1.

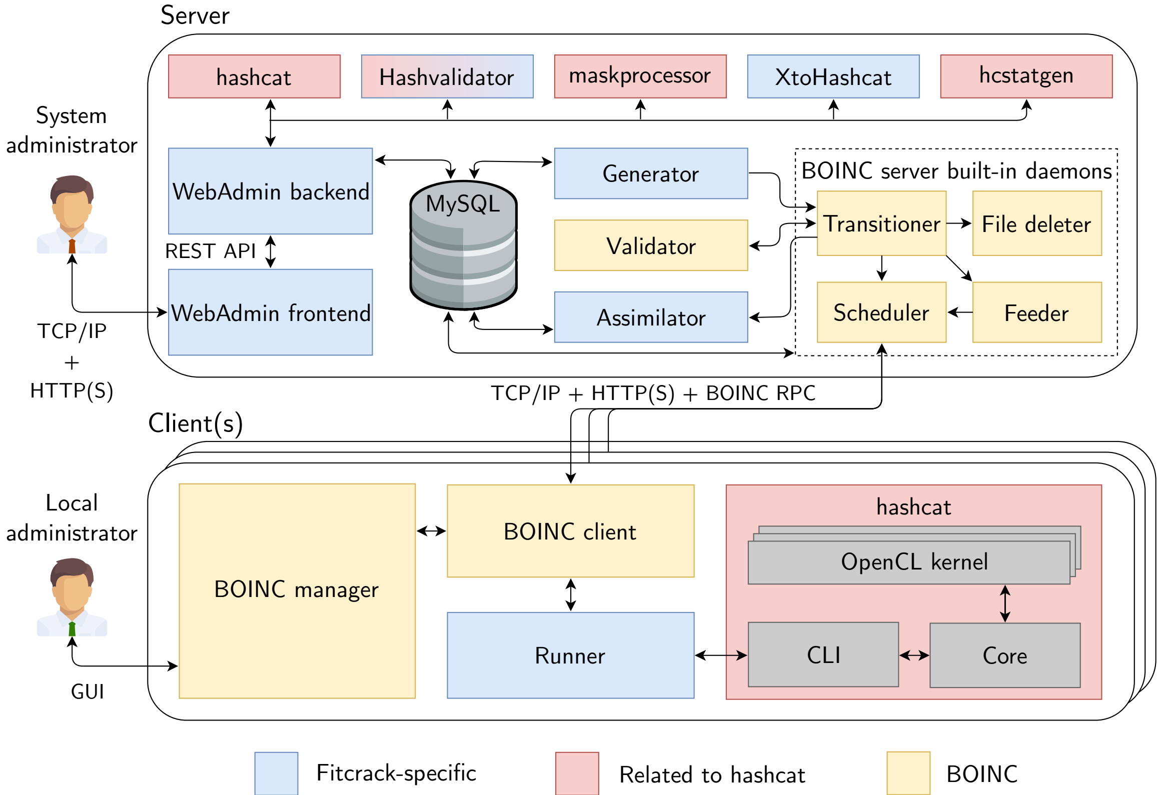

Figure 2.1: Overview of FITcrack system architecture as an example of extended BOINC architecture [15, 23].

The work generator uses the desired time interval explicitly specified for each task. For the work generator to achieve such precise sizes of units of work (*workunits*), FITcrack must tailor each *workunit* for each of the connected nodes based on its last know recovery speed. It also has to determine the initial speed of nodes. Therefore, it generates a special benchmark *workunits* for each node. Only after nodes report their speeds, the work generator can create actual password recovery units. When the node finishes the assigned unit, it submits the results to the server. The server then processes the results to determine whether the node recovered the password or not, updates the node's speed and generates new *workunit* as needed. This way, it adjusts sizes of any future *workunits* of the given task.

The system keeps track of both the real and the *hashcat*'s keyspaces at the same time and converts between them as needed. Therefore, it generates units consisting of a requested number of hashes to compute as opposed to the chunk-based approach used by *JtR*.

Berkeley Open Infrastructure for Network Computing

The *Berkeley Open Infrastructure for Network Computing (BOINC)* authors describe the system as follows:

> *BOINC*[12] is a platform for public-resource distributed computing. *BOINC* is being developed at U.C. Berkeley Spaces Sciences Laboratory by the group that developed and continues to operate SETI@home. *BOINC* is developed as open source [1].

The system consists of the *BOINC* server and *BOINC* client applications. The client application is a *CLI* application or service (daemon) depending on the host configuration. The server is a set of daemons and cron jobs using the database for their communication.

The clean installation of *BOINC* server comes with built-in daemons an sample implementations of project modules and the host application. The sample's purpose is to provide an easy solution for testing of the installation and to provide and example implementation of all system components. The implemented modules of the system are **generator**, **assimilator**, and the **host application** (Figure 2.1). The built-in daemons provide the core features like the creation of the user accounts, authentication of the users, client-server communication, temporary file cleanup, and many others. *BOINC* also provides a web interface which allows user administration, customization of user accounts, *workunit* and result monitoring, host overview, and various statistics regarding the application versions.

BOINC uses *Remote Procedure Call (RPC)* over *Hypertext Transfer Protocol (HTTP)* protocol. The *RPC* requests and responses use XML structures to store data. *BOINC* communicates and transfers the information as files. Thus, the *workunit*'s input files have to contain all of the computation configurations and data. Same as the client application has to save all of its outputs to the files as well. Also, *BOINC* directs the information from the client application's standard error output to a file which *BOINC* then also reports to the server.

Both host and server are implemented using the C programming language. Therefore, the provided libraries are best to be used by C written applications. Client-side libraries contain various functions for *workunit* status notifications, critical sections, *Input/Output (I/O)* wrappers, and many others. The server-side libraries provide interface for manipulation of the *workunits*, database connection and query execution, and similar.

Also, one of the vital features of *BOINC* is the number of supported *Operating Systems (OSs)*, processor architectures, and computation accelerators. The *BOINC* can detect *OpenCL*, *CUDA*, and *ARM* devices on the hosts and ship the appropriate versions of applications.

[12] http://boinc.berkeley.edu

Chapter 3

Hardware for password recovery

Password recovery is not an easy task. It can consume many computation resources. The tools from the previous chapter are highly viable for such a task. To get the most out of the single machine tools, one needs to use the components with the highest performance. Selection of the models and types of such components highly depends on the budget. However, the owners of such machines often overlook the component's power consumption as it is fairly minimal compared to the systems with multiple nodes.

The power consumption is an crucial aspect for the systems with multiple nodes. There is still some budget for the hardware at the start, but the system maintainer has to also consider the power consumption of each machine or maybe of each component inside it. Although there might be two approaches to building computation machines for such scenarios:

1. a maintainer can obtain the components with the most performance but with the higher purchase price and also higher power drain; and

2. a maintainer can obtain the components with the best power efficiency and purchase price at the cost of performance lost per each node.

Considering there is the same budget for purchasing the components, the first approach would lead to the building of fewer machines. The second approach would, because of the lower cost, lead to the possibility of building a few extra machines. There might be pros and cons to both approaches. However, one has to consider the available space for locating the machines, power grid limits, air-conditioning of the rooms and temperatures of the machines and its components, impact on the communication network and some other aspects. This thesis wants to verify whether both of the approaches are viable and what might be pros and cons, and how would nodes in such configuration stand against each other.

The most promising computation nodes designed for password recovery (and *High-Performance Computing (HPC)* in general) take advantage of the hardware accelerators such as *GPU*, *FPGA* [13], and others. Among those, the most viable, affordable, and usable for general-purpose computations in cluster environments is *GPU* [34]. Also, for some hash-modes requiring a more significant amount of memory may be faster to perform password recovery on standard *CPU* rather than on *GPU* [26].

This chapter is about reviewing currently available mid-range and high-end *GPUs*. Such cards are potent, considering their price, for password recovery. The *GPUs* provide excellent data parallelism because of their architecture, which aims at graphics tasks applying the same operation to a set of different pixels [7, 45].

3.1 Computation node configuration

This section describes the configuration of a computation node, which is a machine further used for the performance measurements of the later discussed *GPU* models. Therefore, the only changing component of the whole machines are the *GPUs*. Other components remain the same. The machine contains:

- *processor* (*CPU*) – Intel Core i7-6700 @ 3.40 GHz with the stock cooler;
- *memory* (*RAM*) – 2x 16GB KINGSTON 2400 MHz DDR4 CL16;
- *Solid-State Drive (SSD)* – Samsung 850 EVO 250 GB;
- *Power Supply Unit (PSU)* – 2x EVGA SuperNOVA 1600 G2, 80+ GOLD, 1600 W;
- *motherboard* – Asus ASRock H110 Pro BTC+;
- *GPU* connection – 8x *PCI-e* x1 to x16 Riser *USB* 3.0;
- *chassis* – custom-made U4 standard rack chassis (81.28 x 55.88 x 30.48 cm); and
- *chassis fans* – 3x Delta 190 CFM.

The chosen motherboard has a significant effect on the selection of the components. This motherboard offers eight *Peripheral Component Interconnect Express (PCI-e)* x1 ports but unfortunately supports only up to 32 GB of *Double Data Rate 4 (DDR4) Random Access Memory (RAM)* with 2166 - 2400MHz frequency. Design of this motherboard aims at crypto-currency mining machines which require very few *RAM*. Therefore, the mentioned *RAM* limit is not an issue in the mining field. However, with the use of *PCI-e* x1 to x16 risers, the board can handle up to eight *GPU* at the same time. This parameter is the main deciding factor for choosing this motherboard.

Power supply unit has to be able to power all of the components while not hitting its maximal output drain. Overloaded *PSUs* loses its efficiency and may output unstable power. Most of the power from the *PSU* goes to the *GPUs*. High-end *GPUs* generally use 250 to 350 watts when fully loaded. So to accommodate eight high-end *GPUs*, Intel *CPU*, disk, motherboard, and fans the *PSU* has to provide $8*350+65+2+50+3*18 = \sim 3000[W]$ when everything would be under the full load [2].

When choosing from the generally available *PSUs*, there is none with such output power. The commonly used solution to this problem is to use the two identical units providing enough output power. Power for the first four *GPUs* provides the first *PSUs* and the second *PSU* powers the other four *GPU* — only one of them then powers the motherboard with the *CPU* and *RAM*. There are few requirements on *CPU*:

- it has to provide at least eight *PCI-e* lanes for *GPU* as well as some other for *RAM*;
- it has to fit the socket provided on the motherboard (socket LGA 1151);
- it has to have enough computation power to handle the applications using the *GPUs* and operation system and its services;
- its power consumption has to be lower than 91W (motherboard limitation); and
- it does not have to be the most potent *CPU* out there.

Processor Intel Core i7-6700 meets all of the requirements and is also available for a decent price. It may be relevant to an architect when building a new machine and operating on a limited budget.

The chosen disk is one of the fastest available *SSDs* connected via *Serial ATA (SATA)* port. Unfortunately, the motherboard does not provide any M2.slot which could be equipped by much faster *Non-Volatile Memory Express (NVMe) SSD*. Therefore, the chosen disk is the Samsung 850 EVO.

Since all of the components should be placed in a single chassis, the temperature accumulation occurs. The high temperature then can lead to components overheating and potential failure. In order to prevent such issues, the chassis holds three industry grade fans as stated above. These fans have a maximal rotation speed of 4000 *Rotations Per Minute (RPM)* and provide airflow of 190.0 *Cubic Feet per Minute (CFM)* while maintaining static pressure of 174.4 pascals[1] [38]. Therefore, they provide enough airflow in the chassis to accommodate all of the components with enough new air taken out of the chassis.

3.2 Graphics cards

Graphics cards are essential parts of password recovery computation nodes. Considered are only the cards designed for playing video games for their performance/price ratio. As for their evaluation, the comparison then focuses on the performance, price, power consumption, and cooling solution (dimensions).

Most of the high-end gaming cards have a similarly designed card in the industrial category. Such cards aim at artificial intelligence, visualization, cloud computing, and others. The benefits of the industrial card are: (a) their cooling solutions; (b) port configurations, which are better suited for the use in rack chassis using multiple *GPU* configurations; and (c) they generally offer lower power consumption compared to the gaming cards.

GPUs come with various cooling solutions. There are cards with just passive cooling (not in the gaming category), blower fan cooling, open-air cooling (single/dual/triple fan) and some even with all-in-one solutions, combining water loop cooling with blower fan cooling into one cooler [14].

Most of the gaming *GPUs* come with open-air solutions, then some with blower fan solutions and minority with an all-in-one solution. None of the selected cards has an all-in-one cooling solution. The main reason is that such cards would require extra space for the radiators attached to them. These radiators and pipes attaching them to the platform on the card would take a considerable amount of extra space inside the chassis.

The blower fan solutions have only one intake fan. Blower fan pushes (blows) the air through the length of the whole card and exhausts through the front panel of the card. This solution generally aimed to be used in the chassis with the poor airflow since the fan blows the heat in a single direction. Apart from that, the advantage of this solution is its mostly constant size as it always fits the size of two slots [8, 18, 43].

However, the open-air solution blows the heat to all possible directions [10, 41]. The most significant disadvantage of the design is that it may take up to 2.5 slots depending on the manufacturer of the cooler and the power of the card. The chassis used of the referential computation node has only space of the eight times two slots. This limitation makes it impossible to use some of the triple fan solutions present on the most potent card models.

[1] https://www.digikey.com/product-detail/en/delta-electronics/FFB1212EHE-F00/603-1083-ND/1014414

3.2.1 Nvidia

Nvidia is the brand of the graphics cards holding the highest market share for the last few years among the add-in cards [29]. The benchmarks [42] show that fifteen of Nvidia's cards are among the top twenty cards. The benchmarks also show that all of the top ten cards are from Nvidia. Therefore, the six selected models of the Nvidia cards from the gaming category for benchmarking. Nvidia‘s computation cores are called *CUDA* cores.

Nvidia GeForce GTX 1050Ti 4G

The first and the cheapest and the least powerful benchmarked card is Gigabyte GeForce GTX 1050Ti D5 4G. This card has a low initial cost 4 899 CZK (189 EUR), low power consumption, and for the small dimensions of the single fan version. Nvidia, in its technical specification[2], states the maximal power consumption of only 75 watts. Combination of 748 *CUDA* cores, 4 GB memory, and other mentioned factors make it a viable candidate for testing whether such configuration would have better power efficiency than other cards.

Nvidia GeForce GTX 1060 6G

This card is the most popular card among gamers [5]. The chosen model is the ASUS Dual GeForce GTX 1060 O6G for its availability in stores since parameters of the models provided by all vendors are relatively similar. The card costs 7 399 CZK (287 EUR) and takes advantage of the dual fan solution with metal ribs parallel to the longer side of the card (similar to ribs in blower fan solution). It provides 6 GB of memory, 1 280 *CUDA* cores, and maximal power consumption is 120 watts.

Nvidia GeForce GTX 1070Ti

Three cards fit into performance gap between GTX 1060 6G and GTX 1080Ti, the GTX 1070, GTX 1070Ti, and GTX 1080. The performance of GTX 1070Ti is pretty similar to the performance of GTX 1080 and is generally cheaper. Also, the availability of GTX 1070Ti in stores is significantly better. The model of choice is MSI GeForce GTX 1070Ti ARMOR 8G. It has dual fan open-air cooling solution, 8 GB of memory, 2 432 *CUDA* cores, the maximal power consumption of 180 watts, and costs 11 790 CZK (456 EUR).

Nvidia GeForce GTX 1080Ti

It is the card with the highest computing power in the GeForce GTX series, and even all the other gaming cards until the release of the new cards in Q3 of 2018 (Nvidia GeForce RTX). The low availability of the cards results in benchmarking of Palit GeForce GTX 1080 Ti Founders Edition for 20 025 CZK (774 EUR) cards. The card offers 11 GB of memory, 3 584 *CUDA* cores, and has a maximal power consumption of 250 watts.

Nvidia GeForce RTX 2080Ti

This model is the best Nvidia offers in the gaming sector. The selected version is GIGABYTE GeForce RTX 2080Ti TURBO 11G as one of a few 2-slot versions available on the market. The card offers 11 GB of memory, 4 352 *CUDA* cores, maximal power consumption is 250 watts, and costs 33 567 CZK (1 298 EUR).

[2] https://www.nvidia.com/en-us/geforce/products/10series/geforce-gtx-1050/

3.2.2 AMD

Advanced Micro Devices, Inc. (AMD) is the other brand of the add-in *GPU*. They are not offering any solution with performance comparable to the Nvidia GTX 1080Ti and the new Nvidia RTX 20 series cards. Their market share is much smaller than of the Nvidia. The reasons are higher power consumption and generally lower performance. However, because of their lower purchase price and better efficiency for some specific tasks, their RX 480 and RX 580 cards were the first out of stock during crypto-currency mining fever during 2017 [16]. The computation cores in *AMD* cards are called stream processors.

Radeon RX 580 8GB

This is the only model out of the top AMD models available in local stores in Czechia on December 2018. Also, this model's probable computation efficiency is the main reason behind undergoing it to the benchmarks. Another reason is the similarity of its performance to Nvidia GTX 1060 6G (Section 3.2.1). The selected version of the card is XFX Radeon RX 580 GTS XXX Edition 8GB. It offers 8 GB of memory, 2 304 stream processors, the maximal power consumption of 185 watts, and costs 5 954 CZK (230 EUR).

3.2.3 GPU summary

Brand	Type	Cores	Memory	Cooling solution	Consumption [W]	Price [EUR]
Nvidia	GTX 1050Ti 4G	748	4 GB	Single fan	75	189
	GTX 1060 6G	1 280	6 GB	Dual fan	120	287
	GTX 1070Ti	2 432	8 GB	Dual fan	180	456
	GTX 1080Ti	3 584	11 GB	Blower	250	774
	RTX 2080Ti	4 352	11 GB	Blower	250	1 298
AMD	RX 580 8GB	2 304	8 GB	Dual fan	185	230

Table 3.1: Summary of GPU parameters stated by vendors and manufacturer [4].

The model selection is not an easy task because the majority of *GPUs* wears various dual/triple fan cooling solutions combined with heigh thin metal ribs. It is fairly common that the most potent model coolers require more than two PCI-e slots and therefore do not fit into the machine's chassis in required numbers.

It is hard to find *GPU* with a blower cooler in the gaming segment. The perfect examples are the Radeon RX Vega cards, which have overall bad availability and those available wear too wide coolers to fit the chasis. Therefore, this thesis does not include any data regarding AMD RX Vega cards. The struggle regarding the blower coolers affects other models as well. As Table 3.1 shows, only the top model are available with blower coolers. The table also summarizes the parameters and information about the selected models

Chapter 4

Comparison of computation node configurations

This chapter at first describes and discusses the configuration of the computation node and all of the selected *GPU*. Then, it presents a set containing benchmarks and stability tests designed for evaluation of a different computation node configuration for *password recovery* tasks. Each benchmark or test focuses on different aspects. By its end, the chapter shows the results of the measurements and their analysis. However, there are several viable options on how to benchmark *GPU*:

- *rendering* – stressful testing of *GPU* aiming at graphics rendering, memory bandwidth and other, using DirectX, Vulkan, Mantle, OpenGL, example tool: 3Dmark[1];
- *Floating-Point Operations Per Second (FLOPS)* – benchmark stressing computation performance using floating point arithmetic, example tool: GPUBench project[2] [25];
- *memory latency* – latency of on-card memory using different utilization strategies, example tool: GPUBench project;
- CPU ⇔ GPU *data transfer* – the speed of data transfer between *CPU* and *GPU* memories, example tool: GPUBench project;
- *application specific* – some applications, for example, *hashcat*, *John-the-Ripper*, contain benchmarks of their own, looking for attributes based on their need; and
- many others.

This thesis contains only benchmarks provided by tool *hashcat* as it is the tool using the *GPU* resources in systems mentioned in Section 2.2. Benchmarks output the determined speeds of supported hashing algorithms [44], reached temperatures, *GPU* utilization, memory usage, and others.

4.1 Designed benchmarks and tests

One of the benchmark and test goals is to compare how different operating systems and *GPU* drivers affect the recovery speeds and temperatures. Therefore, measurements of most

[1] http://akamai-dl.futuremark.com.akamaized.net/3dmark-technical-guide.pdf
[2] http://graphics.stanford.edu/projects/gpubench/

of the *GPUs* come from two different *OSs* each with their specific stable drivers installed. The systems and drivers are as follow:

- *Windows 10 Professional build 1809* – Radeon Adrenalin 19.1.1 for the *AMD* cards and GeForce Game Ready Driver 416.94 *WHQL* for the Nvidia cards; and
- *CentOS 7.5* – AMDGPU Pro 18.40 for *AMD* cards and Linux X64 (AMD64/EM64T) Display Driver 410.78 for Nvidia cards.

Hashcat provides a minimal amount of data. Access to additional data is highly platform specific. It is possible to read hardware temperatures by tools like *psensors*[3] on Linux and by *OpenHardwareMonitor*[4] on Windows. However, *psensors* may struggle with a reading of the *GPU* temperatures, so the *System Management Interface (SMI)* included in *GPU* drivers is used instead. A tool called Performance Monitor is yet another possibility as it is already present as a part of the Windows *OS*.

The benchmarks aim to stress the computation node and get the highest possible speeds that node, in a given hardware configuration, can provide. Also, it should be easier to identify the bottlenecks and differences when comparing some of the benchmarks. During the benchmarks, the script always runs *hashcat* with the `--workload-profile` parameter set to mode four (Table 2.1).

4.1.1 Stability testing

Most of the other benchmarks below — especially those who directly use *hashcat*'s included speed benchmarks — have a short execution period. During short benchmarks, the cards are under the full load for only a few milliseconds. That makes them insufficient for testing of the maximal temperatures, average performance, and average power consumption. Therefore, this benchmark concludes of mask attacks against two different hash-modes running for 20 minutes each. The first of the hash-modes is Message Digest 5 Algorithm (MD5) as it is one of the fastest hash-modes supported. On the other hand, the second hash-mode is *Bcrypt* as one of the slowest hash-modes and the one with a high number of computation iterations.

Hashcat uses two different approaches to where it generates password candidates. For the fast hash-modes, it generates the candidates on the *GPUs* into *GPU* memory and then computes their hashes using the desired hashing algorithm. Another approach is the generation of the candidates on *CPU*, and then loading them to *GPU*(s) afterward [44]. Therefore, the benchmark conducts measurements of the utilization, power-drain, and maximal temperature of the *GPU* for each of these approaches.

4.1.2 Usage of the operation memory

The *CPU* and *GPU* communicate and exchange data over the *PCI-e* bus. The system reserves the space for all *GPU* buffers — used during *CPU* $\leftrightarrows$ *GPU* communication — in the main memory to ensure that data from/to *GPU* fits the *RAM*/swap memory space. As mentioned in Section 3.1, the node has limited operating memory to 32 GBs. However, all *GPU* selected for benchmarks operate with 4 to 11 GB of memory (*Video Random Access Memory (VRAM)*). Therefore, the memory may limit the number of usable *GPUs* and affect the stability of the *OS*. However, *hashcat* cannot use all of the *GPU*'s memory due

[3] https://wpitchoune.net/psensor/
[4] https://openhardwaremonitor.org/

to driver and card limitations. However, there are Equations (4.1) and (4.2) representing the minimal and recommended sizes of *RAM* for efficient computations using *hashcat* [44].

$$RAM = 0.25 * VRAM \tag{4.1}$$

$$RAM = 0.75 * VRAM \tag{4.2}$$

This set of benchmarks (measurements) aims to test the usage of *RAM* when using a different number of *GPU*. The set tests the usage of one, two, four, and of eight *GPUs* at the same time. The first measurement focuses on whether the system is even able to handle as many cards without losing its stability. The second part aims at the *RAM* usage by the *hashcat* process for various numbers of used cards.

4.1.3 Benchmarks of all algorithms

The first two benchmarks get speeds of all supported algorithms. The goal is to determine the speed difference between optimized and non-optimized kernels mentioned in Section 2.1.2. The benchmarking uses a script looping over all hash-modes to avoid *hashcat* instabilities due to node configuration or some driver. Because otherwise, it results in benchmark interruption when simply using the parameter `-benchmark-all`. The script creates a new instance of *hashcat* for each hash-mode resulting in higher overhead caused by repeated full initialization of the tool. However, this approach prevents the issue when one of the hash-mode computations fails and therefore prevents the computation of all following hash-modes as *hashcat* exits on any error.

4.1.4 Comparison of speeds of dictionary and mask attack

The last set of benchmarks composes of the dictionary and mask attacks since there are significant differences in speed of these attack for most of the hash-modes. The following hash-modes cover the speed ranges of most of the supported hash-modes. The used hash-modes are *Secure Hashing Algorithm 1 (SHA-1)*, *Secure Hashing Algorithm, 256-Bits (SHA-256)*, *Bcrypt*, *Scrypt*, *Roshal Archive, version 5 (RAR5)*. The hash values are from *hashcat*'s example hashes[5]. These benchmarks aim to detect whether and how big is the speed difference of the attacks. Also, the data from this set should also help to identify the bottlenecks in the node's architecture and overall speed limits based on the attack types.

4.2 Results of benchmarks and tests

The benchmarks and test proposed in Section 4.1 aim to identify candidates for both of the mentioned node configuration approaches. This section takes a look at different attributes during the execution of all benchmarks and tests. The interest attributes are:

- *system stability*;
- *computing power* – generally defined as *Millions of Instructions Per Second (MIPS)* or as *FLOPS*, but hashes per second are more accurate metric in the context of this thesis [7, 25];
- *memory usage* – the amount of memory used by *hashcat* and also required by *OS* to operate with all of the components;

[5] https://hashcat.net/wiki/doku.php?id=example_hashes

- *power drain* – the amount of power drown over the time of benchmarking in watts (kilowatts) per hour [W/h] measured using power strip equipped by Sonoff Pow[6] with Sonoff-Tasmota firmware[7];
- *power efficiency* – the amount of password converted into specified hashes-mode per watt as described by Equation (4.3) [19, 25];

$$E_h = \frac{hashes}{watt}[H/W] \tag{4.3}$$

- *initial costs* – the amount of money spent on the purchasing of components;
- *operating costs* – average power draw per hour [kW/h] converted to currency using average price 4.07 CZK/kWh [17] (0.157356 EUR/kWh) [28][8] in Czechia; and
- *operating temperatures* – temperature during the computation on Celsius scale (room temperature: 21-22 °C).

The measurements provide data for all *GPUs* models and both *OS*, except for the Nvidia RTX 2080Ti. Therefore, the measurements provide only data from the Windows *OS*. The cause of this issue is delayed card's delivery, which results in complications with testing machine's accessibility and new usage restrictions forbidding installation of other *OSs*.

4.2.1 Stability of the systems

Stability is the first and probably the most important aspect to consider. It is affected by the used *OS*, *GPU* drivers, all of the hardware components, operating temperatures, quality of *PSU* and stability of its output power, and other.

The picked releases of the *OSs* are considered to be stable by default as well as the version of the installed drivers. Also, the maximal output power of the selected *PSUs* is significantly higher than the calculated component's maximal input power. That leaves the hardware components and their cooling as the most significant variable affecting the stability of described node.

Software instabilities

The configuration, including eight Nvidia GeForce RTX 2080 cards, is the only one experiencing some major stability issues. The issues arise when trying to run the Windows 10 *OS* with more than three cards. Windows then reports an issue of running out of resources, probably the operation memory (including the virtual part of it), after adding the fourth card. However, even manual increasing of the available virtual memory has got no effect on the system's behaviour. Therefore, the data for the benchmarks and tests on this system are missing in the overviews.

The same configuration also experiences issue on the CentOS. Any attacks following the dictionary attack against the fast hash-modes are getting stuck during the computation. The issues occur during initialization *GPU* process of the new attack. All the other node configurations work properly without any recognizable instabilities.

[6] https://www.itead.cc/sonoff-pow.html
[7] https://github.com/arendst/Sonoff-Tasmota/wiki
[8] Ratio: 25.865 CZK/1 EUR

Cooling

In the used chassis, the *GPUs* have to be placed tightly together, and all heat they exhaust flows across all of the other components before it gets out. Therefore, the chassis fans have to provide a sufficient airflow to exhaust all the emitted heat.

Table 4.1 shows the achieved temperature for two long password recovery tasks. The table further shows that none of the cards is overheating. Therefore, both of the cooling solutions and the chassis fans provide enough cooling power.

Models	Minimal [°C]	Average [°C]	Maximal [°C]	Critical [°C]
GTX 1050Ti	36.00	52.73	71.00	97.00
GTX 1060	38.00	52.17	66.00	94.00
GTX 1070Ti	43.00	56.88	74.00	94.00
GTX 1080Ti	51.00	69.72	86.00	91.00
RTX 2080Ti	49.00	61.43	70.00	89.00
RX 580	no data	no data	no data	-

Table 4.1: Minimal, average and maximal temperatures of GPUs during two 20 minutes long attacks against MD5 and *bcrypt* hashes compared to critical temperatures obtained from the manufacturer [4]. The temperatures for AMD RX 580 could not be recorded because they were not reported by the driver.

4.2.2 Usage of operation memory

The *Portable Operating System Interface for Unix (POSIX)* tool `time` can except measuring the execution time of a process also measure maximal memory usage. It is possible via its parameter `--format="%M"` followed by the command to execute. The tool is available on Windows as a part of the MSYS2[9] platform.

Models	1 GPU [kB]	2 GPUs [kB]	4 GPUs [kB]	8 GPUs [kB]
GTX 1050Ti	831 104	1 163 032	1 873 584	3 325 104
GTX 1060	824 876	1 171 548	1 876 956	3 328 644
GTX 1070Ti	837 520	1 171 384	1 887 940	3 345 828
GTX 1080Ti	830 620	1 169 508	1 871 908	3 322 188
RX 580	675 292	939 824	1 464 368	2 523 416

Table 4.2: Usage of operation memory by *hashcat* process on CentOS with different number of GPUs.

As Table 4.2 shows the amount of used memory by *hashcat* increases when enabling changing the number of *GPUs*. Moreover, the memory difference per each of the *GPU* is approximately 340 MB for Nvidia cards and about 260 MB for AMD cards. However, Table 4.3 shows that MSYS2 `time` presumably struggles to read process information properly .

4.2.3 Speed benchmarks of all algorithms

GPU performance is undoubtedly one of the most important facts when buying a new card. Table 4.4 shows that Nvidia RTX 2080Ti significantly out-performs the Nvidia GTX 1080Ti

[9] https://www.msys2.org/

Models	1 GPU [kB]	2 GPUs [kB]	4 GPUs [kB]	8 GPUs [kB]
GTX 1050Ti	256 512	256 512	256 512	256 512
GTX 1060	256 256	256 512	256 512	256 256
GTX 1070Ti	256 768	256 512	256 512	257 024
GTX 1080Ti	256 512	256 768	256 512	256 512
RTX 2080Ti	264 192	264 192	263 936	264 192
RX 580	255 744	255 744	255 744	256 256

Table 4.3: Usage of operation memory by *hashcat* process on Windows with different number of GPUs.

by 124.90 % on average. Thus, the Nvidia GTX 1080Ti is no longer the most potent gaming *GPU* for based on the *password recovery* speeds. It is also apparent from the Tables 4.4 and 4.5 that differences in performances of Nvidia GTX 1060 and AMD Radeon RX 580 are quite insignificant.

Kernel-type	1080Ti	1070Ti	1060	1050Ti	RX 580
Non-optimized	0.00 %	-29.26 %	-58.93 %	-77.78 %	-65.17 %
Optimized	57.48 %	11.60 %	-38.69 %	-66.81 %	-41.73 %

Table 4.4: The average performance gain/loss of GPU models across all hash-modes compared to non-optimized kernels on Nvidia GTX 1080Ti on CentOS.

Kernel-type	2080Ti	1080Ti	1070Ti	1060	1050Ti	RX 580
Non-optimized	78.95 %	0.00 %	-26.05 %	-56.57 %	-76.49 %	-62.22 %
Optimized	170.84 %	60.71 %	17.83 %	-35.10 %	-64.78 %	-36.79 %

Table 4.5: The average performance gain/loss of GPU models across all hash-modes compared to non-optimized kernels on Nvidia GTX 1080Ti on Windows.

4.2.4 Speed of dictionary and mask attacks

All charts in Appendix B present the speed comparison of the dictionary and mask attacks. Figure 4.1 combines some of them to show how significant are the speed differences of the dictionary and mask attacks when targeting fast hashes. While it also shows the speed similarities of those attacks when slow hashes are the target. Furthermore, the other figures in Appendix B visualize the same data but from the speedup perspective.

What stands out in Figure 4.1 is how the speeds of all *GPUs* reach the almost same limits for dictionary attack on fast hashes. While the mask attack speeds differ significantly form one *GPU* model to another. Thus, it shows the hardware limitations affect the dictionary attack speeds.

4.2.5 Average power consumption and peak power drain

The Appendix C contains figures show-casing the power consumption comparisons of all node configurations. One of the figures also shows the total duration of the benchmarks as the computed values could be misleading without it. Since initialization times of each *GPU* with different drivers vary, the ratio between idle and full-load power states changes

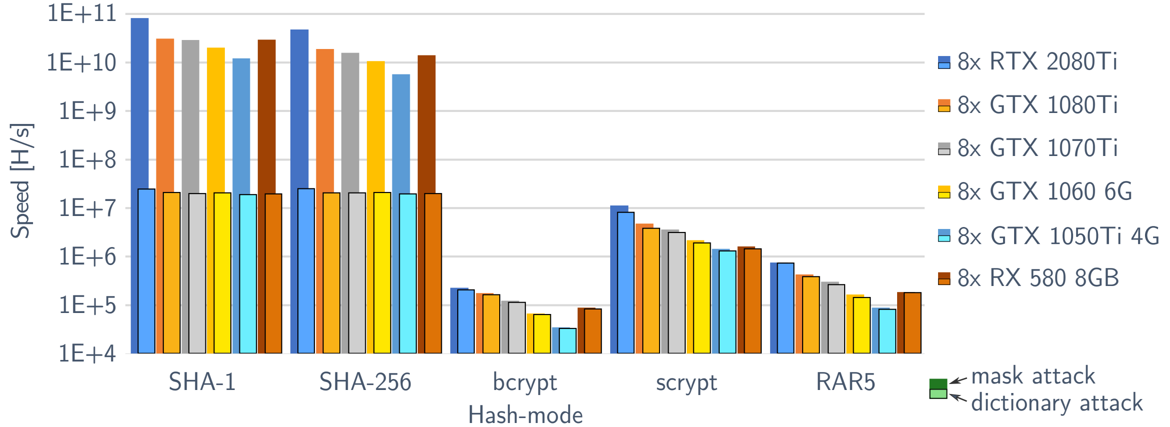

Figure 4.1: Comparisons of summarized average speeds each model reached for the five different hash-modes.

as well. Therefore, the total power consumption can be the same but consumed over a various portion of time by the end of the benchmarks. Finally, the maximal power consumption comparisons presented by Table 3.1 confirms that the maximal power drains specified by manufacturers are accurate.

4.2.6 Initial and operation costs

The *GPU* prices vastly differ as shows Table 3.1. The recovery speeds and the power drain when the cards are under full use differ significantly as well (Sections 4.2.3 and 4.2.5). This section adds other values to consider, and gives different perspective to the numbers from previous sections.

Based on the benchmarks and the measurements, it is possible to evaluate other, slightly hidden, aspects of different node configurations like maximal (theoretical) power drain over a longer time period or a ratio between *GPU* purchase and power consumption costs. However, they may be greatly important when deciding what *GPUs* should a system architect use when building or extending a cluster.

The figures in Appendix D present a computed hash per total cost overview. The figures consider a longer uninterrupted computation time periods up to 3 years and including initial (purchase) node costs as well. The figures use mask attack speeds from Section 4.2.4 to compute the total amount of computed hashes over the period. They also use values from Section 4.2.5 as well as *GPU* prices from Table 3.1 and node's other hardware (from Section 3.1) price evaluation, approximately 2239 EUR. The Equation (4.4) is the formula hashes per 1 EUR (H_e) computation.

$$H_e = \frac{(seconds_in_year * hashes_per_second * years)}{(8 * gpu_cost + other_hardware_cost)} \tag{4.4}$$

The considered computation time periods end at 3 years. The reason is that the new and more potent high-end *GPU* models are generally released every two to three years. They bring significantly increased performance for the same, or at least similar, power consumption e.g., Nvidia GeForce GTX 1080Ti (Section 3.2.1) and Nvidia GeForce RTX 2080Ti (Section 3.2.1). Moreover, the design of the gaming *GPUs* does not take into account 24/7 computation and high temperature over a long term. The cooling fans may wear off as well as the electrical part can get damaged over time.

The figures in Appendix D highlight how crucial the purchase price and differing power consumption may be for some hash-modes. E.g., Nvidia GeForce GTX 1060 6GB gets more cost-effective than AMD RX 580 8GB after 1.5 years when computing *SHA-1* and cannot even reach the same card for *bcrypt* in 3 years. Another interesting information provided by the figures is Nvidia GeForce GTX 1080Ti and Nvidia GeForce GTX 1070Ti having better cost efficiency than Nvidia GeForce RTX 2080Ti on *bcrypt*.

4.2.7 GPU selection

The chart in Figure 4.2 shows the decision areas for which individual node configurations are relevant. The areas aim to simplify the configuration selection by considering price, power consumption, and performance. The selection process then works as follows:

1. Set the priority point on power consumption axis in range $(0, 1.0)$.
2. Set the priority point on node configuration price axis in range $(0, 1.0)$.
3. Set the priority point on node performance axis in range $(0, 1.0)$.
4. Draw a line between the priority points to set a new requirements area.
5. Evaluate which *GPU* area fits the requirement area the best.

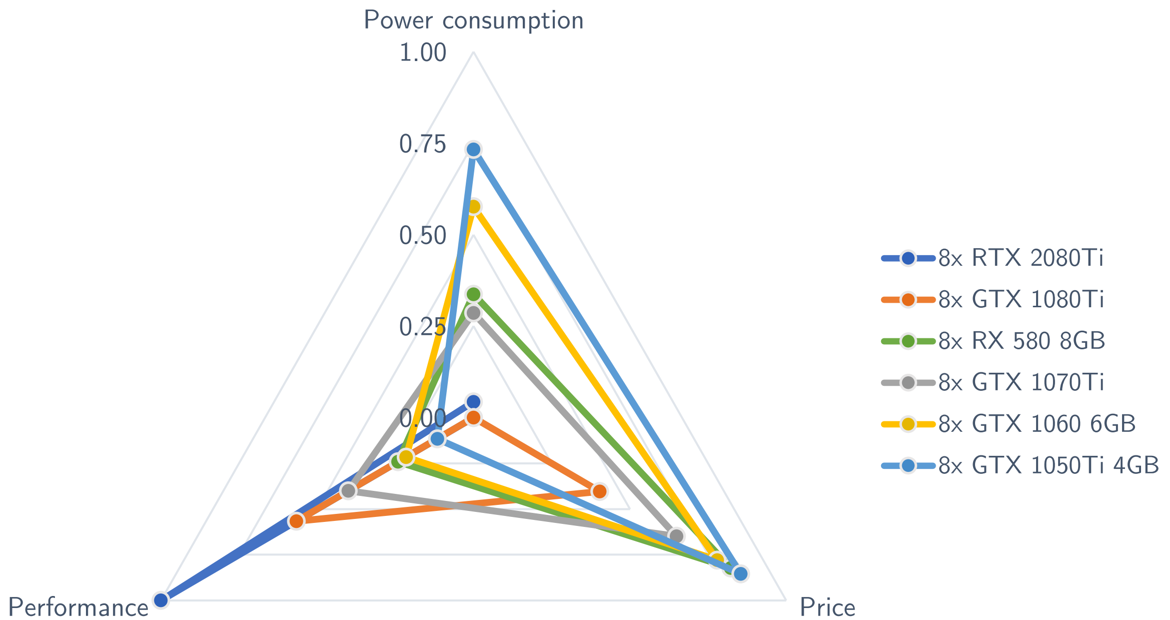

Figure 4.2: GPU selection decision chart visualizing node configuration relevance areas formed by the following coefficients. The chart's purpose is to compare different models rather than provide precise values. **Price coefficient** - higher the value, more affordable is the node configuration. **Power consumption coefficient** - higher the value, least power consuming is the node configuration. **Performance coefficient** - higher the value, more potent is the node configuration.

Chapter 5

Recommendations and strategies for password recovery

When it comes to password recovery, several aspects are influencing the complexity of the task. The most significant impact has the hashing algorithm. The algorithms differ from simple one iteration hashing functions like *MD5* and *SHA-1* through functions like HMAC-SHA256 to PBKDF2-HMAC-SHA256, scrypt, and bcrypt which use multiple iterations of hashing functions to make the *password recovery* task significantly harder. Thus, as the recovery speed changes, various attack recommendations and strategies should take place in order to be as efficient with this task as possible.

Therefore, this chapter introduces some of these recommended steps to consider before starting the password recovery. Section 5.2 then discusses the strategies for distribution of prioritized recovery tasks to multiple nodes in a stable environment (cluster).

5.1 Recommendations

From time to time, experts and authors of password recovery tools share some tips about how to use the full potential of the tools. Incorporation of some tips is more straightforward than of others. Moreover, some tips are computation resource demanding while another demand more of user time. Following paragraphs mention and explain some of them:

- The inputs of tools like *JtR* and *hashcat* can contain multiple hashes and try to recover passwords for all of them at once. This feature can, therefore, lower the recovery time compared to targeting the hashes individually.

  When put into the context of the distributed systems, which generally have multiple users independently adding different tasks with different attacks, it is convenient to gather all not yet recovered hashes of the same hash-mode and run the attacks on top of all of them. However, this approach is viable only for simple hashing-function without salt or any other input variable except the password.

  The password recovery for these simple hashing-functions composes of computation of the hash from the candidate followed by a comparison of the result to the targeted hash value. It takes advantage of the fact that computation of the hash is more resource expensive while the comparison of values is trivial.

  On the other hand, more complex hashing algorithms incorporate salts, key stretching functions [39], multiple iterations of key scheduling or hashing functions [40], and

checksum comparison of the decrypted data and the origin data checksum. Therefore, the tool has to recompute hash values for every entry separately when processing multiple input hashes. Thus, the number of computed hashes increases task's keyspace significantly. Each unique salt value results in the new hash computation for each password candidate.

- Since the single crack mode is recommended by *JtR* author as first to use [12], it is probably worth to research the person who created the password and therefore secured the data of interest (e.g., the person's native language). The user can then supply the recovery tools with the gathered information in the form of dedicated dictionaries (e.g., only English words). Since such a dictionary is considered to be shorter, an attack involving a high amount of rules is executable with smaller computation times.

- The sorting of the dictionaries by the password length and then alphabetically is advantageous [12]. For some formats (like 7-zip), the recovery speeds differ based on the lengths of the passwords. Therefore, the recovery speeds are more likely to drop when passwords of different lengths are mixed [44].

- Another thing to consider when dealing with the dictionaries is the line-ending (i.e., password separator). Dictionaries may come from a different *OS*, and multiple dictionaries may use different line-endings which might cause troubles later on.

- When dealing with huge dictionaries, it may be needed to divide the dictionaries into smaller ones due to the possibility of overflowing the maximal keyspace value depending on what data typed the specific tool uses.

- Also, when using *hashcat*, it is recommended to use the hand-optimized kernels (Section 2.1.2) as often as possible. They generally provide significantly higher speeds.

5.2 Strategies

Before performing any set of attacks, NETMUX LLC recommends determining the expected duration of such attacks [32]. It is relatively easy to specify an incremental attack for 3 to 12 ASCII lower-case letters long passwords. The keyspace of such attack is significantly big and depends not only on the performance of computation resources but also on the targeted hash-mode. While it would be possible to run this attack against a simple *MD5* hash, it definitely would be an issue against bcrypt hashes or other more complex hashes. It means that the attack creator should also consider recovery speeds when designing the attack.

Due to the high speed differences between the slowest and the fastest hash-modes, some attack types are viable only for the faster hash-modes but would take way too long for the slow hash-modes. The incremental, mask, or any other exhaustive attack types are the perfect examples. They mostly have huge keyspace that can pose a challenge for slow modes when searching for a password in an acceptable time. Therefore, there are some recommendations on how to eliminate those exhaustive attacks. It seems to be the best to run the attacks in this order [32]:

1. *custom wordlist* – using small dictionary based on information about the hash's creator;

2. *custom wordlist with rules* – using small dictionary based on information about the hash's creator with a few mangling rules;
3. *dictionary attack* – using a dictionary of popular passwords;
4. *dictionary attack with rules* – using mid to big sized dictionary with a modest number of rules;
5. *custom wordlist with rules* – using the custom wordlist extended by the already recovered password with more and more subtle mangling rules;
6. *mask attack* – using masks extracted from the popular passwords from leaked datasets or commonly know patterns like `?u?l?l?l?l?s?d?d`, `?u?l?l?l?l?d?d?s`, and many other;
7. *hybrid attacks* – using any dictionary with a prepending or appending masks;
8. *combinator attack* – using combination of two dictionaries; and
9. *brute-force attack* – using an incremental masks of `?a` up to eight position long passwords or longer depending on computation system power.

When it comes to the distributed *password recovery*, it is challenging to perform any attacks based on pre-generated or leaked password stored in dictionary files. Such attacks can easily overload the controller and potentially network in the systems with a single controller and multiple computation nodes (e.g., FITcrack, Hastopolis). The controller — in such a system — has to prepare and transmit the dictionaries to the nodes. Therefore, the controller's network interface can become the bottleneck when it has to accommodate several node's needs. Also, considering the speeds of the fast hash-modes, each node then can require hundreds of thousands, millions, or even billions of passwords per second.

The distributed dictionary attacks on fast hash-mode are not easy to perform, and their effectiveness is not ideal [22]. Therefore, the total amount of transmitted data should be reduced. Either by adding compression and decompression steps or by limiting the number of nodes performing the dictionary attacks. The other nodes can meanwhile work on the other attack types with lower bandwidth requirements.

Another option is to add more work to each node by using rules. Rule files are generally few lines long, but can greatly increase the keyspace of an attack. Therefore, dictionaries can be smaller and more nodes can use a single controller. Furthermore, the rules also increase the computation node recovery speed for dictionary attack when using *hashcat* with its built-in rule engine kernel. The simple version of the distributed dictionary attacks is feasible for any hash-modes with lower recovery speeds, which bandwidth requirements are not as high [22].

Chapter 6

Proposed system changes

The current version of FITcrack has multiple design flaws, which are limiting its full potential, effectiveness, and scheduling precision. Not only its scheduling struggles with salted hash-modes, but it also does not handle transitions between masks of different sizes all too well.

6.1 Salted hashes

Generally, FITcrack does not handle the salted hashes well. It does not reflect the increasing number of computed hashes when the tasks contain multiple salted hashes. This results in the incorrect size of generated *workunits*, which is crucial as *workunit* crafted to last one hour may take several hours or maybe even days to compute.

The system has to reflect the number of unique salts or at least salted hashes and the increasing number of hashes to compute throughout the *workunit* generation process. FITcrack computes several hashes in a *workunit* by multiplication $recovery_speed * desired_workunit_duration$. This value has to be divided by the number of salted hashes, or even better by the number of unique salts if the *workunit* size should be accurate.

6.2 Transition between masks

A single task can contain multiple masks of different lengths and keyspace. Therefore, it is necessary to adjust the *workunit* keyspace calculation when moving from one mask to another. FITcrack recognizes the keyspaces of different masks, but the computation adjustment between masks is missing. It results in a *workunit* with either much bigger or much smaller duration. That makes *workunit* sizes unpredictable and scheduling difficult. FITcrack has to convert the speed of mask attack from mask indexes per second to hashes per second and perform the conversion back and forth when needed. It is principally similar to operations required for salted hashes Section 6.1. Therefore, both issues may use similar code and require identical actions to fix them.

6.3 Inclusion of unrecovered hashes

The unrecovered hashes (i.e., hashes for which are not yet found the password) start to pile up when the user uses the FITcrack system with real data. It is a common practice to perform a new dictionary attack against unrecovered hashes once a user gathers a new

dictionary. However, the system — at its current state – forces the user to re-insert all of the hashes even though it had got hashes already.

Both *hashcat* and FITcrack support hash-files[1] for all non-binary hashes and thus a password recovery of multiple hashes at once. Implementation of this new feature would remove the step of re-inserting the hashes. Instead of that, the system can perform a rather simple database query to retrieve all unrecovered hashes of given hash-mode.

Although the tasks support salted hash-modes. The keyspace of such task grows with every hash unique salt, which makes it harder to go through all password candidates with each salt. Thus, the system cannot perform this feature automatically, or at least not in all cases as it can make the task inexhaustible. The user has to be responsible for the decision of whether to turn on the feature or not.

6.4 Distribution of dictionaries

A distributed dictionary attack is a challenging task, as already explained in Section 5.2. FITcrack reduces the data volume, which has to be transmitted from the controller to computational nodes over the network. Its *workunit* generator crafts dictionary fragments from the dictionaries based on the computed size of new workunit. While that leads to precise *workunit* sizes, this approach adds a considerable amount of overhead which grows with the size of the fragmented dictionary.

The generator goes through the dictionary line by line and dumps the lines into a new file. This approach results in a significant overhead when dealing with huge dictionaries (tens to hundreds of gigabytes). With this approach, the generator steps through the dictionary sequentially from the start to the reach position where it ended in the previous run. Therefore, the generator spends a lot of time time just skipping through the dictionaries instead of doing any useful work. That results in the computation node running idle instead of checking hashes.

The repetitive dictionary fragmentation is another generator's issue as generator assigns one fragment to only a single *workunit* and then deletes it to preserve disk space. Therefore, the generator has to always generate new fragments even if there are two same tasks following each other and has precisely the same nodes assigned to compute them. That results in many disk *I/O* operations, which slow generator significantly. The problem is even worse; the generator is not multi-threaded. So, every extensive operation for any host blocks generator for all other hosts.

The dictionary fragmentation, in general, seems to be a valid approach, but the system should pre-compute the fragments after adding a new dictionary. That should remove the overhead tight with the fragmentation. It also allows execution of other operation like lexical sorting, password length categorization, filtering of password with invalid symbols, or any other pre-processing or in-advance password analysis. Furthermore, the system components running on the controller also handle transmissions of fragments to nodes. Transmission can fully exhaus controller's resources like network interface or even data storage. Incorporating some a high-throughput shared network storage may be advantageous. Especially for the case when FITcrack runs on a stable cluster. Such clusters mostly have a robust architecture with high bandwidth interconnections. Thus, network

[1]Files containing multiple unique hashes of the same hash-mode.

filesystems — like Gluster[2], BeeGFS[3], Samba[4], and similar — combined with *Storage Area Network (SAN)/Network Attached Storage (NAS)* storage become a suitable option. They should provide faster file access times, easier file manipulation, and higher throughput than the controller's current approach using file transmission over *HTTP/Hypertext Transfer Protocol Secure (HTTPS)*.

6.5 Task prioritization

FITcrack treats all user tasks equally. That is a *fair-play* approach which does not correlate with reality. In real-world scenarios, each task has some kind of priority. The priorities become even more critical when multiple users operate the same system as the users may have different roles. Priorities might be anything, but likely one of the following:

- manually set priority;
- user role;
- attack type;
- attack keyspace;
- task adding time; and
- potentially some others.

The expectations are that a single priority is not enough as there are multiple possible aspects to consider. Therefore, the following sections propose a four-level priority system which covers the basic priority set, leading to unambiguous task selection. The level sections follow in the top to bottom order.

User task priority

Each user determines the input data priority differently. Therefore, this priority level is not algorithmically assignable as are the other levels. Therefore, there have to be a few predefined user-assignable priority values from which the user can choose. This priority level is the most crucial from the user perspective. The user can categorize the input data (hashes) by their importance and by that adjust the order in which the system computes the tasks.

Attack type priority

The motivation behind this second priority level is to go through the attacks from the least exhaustive and specifically targeted attacks to the more general and more exhaustive ones. The list from Section 5.2 recommends an attacks order based on their types. That list is convertible to a slightly reduced priority list. The conversion is necessary since FITcrack does not know the attack's context. Therefore, it is necessary to incorporate recommended list items generalization like considering the wordlists and dictionaries equal, or all masks as equal.

[2] https://www.gluster.org/
[3] https://www.beegfs.io/
[4] https://www.samba.org/

Attack keyspace priority

The third priority level reflects the task difficulty. Task keyspace serves as the difficulty indicator since the hash-mode's complexity is unchangeable. The motivation behind this priority level is to compute tasks from the easiest to the hardest to compute when the two previous priority levels are identical for several tasks.

The hash-mode should not affect the task order and the selection of keyspace over the task's time to compute. The user should consider the hash-mode complexity in the first priority level (Section 6.5). Alternatively, organization and companies can incorporate this into their user priority user manual.

Task creation time

The last priority level is necessary to decide any priority draws of several different tasks. That leads to a need for some unique value resolving potential ties. New tasks emerge over time. Therefore, the *first-come, first-served* seems to be applicable and on point. Because each task has already got its adding time, which well fits this priority level purposes.

6.6 Task scheduling

FITcrack's task generator incorporates the formula proposed by Hranicky et al. in their paper [20], where they discuss the motivation behind it and the system's use-case. There introduced formula considers the yet uncomputed part of a given task and divides it by all active node speeds sum. This way, it obtains the remaining computation time required for finishing the task. Then, it multiplies the resulting time by a so-called *distribution coefficient* which limits the upper time bound. That approach results in *workunit* size shrinking with every computed part as the task computation progress. Unfortunately, it can lead to a significant overhead as it generates more and shorter *workunits* in order to utilize all available nodes with a single task.

Even though that approach might fit the original FITcrack's targeted use-case, the use-case includes unstable and dynamic topologies, various node configurations, and a different physical node location. That use-case aims at using FITcrack with standard office computers as computation nodes. Meanwhile, the real-world deployments show the inexistence of such use-case as office computers could not withstand the heat and long term utilization. Furthermore, the office data and power network designs do not consider such a load. Thus, the users rather use dedicated multi-GPU computation nodes designed for high utilization over long periods instead, which enormously changes the use-case.

Algorithm 1: Original FITcrack *workunit* scheduling algorithm.

```
1 foreach running task do
2     load assigned hosts
3     foreach assigned active host do
4         if host does not have enough work then
5             generate new workunit
```

Usage of dedicated computation nodes moves the use-case more towards a grid computing, stable topology, generally a centralized node location, high node availability, and a se-

cure environment. Therefore, the task scheduler has to reflect these new needs and also aim for the highest possible utilization efficiency.

The scheduling process has to undergo significant changes to improve usability in the new use-case. Algorithm 1 presents how the core of FITcrack's scheduling process operates. The process uses a task-centric approach which has got its flaws, and its extension by the proposed functionalities would be cumbersome. Therefore, there is need for a new, more suitable scheduling process fulfilling the following requirements (ordered by its importance):

1. It has to use a computation-node-centric approach to simplify the priority scheduling process. That should also reduce the number of idle *workunits* for every node attached to the system.
2. It has to use the task prioritization process proposed in Section 6.5.
3. It has to replace the original formula to keep *workunit* duration as long as possible.
4. It has to replace the current approach on dictionary fragmentation by using pre-fragmented dictionaries leading to significant overhead elimination.
5. It has to consider the fact that salted hashlists increase the overall the number of computable hash values in tasks.

Chapter 7

Implementation of proposed changes

This chapter describes the implementation of proposed changes. The sections below mention all the faced problems and complications when extending FITcrack. Following subchapters present the changes in the feature-based view instead of the system-module view.

7.1 Salted hashes and mask attack improvements

Adding support for salted hashes means changes to user input post-processing, node computation speed, scheduling, and others. A number of used salted hashes crucially changes task's difficulty (increases keyspace) so do the changes to masks. So, both the user and then the system has to be able to acknowledge this information. Therefore, the majority of server modules requires changes in order to provide everybody with as accurate as possible information.

Web-backend

FITcrack needs to identify which input hashes are salted in order to work with them correctly. Therefore, the hash-mode records in the database newly contain a flag whether the hash-mode is salted or not. *Hashcat* contains the information which hash-modes are salted in its source code. Thus, it is possible to extract the information with a script parsing the source codes which can export the data to a serializable data structure. Then, the hash-mode table extension is just a matter of database revision. Implementing this revision is unchallenging as this web-backend already contains migration and revision submodule called Alembic[1] providing required features.

Furthermore, the module's database models `src/database/models.py` incorporate the revision. That prevents any unintentional data accesses, which could cripple the module's functionality. Also, the module's REST API endpoints have to reflect on this change as well in their parsers and response models. Therefore, the endpoint `/hashcat/hashtypes` newly provides `is_salted` flag, which indicates the saltiness of hash-mode as it provides access to the values stored in the new database column.

[1] https://alembic.sqlalchemy.org/

Web-frontend

The number of hashes to compute when working with salted hashes increases with every additional hash on the input. As for the masks, it calculates the mask magnitudes, which it then sums to determine the task's complexity. Web-frontend uses this information when adding a new task or editing an already added task. The module computes and presents the estimated cracking time of all hashes to the user. Therefore, the `addJobView` newly checks, whether the selected hash-mode is salted. Web-frontend checks the hash's saltiness by examining the new flag of `/hashcat/hashTypes` endpoint mentioned in the previous section. Then, it uses a naive approach where it counts the hash entries which it then multiplies by the tasks keyspace. That provides worst-case results for a given task i.e., each hash having unique salt.

Generator

Work generator treats tasks targeting salted hashes a little differently as it adjusts the calculated *workunit*'s magnitude by using Equation (7.1). The equation incorporates an adjustment ratio, which corrects the value. It uses the same approach for both the mask attack and salted hashes.

$$workunit_magnitude = \frac{desired_workunit_time * speed}{\frac{total_hashes_in_task}{task_keyspace}} \tag{7.1}$$

Assimilator

Assimilator finishes the whole specialized treatment of salted hashes. It converts the *workunit* keyspace, defined as number of mask indexes, to a real number of computed hashes after receiving the host's result. It calculates the same ratio as the generator and applies it to reverse the keyspace adjustment used in the generator. Then, it calculates the achieved recovery speed for a given *workunit* employing the number of computes hashes and *workunit* duration.

7.2 Inclusion of unrecovered hashes

As already mentioned in Section 6.3, the feature simplifies adding not yet unrecovered hashes to a new task only enhances the user experience. It automatizes the process of adding previously inserted hashes to a new task base on their hash-mode. It may speed up the recovery process for multiple hashes. Although, the feature is available only for the un-salted and non-binary hashes because: (a) the task complexity grows significantly with every hash; and (b) *hashcat* does not offer support for multiple binary hashes.

Web-frontend

The web-frontend newly provides a switch button for turning the adding of unrecovered hashes on and off. The button uses information about the selected hash-mode to disable itself when the selected hash-mode is either binary or salted. It moves to the `off` state before it turns to the `disabled` state. The implementation of the switch button extends the `AddJobView` page. The switch state (`on`/`off`) becomes a part of the request message sent by submitting a new task to the system.

Web-backend

The web-backend's endpoint responsible for task creation newly accepts the value through the frontend switch. The value then affects whether it includes the previously inserted hashes or not. It then enlists all unrecovered hashes of the given hash-mode from the database via SQL query and links them the new task when the flag tells it to do so.

7.3 Distribution of dictionaries

The distributed dictionary implementation moves away from *BOINC*'s native file transmission over *HTTP* or *HTTPS* protocol. It keeps dictionaries on a dedicated storage device which is reachable by both the server and the computing nodes by using one of the possible *Network File Systems (NFSs)*. Each device, which wants to access a dictionary, can mount the dictionary to a path of its needs and access the files the same way of how it would access the local files. Because manipulation with big dictionaries would result in significant network load, the server pre-fragments the dictionary before placing it into storage device by using the developed high-performance tool.

Tool fragmenting dictionaries

The dictionary fragmentation tool normalizes an input dictionary by unifying the password separators and removing empty lines. Then, it analyzes all password and sorts them into separate dictionaries (fragments) by their length. Each output fragment contains up to 250 000 passwords. Smaller output files with fewer passwords provide a better overall fragmentation granularity. Which then enables more precise scheduling and shorter fragment download times.

The tool is implemented in C++ language, because the tool has to achieve a high performance. The application relies and builds on top of functions, classes, templates, and structures provided by the C++ language and its *Standard Template Library (STL)*. Other than that, it uses a set of custom classes and structures extending the functionality and simplifying the code readability.

The tool requires the path to input dictionary file (`-i`), fragment's size (`-s`), and output directory (`-d`). There is also optional parameter `-l` limiting the maximal password length. Otherwise, it accepts any password shorter than 4096 characters.

The basic C++ `getline()` function does not allow the detection of multiple possible delimiters. Therefore, the tool implements a function similar to `getline()`, which detects the most common line separators like *Line Feed (LF)*, *Carriage Return (CR)*, *CRLF*. Although, it does not keep context and it may not handle multi-byte characters containing delimiter like bytes correctly.

The implemented reading algorithm is faster than implementation using `getline()` as shows Table 7.1. The used algorithm reads binary data from file to a 4 kB buffer. It then processes the buffer in order to extract all of the passwords. The file pointer moves both ways throughout the runtime. It moves forward on file read, and it moves back to the position of last found delimiter when the delimiter misses at the end of the buffer.

Extracted passwords then tranferes to the part of the algorithm responsible for categorizing it to the proper fragments for given password length. The tool uses `std::vector` as storage of active fragment. The vector contains only one fragment per each password size, which allows using the password length as an index. This method eliminates any memory

Implementation	Passwords	Bytes	Wall time [s]	Throughput [MBps]
implemented	1.00E+08	614 368 919	5.80	101.04
standard			7.50	78.16
implemented	1.00E+09	8 678 038 624	74.02	111.93
standard			84.00	98.53

Table 7.1: Benchmarks of the different implementations using a custom and standardized `getline()` functions for reading passwords from a file, categorizing them by length, and storing them to new files. Benchmark configuration: *OS*: Ubuntu 18.04; *CPU*: AMD FX-8320 8-Core, 3.50 GHz; *RAM*: 16 GB 1866 MHz; source *SSD*: Kingston V300 120GB; destination *SSD*: Samsung 860 EVO 500 GB; both connected via *SATA* 6 Gb/s bus.

lookups and therefore results in constant memory access times. Because it keeps a password counter for each fragment, it can close the full fragment, flush its data to the driver, and replace it with a new empty fragment, which repeats over and over. Finally, the tool outputs formatted *JavaScript Object Notation (JSON)* string to *stdout* containing the path to fragment, password counter, password length, and fragment sequence number for given password size and input dictionary.

Generator

There are several required changes to the generator in order to add support for pre-fragmented dictionaries. The first change is removing the current implementation of dictionary fragmentation, which is tightly coupled with changes to the original file with the generated fragment. The file newly contains metadata of fragments assigned to host. In order to do that, it has to enlist unused dictionary fragments linked to the task. Only then, it selects specific few enlisted fragments and adds them to the *workunit*. Generator adds the fragments until their keyspaces sum reaches the calculated *workunit* size. Alternatively, it uses at least one fragment, when the *workunit* size is smaller than keyspaces of fragments. Finally, the generator creates a *JSON* structure (Listing 7.1) from the metadata and stores it into the file which it then sends to the host.

```
[
  {
    "path" :"<fragment_path_from_dictionary_root_directory>",
    "keyspace" :"<keyspace>"
  },
  {... },
  ...
]
```

Listing 7.1: *JSON* array of fragment metadata.

Runner

Runner requires modifications as well because inputs from the server have changed. It newly uses a minimal Rapidjson[2] library for parsing *JSON* structures. That allows Runner to use

[2] http://rapidjson.org/

a *JSON* based per host configuration file structure from Listing 7.2. This change to the file `ConfigHost.cpp` is necessary since the original one-line configuration format allows only the configuration of additional *hashcat* parameters. Runner currently requires the path to the directory with dictionaries i.e., the path of *NFS* mount-point.

```
{
  "hc_extra_params" :"<hashcat_parameters>",
  "network_fs_root" :"<path_to_mounted_dictionary_directory>"
}
```

Listing 7.2: Host configuration file structure.

Moreover, Runner employs the *JSON* library for fragment metadata parsing (i.e., parsing of structure from Listing 7.1 sent by the server). After the parsing, it stores the metadata to its internal representation and checks whether it can access the fragment(s).

Runner also uses a new approach to passing dictionaries (fragments) to *hashcat*. It newly spawns an additional process thread which reads the fragments as binary files to a buffer. The thread then writes the buffer to the *hashcat*'s *stdin*. This approach increases the achieved performance even when using small fragments as it prevents *hashcat* re-initialization for every fragment, which would reduce the performance significantly. However, *hashcat* can accept multiple dictionaries by itself. Although this approach increases *hashcat*'s and network overhead because it performs a dictionary analysis instead of directly running the attack. Therefore, it requires additional network traffic as each interaction with a file results in data re-transmission, because the *NFSs* does not store local copies.

The need for multi-threading results in moving the code-base to the newer C++11 standard, which provides multi-platform threading implementation in form of `std::thread` class. Other alternatives would either: (a) require self-implementation of threading; or (b) require usage of a big library like Boost[3]. However, neither those alternatives would be better because solution: (a) could be buggy and would require extensive testing; and (b) could complicate both the compilation process and future development with newer versions.

Runner performs no password processing. It solely feeds *hashcat* with data from the file. Same as the dictionary fragmentation tool (Section 7.3, it uses 4 kB buffer and fast language C-like *POSIX I/O* functions. Those functions require minimal function calls and provide the maximal read and write speeds.

7.4 Task prioritization

Task prioritization is a new way of how to select a task for a given host in the context of FITcrack. It favors attacks with the highest potential of founding the password as it applies criteria and recommendations from Section 6.5. The modifications only affect the generator and database scheme.

Database

The task prioritization requires some database changes in addition to the need for new data. The starting database contains only some task metadata matching the needs of criteria from Sections 6.5 to 6.5. The task creation time and attack keyspace are straightforward and already in the database. However, the user priority is there only partially as there is an

[3] https://www.boost.org/

`INTEGER` column for storing some value. The system misses any description of the available user priority levels. Therefore, the original column transforms into a `FOREIGN KEY` pointing to the records in the new `fc_user_priority` table. The new table contains `id`, `priority`, and `name` values.

The original database completely misses any information about attack type priority (Section 6.5). Although, each task — in FITcrack's context called job — record contains information about job mode and submode. So, the job's mode and submode move to a new static table `fc_job_type` containing `id`, `mode`, `submode`, `description`, and `priority` columns. Therefore, the job records now contain a `FOREIGN KEY` to the `fc_job_type` table instead of the original `job_mode` and `job_submode` values.

Web-backend

Web-backend also needs to reflect the changes in its database models with additional virtual link of `mode` and `submode` from `fc_job_type` to the original `job_mode` and `job_submode` database object properties. Moreover, the REST API has to newly provide the list of available user priorities through a new endpoint `/attack/priorities` and it has to accept user priority when creating a new job.

Web-frontend

There are only two small things to add from the web-frontend perspective. The first thing is a new list to the `AddJobView` offering selection of user-defined priority. The list visualizes data from the new `/attack/priorities` endpoint. The second thing is passing of the priority information with the job data when submitting a new job to the API.

Generator

The implemented algorithm extends the generator's `SimpleGenerator.cpp` and `SqlLoader.cpp`. The algorithm implements the criteria from Section 6.5 as a complex SQL query. The query runs over several tables and uses various `WHERE` clauses specifying values of few columns. At last, it orders the selected records by several columns. It is not a simple nor easily readable query, but it is faster than fetching all records from database to memory and performing the filtering and ordering in the generator's code. The *workunit* generation takes place after the attack selection. It creates a new *workunit* for selected host using the recovery task data and settings.

7.5 Scheduling

The reworked scheduling algorithm uses online nodes as its main commodity instead of the existing recovery tasks which reduces the number of waiting *workunits* in the system to a minimum as well as it prevents *workunit* generation for offline nodes. It also reduces the algorithm complexity. The original algorithm's outer loop loops through all active tasks (n) and inner loops through all task assigned nodes (m). Therefore, its complexity is $O(n * m)$. The new version uses only a single loop on online nodes (p), where $p <= m$. The algorithm then fetches only the most prioritized task assigned to the node. That results in $O(p)$ complexity, as there is no looping through tasks.

The original generator uses the algorithm from Hranicky et al. publication at ICDF2C 2016 [20] for computation of new *workunit* duration. The algorithm has got its flaws for grid

Algorithm 2: Implemented altered task scheduling algorithm.

```
1 load online nodes
2 foreach online_node do
3     load highest prioritized assigned task(node)
4     if no_task_loaded then
5         continue
6     else
7         generate new workunit(task, node)
```

computing, as described in Section 6.6. Thus, a replacement of the algorithm is necessary. So, instead of that algorithm, the new *workunit* duration is straightly set to the desired *workunit* duration value without any changes to it. It provides a more consistent *workunit* sizes no matter the position in the or number other nodes assigned to the same task. That results in as big as possible *workunits* for every single connected computation node.

Chapter 8

Comparison against original state

This chapter presents the measurements and comparison results for implemented system modifications compared to the original system state. Tables in this chapter present precise data which stand as benchmarks. On the other hand, the charts mostly highlight the deviations between the implementations.

8.1 Mask attack improvements

Figure 8.1 shows how the implemented corrections affect the *workunit* size. That figure visualizes the duration of the first generated *workunit* axis Y and mask lengths on the axis X. The goal line shows the desired *workunit* duration, which is constant throughout the task. What stands out from the figure is how significant is the original implementation's issue as it generates *workunit* approximately 50-times bigger than the set goal. It also shows the adjusted implementation's correctness. There are two proofs: (a) it does not significantly overreach the goal line at any point; and (b) it correctly moves from the length of four to length of five and six, while the old approach generates same sizes for lengths four and five.

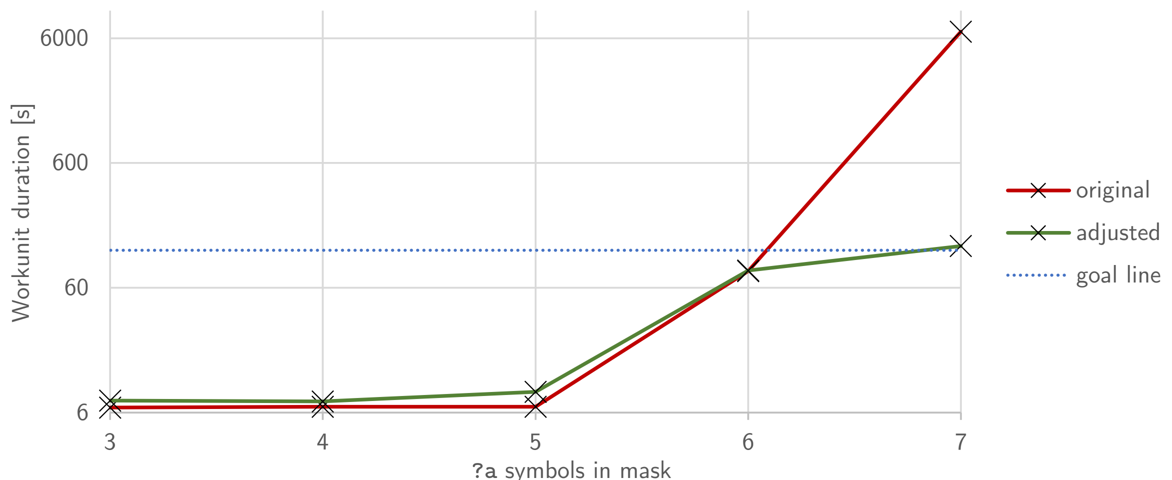

Figure 8.1: Workunit duration generated by the original and adjusted system using attack containing three to seven symbols (`?a`) long masks.

8.2 Handling of salted hashes

Similarly, as in the previous subsection, Figure 8.2 displays the *workunit* duration on axis Y, while the axis X — in this case — displays a number of used salted hashes. The figure and the Table 8.1 shows the practically exponential duration increase with the growing number of salted hashes for the old implementation. Although the new implementation performs significantly better, the duration still grows over the goal line. What is even worse is that the duration slightly grows with every added hash.

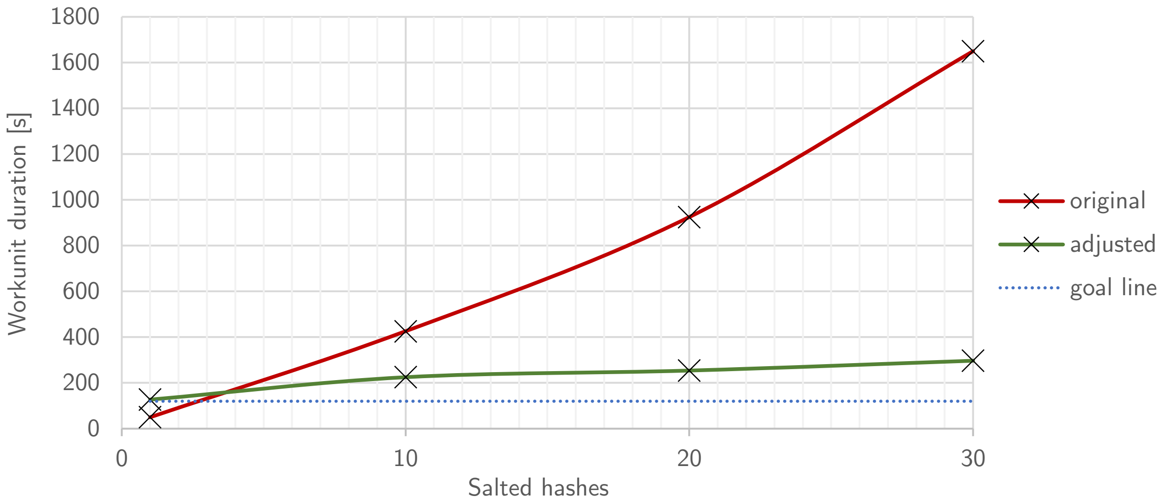

Figure 8.2: Duration of initial workunits using original and adjusted system with various numbers of salted hashes.

Salted hashes	Original algorithm		Adjusted algorithm	
	Duration	Keyspace	Duration	Keyspace
1	50s	244 926	2 m 07 s	869 706
10	7m 06s	231 444	3 m 45 s	85 865
20	15m 25s	247 440	4 m 14 s	35 133
30	27m 30s	291 000	4 m 57 s	28 508

Table 8.1: Initial workunit duration/size difference using single/multiple salted hashes comparing the original and adjusted implementations.

8.3 Distribution of dictionaries

Tables 8.2 and 8.3 provide a comparison between the two FITcrack versions:

1. version with improved in-generator fragmentation using stored last read byte position to for skipping already fragmented parts of the dictionary file; and

2. version with pre-fragmented dictionaries distributed using *NFS*.

The comparison uses one dedicated PC equipped with AMD Ryzen 5 2600X *CPU*, Patriot Burst 560/540 MB/s *SSD* as a server. The server not only runs all of the FITcrack modules but also contains the dictionary storage. It uses Samba daemon providing access

to the dictionaries. The main reason behind selecting Samba out of all other options is its easy deployment and its multi-platform usability. Other than that, it uses eight more hosts all equipped by a single Nvidia GTX 1050Ti 4G *GPU*. Each host runs CentOS 7 with Nvidia *GPU* drivers. They gain access to dictionaries via the *Common Internet File System (CIFS)* application-layer protocol using `mount.cifs` as a client. All devices communicate over a 1 Gb/s network.

Both tables present information about used dictionaries sizes and keypaces as well as the task duration from pressing the task's start button till its finishing. The first Table 8.2 provides data regarding a task using *SHA-1* hash-mode. The second table (Table 8.3) provides data for the task using a bit slower *Whirlpool* hash-mode. Data in column *Duration* show that the new approach is much faster for all dictionary sizes and both hash-modes. Even though, only a single host computes all of the hashes as shows the number of generated workunits.

Dictionary		Original		Adjusted	
Size	Keyspace	Duration	Workunits	Duration	Workunits
1.1 GB	114 076 081	2 m 40 s	2	1 m 17 s	1
2.1 GB	228 152 161	3 m 28 s	4	1 m 45 s	1
4.2 GB	456 304 321	4 m 2 s	8	2 m 11 s	1
8.3 GB	912 608 641	4 m 58 s	16	3 m 38 s	1

Table 8.2: Task duration measurements comparing file transfer method using SHA-1 hash-mode.

Dictionary		Original		Adjusted	
Size	Keyspace	Duration	Workunits	Duration	Workunits
1.1 GB	114 076 081	3 m 11 s	2	1 m 26 s	1
2.1 GB	228 152 161	3 m 14 s	4	1 m 46 s	1
4.2 GB	456 304 321	4 m 00 s	8	2 m 32 s	1
8.3 GB	912 608 641	5 m 02 s	16	4 m 05 s	1

Table 8.3: Task duration measurements comparing file transfer method using Whirlpool hash-mode.

8.4 Task prioritization

The task prioritization and automatized task reordering are hard to compare to the original approach. Such comparison would require examination of runtime data from a system using only real word data. It would require gathering of a long-term statistics for each used mask, dictionary, and attack type. Only a correlation of the long-term statistics against the proposed and implemented criteria would provide the needed results. However, the usage of artificially generated hashes could result in skewed results and conclusions because the person preparing and performing such tests would most probably be biased in the password candidate selection on either side of the comparison. Therefore, this thesis proposes only the modification and offers an experimental implementation. However, it does not evaluate the proposed approaches against any data.

8.5 Scheduling

The last comparison focuses on the impact of the modifications to the scheduling algorithm. It uses a mask attack created in a way to last about 25 to 30 minutes with desired 120 s long *workunits*. Figure 8.3 presents: a number of generated *workunits*, their duration, and total time purely spent by the recovery process. Meaning, stripped of all transmission, *workunit* generation delays, and other system overheads. The only left overhead is *hashcat*'s initialization, which is unignorable.

The Figure 8.3 further shows how the modified algorithm generates fewer *workunits* respecting the desired duration compared to previous implementation. It also shows how the algorithm overestimates the first two *workunit* sizes, which result in their prolonged duration. After that, it adjusts the sizes accordingly to the nodes maximal capabilities. The reason why it overestimates the first two *workunits* is due to the short and inaccurate benchmarks. On top of that, it highlights how the recovery efficiency increases as the recovery time of the modified version are significantly shorter, precisely 3 m 47 s shorter. In reality, the saved time is even longer as there are generation, transmission, Runner, and *hashcat* overheads tight with each *workunit*.

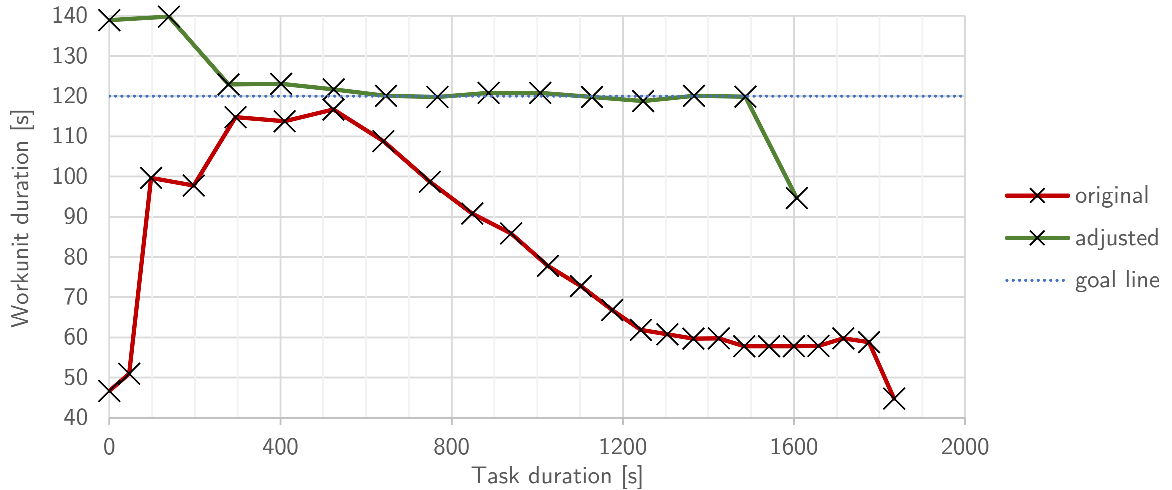

Figure 8.3: Original and adjusted scheduling algorithm workunit duration throughout task duration comparison without considering communication overheads.

Unlike the rest of overheads, the generation overhead in the original implementation grows with every active task and active host. Although, this thesis does not explicitly contain the generator's internal overhead measurements. The algorithm complexity described in Section 6.6 supports the concerns. Therefore, the modified approach out-performs the original approach in this area, as well.

Chapter 9

Conclusion

While this thesis mentions some tools usable with either/both single machine and multiple node systems. The thesis discussed the selection of the *hashcat* as the only tool used for benchmarking. The main reasons for selecting the *hashcat* are its large supported hash-mode pool and its ability to use accelerators via *OpenCL* kernel. The thesis then follows by a description of FITcrack system, its architecture with a brief look at the underlying *BOINC* framework. The thesis follows by selecting viable *GPU* accelerators and benchmarks of their behaviour under different attack types, hash-modes, and several used cards. Later at the end of its first half, the thesis summarizes and discusses some of the benchmark results.

Thesis' second half follows by presenting the recommendations and attack strategies so crucial for efficient *password recovery*. In the next chapter, it then transforms the theory into specific features and system modifications. The implementation of all new features and modifications follows with its separated chapter. The chapter discusses the challenges and the impact on the system modules. Finally, the last chapter compares the original and modified system versions with a focus on the proposed changes.

Contribution

The process of obtaining selected *GPUs* is rather complicated, as the variety of widely available and powerful model versions is somewhat limited. The task has not become any easier even after the release of Nvidia's new card series. Since the new cards have been considered overpriced [44] for the power gain, they provide. Also, some cards — mostly the top models from *AMD* — are hard to obtain since the gamers have no interest in them or manufacturer releases an only limited amount of the cards around their release, as in February 2019 with Radeon VII [6]. It results in only a minimal amount available in stores, which then define a per customer quota.

Nevertheless, the data from obtained cards show that the Nvidia GeForce RTX 2080Ti substantially outperforms the Nvidia GeForce GTX 1080 Ti while also consumes less power. Also, the measurements show a performance similarity of the AMD Radeon RX 580 8G and the Nvidia GeForce GTX 1060 6G while both consume a similar amount of electric power throughout the benchmarks. Moreover, the same data neglect the possibility of building a higher amount of less powerful computation nodes, as introduced in Chapter 3. The data show that impossible with keeping a least the same performance and power costs without increasing the initial costs. The extensiveness of the measurements makes up for an in-

dependent paper, which was already accepted and published at the student's conference Excel@FIT 2019 [46].

The recommendations and strategies are not easy to find as most of the experts in this field work in companies or *LEA* protecting their know-how. However, there are companies like NETMUX releasing at least some password recovery manuals [32]. Nevertheless, Chapter 5 presents all gathered information regarding the topic and presents them in the form of lists.

The proposed task prioritization methodology builds on the information provided by NETMUX's manual [32] and its summarization from Chapter 5. Chapter 6 presents the FITcrack's shortcomings identified throughout the personal usage of the system. Moreover, the proposed changes also reflect the author's experiences with the FITcrack from its development throughout the participation on the *I*ntegrated platform for analysis of digital data from security incidents research project[1] conducted on the faculty.

Finally, the last comparison and benchmarks show-case the relevance of the modifications as well as their impact on various recovery tasks. The change to the distribution of dictionaries reduces the time required for the finishing of the task by 18 to 55 % according to Tables 8.2 and 8.3.

Future work

The future topics of interest may be:

- Comparison of the node with the described configuration against configurations using server motherboards.
- Analysis of how the PCI-e bandwidth impacts the password recovery in both the single and multi-*GPU* system and how that changes based on the attack type.
- Analysis of how specific password modification rules and size of their sets impact dictionary attack recovery speeds.
- Further analysis of proposed task prioritization and search for other methodologies.
- Replacement of *BOINC* subsystem by some other, which would reduce the communication overheads in deployment on a cluster.

[1] http://www.fit.vutbr.cz/units/UIFS/grants/index.php.en?id=1063

Bibliography

[1] Anderson, D. P.: Boinc: A system for public-resource computing and storage. In *Grid Computing, 2004. Proceedings. Fifth IEEE/ACM International Workshop on*. IEEE. 2004. pp. 4–10.

[2] buildcomputers.net: Typical Power Consumption of PC Components - Power Draw in Watts.
http://www.buildcomputers.net/power-consumption-of-pc-components.html. [Online]. Accessed: 15.1.2019.

[3] Chou, H.-C.; Lee, H.-C.; Yu, H.-J.; et al.: Password cracking based on learned patterns from disclosed passwords. *IJICIC*. vol. 9, no. 2. 2013: pp. 821–839.

[4] corp., N.: Graphics Cards, Gaming, Laptops, and Virtual Reality from NVIDIA GeForce. https://www.nvidia.com/en-us/geforce/. 2019. [Online]. Accessed: 26.01.2019.

[5] Corporation, V.: Steam Hardware & Software Survey.
https://store.steampowered.com/hwsurvey/. December 2018. [Online]. Accessed: 14.1.2019.

[6] CrinsomRayne: AMD Vega based Radeon VII For Consumers and Gamers | Exclusive RedGamingTech. http://www.redgamingtech.com/amd-vega-based-radeon-vii-for-consumers-and-gamers-exclusive/. December 2018. [Online]. Accessed: 17.4.2019).

[7] Dally, W. J.; Nickolls, J.: The GPU Computing Era. *IEEE Micro*. vol. 30. 04 2010: pp. 56–69. ISSN 0272-1732. doi:10.1109/MM.2010.41.

[8] Damaraju, S. R.; Walters, J. D.; O'connor, D. S.: Counter rotating blower with individual controllable fan speeds. November 13 2014. US Patent App. 13/889,172.

[9] Das, R.: Top Pentesting Tools.
https://resources.infosecinstitute.com/category/certifications-training/pentesting-certifications/top-pentesting-tools/. [Online]. Accessed: 14.01.2019.

[10] Dean, R. P.: Heat sink device having radial heat and airflow paths. August 18 1998. US Patent 5,794,685.

[11] Designer, S.: John the Ripper password cracker. https://openwall.com/john/. [Online]. Accessed: 10.01.2019.

[12] Designer, S.: John the Ripper documentation.
https://www.openwall.com/john/doc/. 2015. [Online]. Accessed: 14.01.2019.

[13] Dürmuth, M.; Kranz, T.: On Password Guessing with GPUs and FPGAs. In *Technology and Practice of Passwords*, edited by S. F. Mjølsnes. Cham: Springer International Publishing. 2015. ISBN 978-3-319-24192-0. pp. 19–38.

[14] Fore1gn: Video Card Coolers: Blower vs Open-Air vs AIO - Logical Increments Blog. http://blog.logicalincrements.com/2017/07/video-card-coolers-blower-vs-open-air-vs-aio/. July 2017. [Online]. Accessed: 13.1.2019.

[15] Freepik: Boss - Free people icons. https://www.flaticon.com/free-icon/boss_265674#term=administrator&page=1&position=3. [Online]. Accessed: 13.05.2019.

[16] Hagedoorn, H.: AMD Radeon RX 570 And RX 580 GPUs Sold Out Due Cryptocurrency Mining. https://www.guru3d.com/news-story/amd-rx-570-and-rx-580-gpus-sold-out-due-cryptocurrency-mining.html. June 2017. [Online]. Accessed: 13.1.2019.

[17] Hamalčíková, K.: Cena elektřiny za kWh v roce 2018 poskočila na 4,1 Kč. Proč koukat i na jiné částky? https://www.elektrina.cz/cena-elektriny-za-kwh-2018-cez-eon-pre-a-jini-dodavatele-elektriny. March 2018. [Online]. Accessed: 05.01.2019.

[18] Han, T.-S. A.: Graphics card apparatus with improved heat dissipating mechanisms. January 22 2008. US Patent 7,321,494.

[19] Hong, S.; Kim, H.: An Integrated GPU Power and Performance Model. *SIGARCH Comput. Archit. News*. vol. 38, no. 3. June 2010: pp. 280–289. ISSN 0163-5964. doi:10.1145/1816038.1815998.

[20] Hranický, R.; Holkovič, M.; Matoušek, P.; et al.: On Efficiency of Distributed Password Recovery. *The Journal of Digital Forensics, Security and Law*. vol. 11, no. 2. 2016: pp. 79–96. ISSN 1558-7215.

[21] Hranický, R.; Matoušek, P.; Ryšavý, O.; et al.: Experimental Evaluation of Password Recovery in Encrypted Documents. In *Proceedings of ICISSP 2016*. SciTePress - Science and Technology Publications. 2016. ISBN 978-989-758-167-0. pp. 299–306. doi:10.5220/0005685802990306.

[22] Hranický, R.; Zobal, L.; Večeřa, V.; et al.: Distributed Password Cracking in a Hybrid Environment. In *Proceedings of SPI 2017*. University of Defence in Brno. 2017. ISBN 978-80-7231-414-0. pp. 75–90.

[23] Hranický, R.; Zobal, L.; Večeřa, V.; et al.: The architecture of Fitcrack distributed password cracking system. Technical report. Faculty of Information Technology BUT. 2018.

[24] Incorporation, T.: Conformant Products - The Khronos Group Inc.
https://www.khronos.org/conformance/adopters/conformant-products/opencl. [Online]. Accessed: 10.1.2019.

[25] Jiao, Y.; Lin, H.; Balaji, P.; et al.: Power and Performance Characterization of Computational Kernels on the GPU. In *Proceedings of the 2010 IEEE/ACM Int'L Conference on Green Computing and Communications & Int'L Conference on Cyber, Physical and Social Computing*. GREENCOM-CPSCOM '10. Washington, DC, USA: IEEE Computer Society. 2010. ISBN 978-0-7695-4331-4. pp. 221–228. doi:10.1109/GreenCom-CPSCom.2010.143.

[26] Kranz, T.: *GPU-Assisted Password Hashing: The Example of scrypt*. Master's Thesis. Ruhr-Universität Bochum. 2014.

[27] Kumar, H.; Kumar, S.; Joseph, R.; et al.: Rainbow table to crack password using MD5 hashing algorithm. In *2013 IEEE Conference on Information Communication Technologies*. April 2013. pp. 433–439. doi:10.1109/CICT.2013.6558135.

[28] Kurzy.cz; AliaWeb: Kurzy historie, kurzovní lístek NB 12.12.2018, historie kurzů měn. https://www.kurzy.cz/kurzy-men/historie/ceska-narodni-banka/D-12.12.2018/. December 2018. [Online]. Accessed: 12.12.2019.

[29] Leonidas: Die Grafikchip- und Grafikkarten-Marktanteile im dritten Quartal 2018 | 3DCenter.org. https://www.3dcenter.org/news/die-grafikchip-und-grafikkarten-marktanteile-im-dritten-quartal-2018. November 2018. [Online]. Accessed: 14.1.2019.

[30] Lim, R.: Parallelization of John the Ripper (JtR) using MPI. *Nebraska: University of Nebraska*. 2004.

[31] Narayanan, A.; Shmatikov, V.: Fast Dictionary Attacks on Passwords Using Time-space Tradeoff. In *Proceedings of the 12th ACM Conference on Computer and Communications Security*. CCS '05. New York, NY, USA: ACM. 2005. ISBN 1-59593-226-7. pp. 364–372. doi:10.1145/1102120.1102168.

[32] Netmux LLC: *Hash Crack: Password Cracking Manual*. Independently published. third edition. 2019. ISBN 1793458618.

[33] Neuman, B. C.: Scale in Distributed Systems. In *Readings in Distributed Computing Systems*. IEEE Computer Society Press. 1994. pp. 463–489.

[34] Niewiadomska-Szynkiewicz, E.; Marks, M.; Jantura, J.; et al.: A Hybrid CPU/GPU Cluster for Encryption and Decryption of Large Amounts of Data. *Journal of Telecommunications and Information Technology*. vol. nr 3. 2012: pp. 32–39.

[35] Northcutt, S.: Hash Functions. https://www.sans.edu/cyber-research/security-laboratory/article/hash-functions. 2019. [Online]. Accessed: 27.01.2019.

[36] Oechslin, P.: The ophcrack password cracker. http://ophcrack.sourceforge.net/. [Online]. Accessed: 10.01.2019.

[37] O'Gorman, L.: Comparing passwords, tokens, and biometrics for user authentication. *Proceedings of the IEEE*. vol. 91, no. 12. Dec 2003: pp. 2021–2040. ISSN 0018-9219. doi:10.1109/JPROC.2003.819611.

[38] Ower, E.; Pankhurst, R.: *The Measurement of Air Flow (Fifth Edition)*. Pergamon. fifth edition edition. 1977. ISBN 978-0-08-021281-4. 6 - 21 and 112 - 147 pp.. doi:https://doi.org/10.1016/B978-0-08-021281-4.50008-7.

[39] Percival, C.: Stronger key derivation via sequential memory-hard functions. *Self-published*. 2009: pp. 1–16.

[40] Provos, N.; Mazieres, D.: A Future-Adaptable Password Scheme. In *USENIX Annual Technical Conference, FREENIX Track*. 1999. pp. 81–91.

[41] Shunzhi, H.; Yujan, M.: Double-fan heat-dissipating device. May 2015. CN102137581B.

[42] Software, P.: Video Card (GPU) Benchmarks - High End Video Cards. https://www.videocardbenchmark.net/high_end_gpus.html. January 2019. [Online]. Accessed: 14.1.2019.

[43] Stefanoski, Z.: Cooling system for computer hardware. June 3 2008. US Patent 7,382,616.

[44] Steube, J.: Hashcat advanced password recovery. https://hashcat.net/hashcat/. [Online]. Accessed: 06.01.2019.

[45] Tarditi, D.; Puri, S.; Oglesby, J.: Accelerator: Using Data Parallelism to Program GPUs for General-purpose Uses. *SIGOPS Oper. Syst. Rev.*. vol. 40, no. 5. October 2006: pp. 325–335. ISSN 0163-5980. doi:10.1145/1168917.1168898.

[46] Večeřa, V.: Selecting the right GPU for password recovery. In *Proceedings of Excel@FIT 2019*. Faculty of Information Technology, Brno University of Technology. 2019.

[47] Yan, J.; Blackwell, A.; Anderson, R.; et al.: Password memorability and security: empirical results. *IEEE Security Privacy*. vol. 2, no. 5. Sep. 2004: pp. 25–31. ISSN 1540-7993. doi:10.1109/MSP.2004.81.

[48] Zobal, L.: *Distribuovaná obnova hesel s využitím nástroje hashcat*. Master's Thesis. Vysoké učení technické v Brně, Fakulta informačních technologií. 2018.

Appendix A

Content of CD

- *dictionary_processor/* – directory with sources of the fragmentation tool.
- *gpu_benchmarks/* – directory with scripts and all GPU related results.
  - *post_processing/* – directory with scripts converting results to CSVs.
  - *power_reading/* – directory with MQTT broker, client, and Sonoff POW firmware.
  - *results/* – directory with the GPU measurement results.
  - *hashcat-5.0.0.7z* – archive with precompiled *hashcat* used for benchmarking.
  - *how_to_measure.dm* – measurement guide written in Markdown.
- *masters_thesis-print.pdf* – rendered text of thesis with disabled links.
- *masters_thesis-wis.pdf* – rendered text of thesis with active links.
- *server_sources/* – directory with FITcrack module's source codes.
  - *boinc_server/* – directory with FITcrack server source files.
  - *runner/* – directory with FITcrack client application.
  - *web_backend/* – directory with FITcrack backend source files.
  - *web_frontend/* – directory with FITcrack frontend source files.
- *text/* – directory with the LaTeX source files and figure of the thesis.

Appendix B

Comparison of dictionary and mask attack speeds

B.1 SHA-1

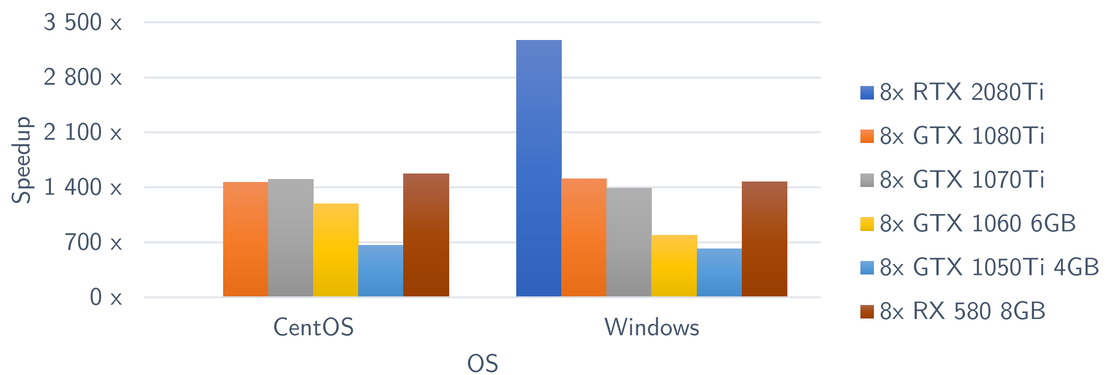

Figure B.1: Speed gain of mask attack compared to dictionary attack targeting *SHA1*.

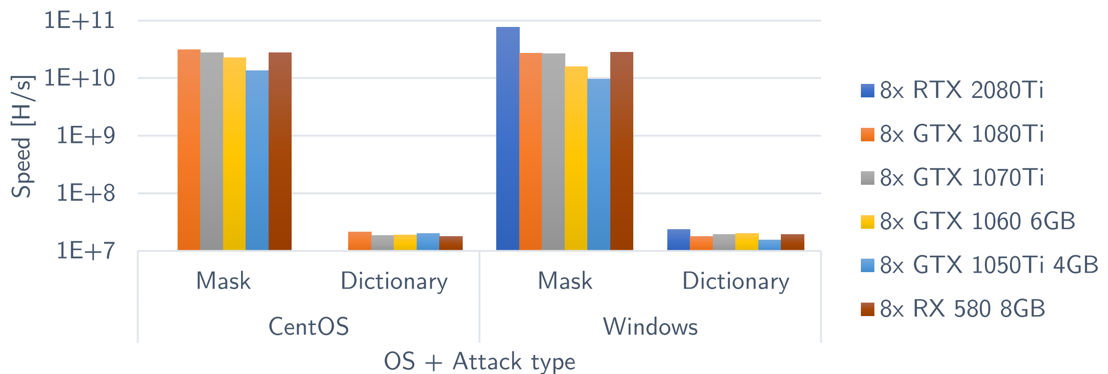

Figure B.2: Average speeds of mask and dictionary attacks on eight GPUs targeting *SHA1*.

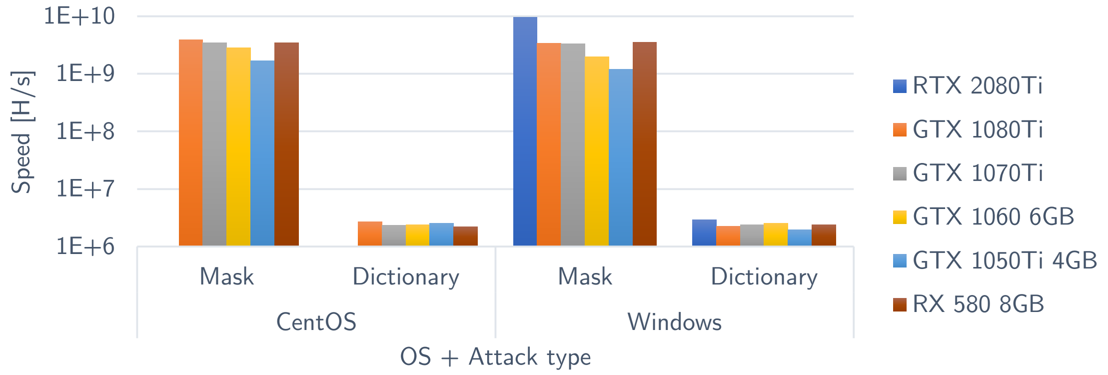

Figure B.3: Average speeds of mask and dictionary attacks on single GPU targeting *SHA1*.

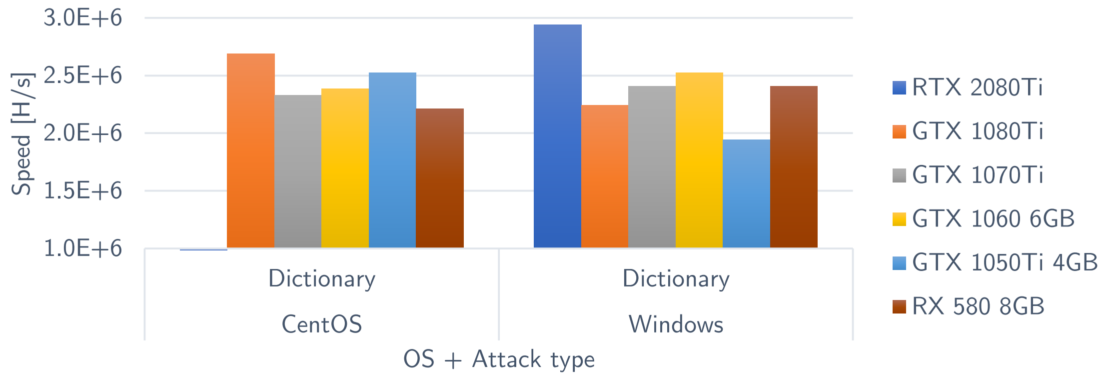

Figure B.4: Detail of dictionary attack average speeds on single GPU targeting *SHA1*.

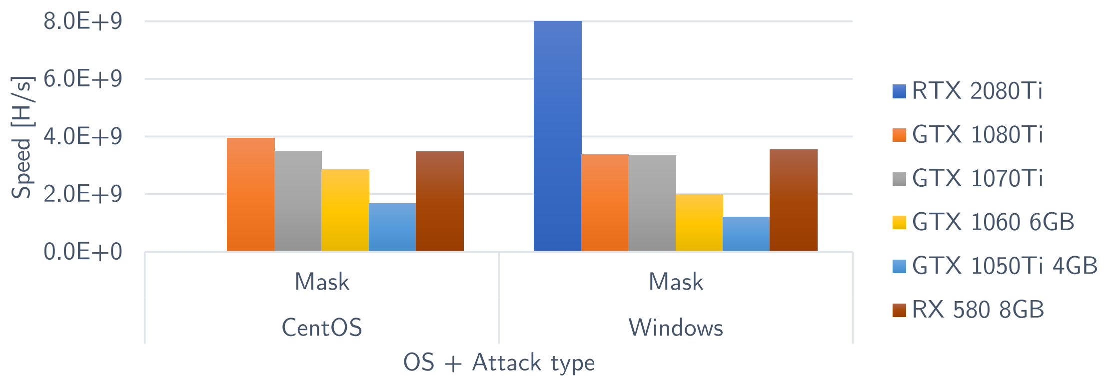

Figure B.5: Detail of mask attack average speeds on single GPU targeting *SHA1*.

B.2 SHA-256

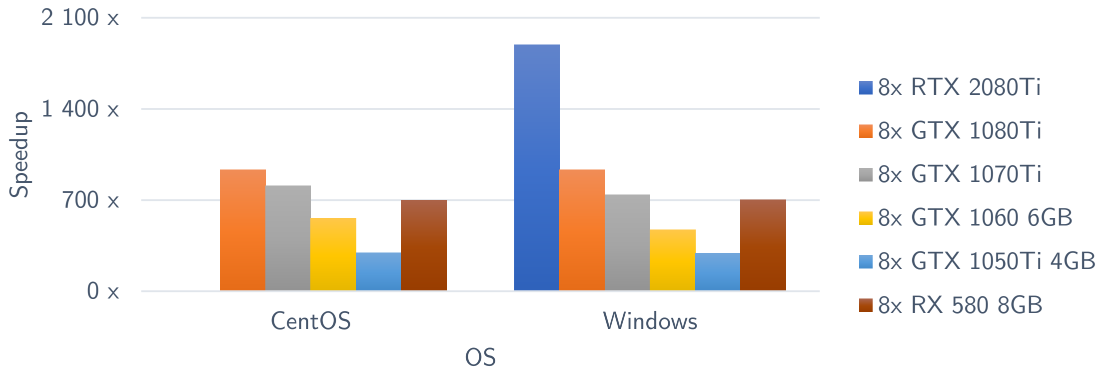

Figure B.6: Speed gain of mask attack compared to dictionary attack targeting *SHA256*.

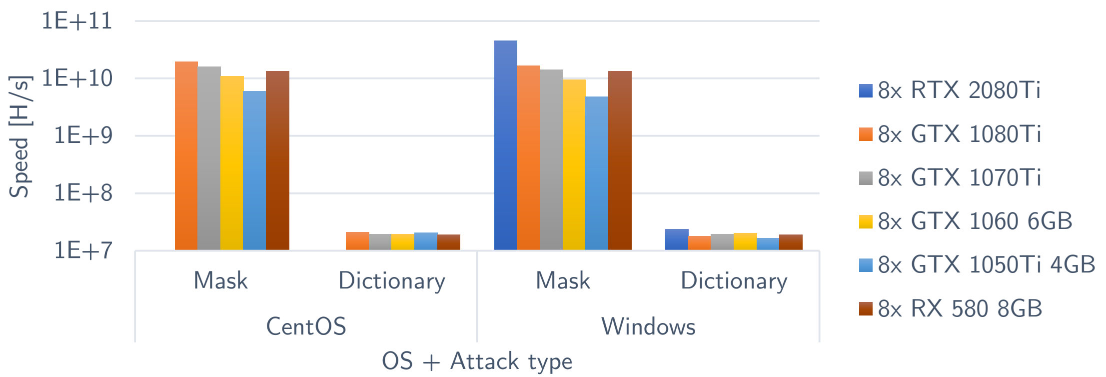

Figure B.7: Average speeds of mask and dictionary attacks on eight GPUs targeting *SHA256*.

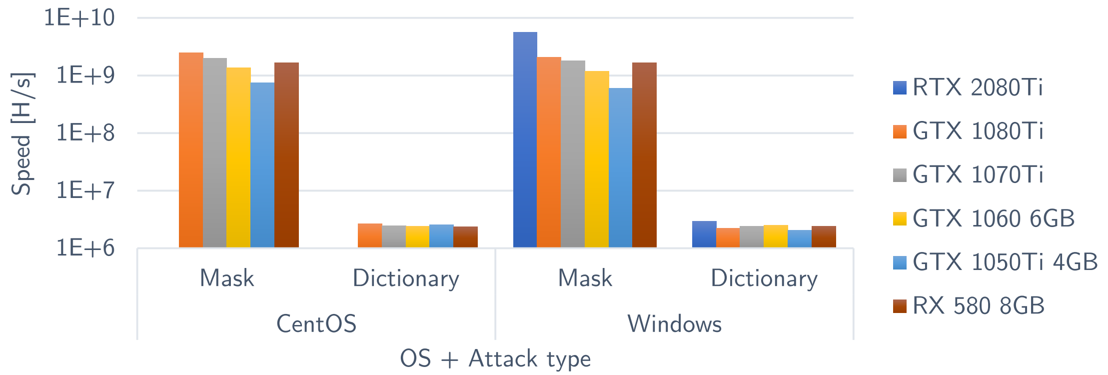

Figure B.8: Average speeds of mask and dictionary attacks on single GPU targeting *SHA256*.

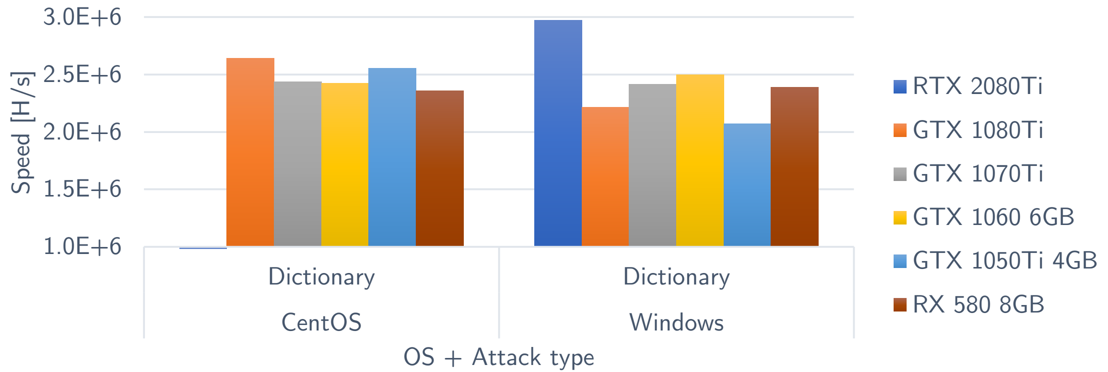

Figure B.9: Detail of dictionary attack average speeds on single GPU targeting *SHA256*.

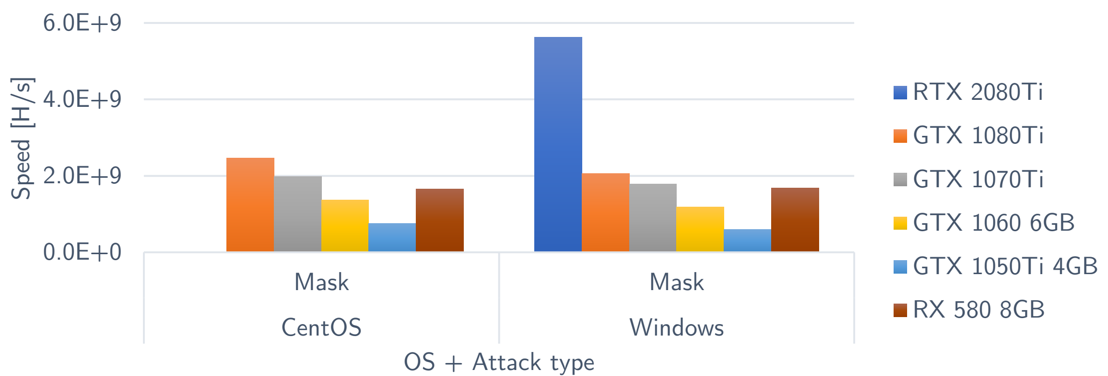

Figure B.10: Detail of mask attack average speeds on single GPU targeting *SHA256*.

B.3 Bcrypt

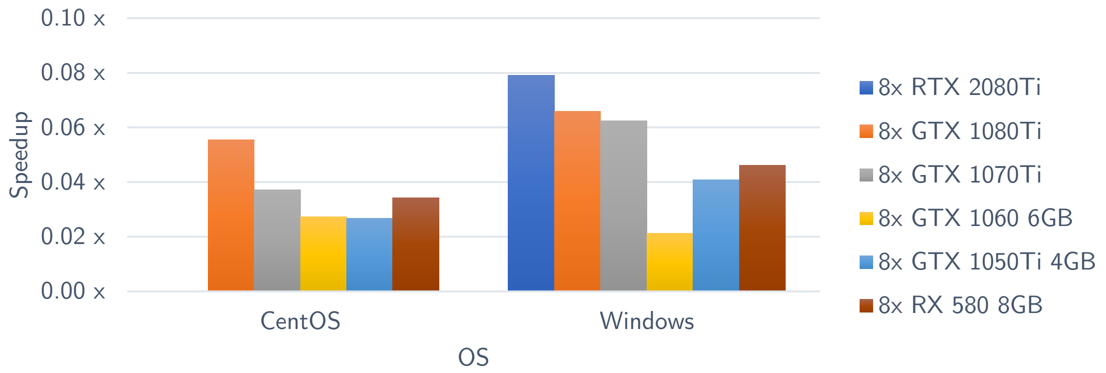

Figure B.11: Speed gain of mask attack compared to dictionary attack targeting *bcrypt*.

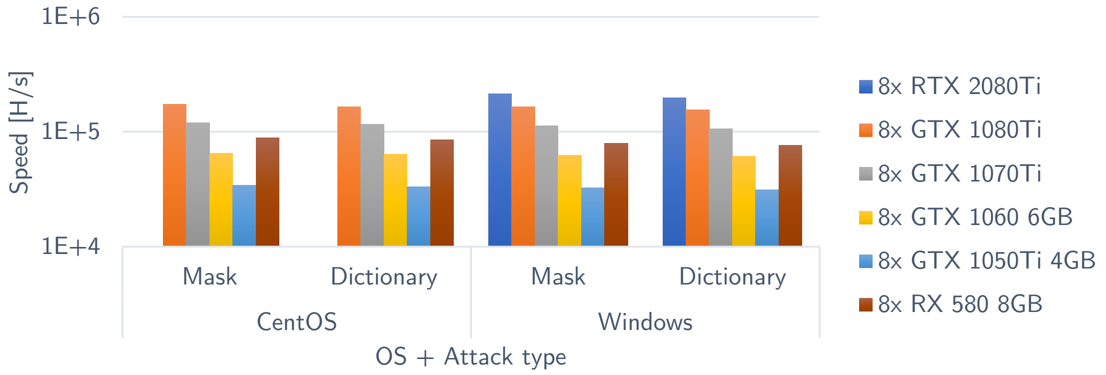

Figure B.12: Average speeds of mask and dictionary attacks on eight GPUs targeting *bcrypt*.

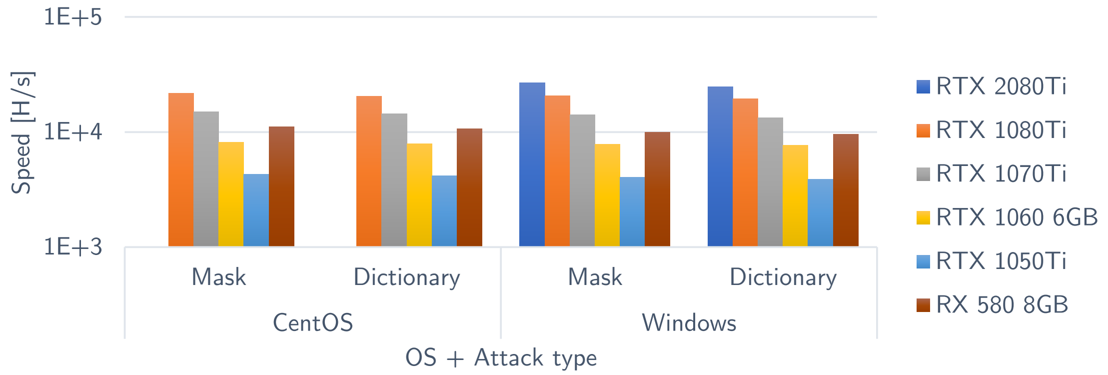

Figure B.13: Average speeds of mask and dictionary attacks on single GPU targeting *bcrypt*.

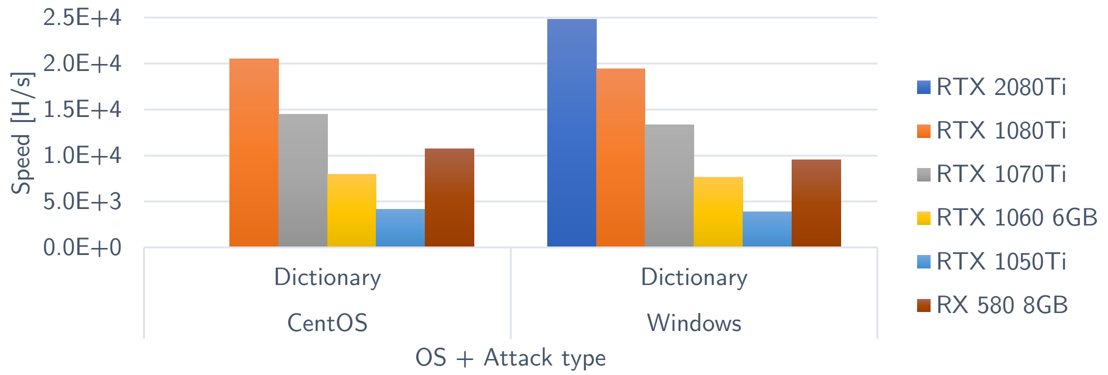

Figure B.14: Detail of dictionary attack average speeds on single GPU targeting *bcrypt*.

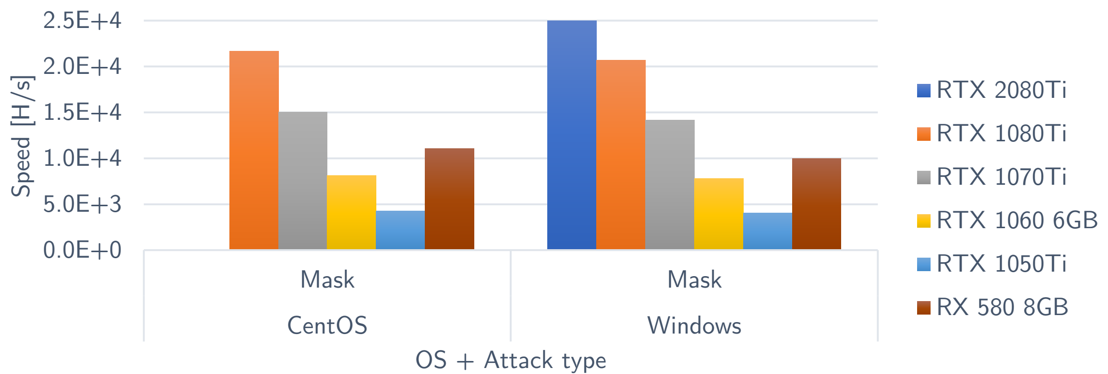

Figure B.15: Detail of mask attack average speeds on single GPU targeting *bcrypt*.

B.4 Scrypt

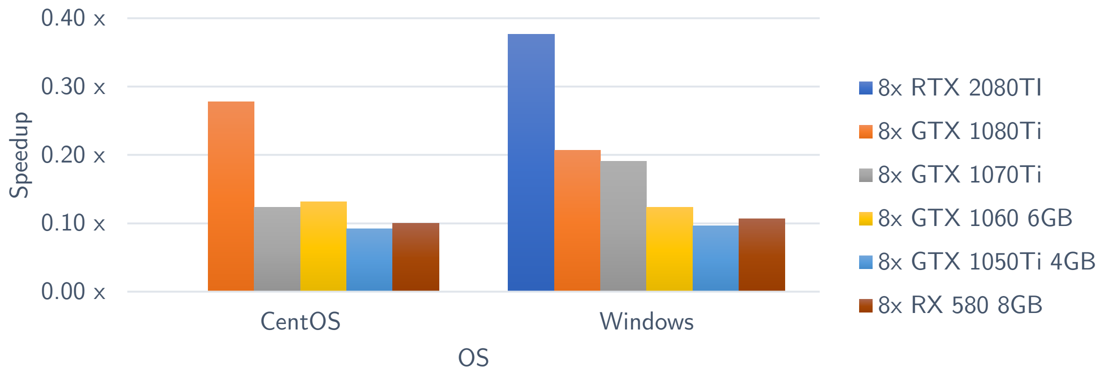

Figure B.16: Speed gain of mask attack compared to dictionary attack targeting *scrypt*.

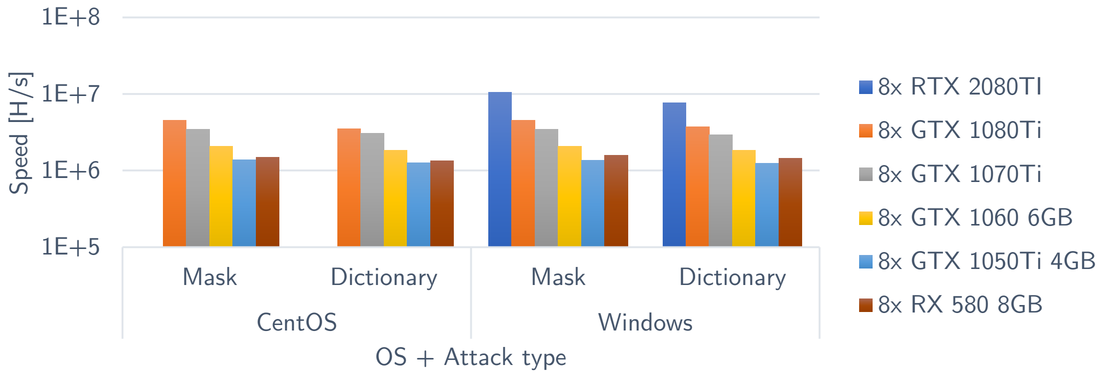

Figure B.17: Average speeds of mask and dictionary attacks on eight GPUs targeting *scrypt*.

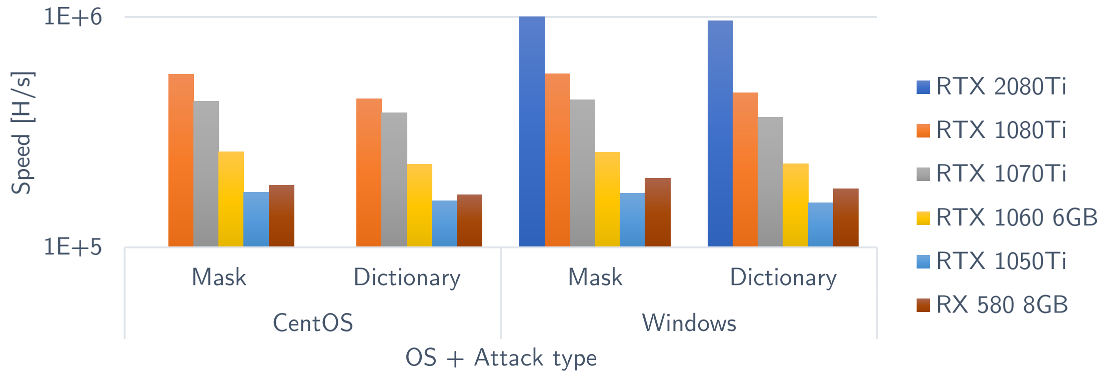

Figure B.18: Average speeds of mask and dictionary attacks on single GPU targeting *scrypt*.

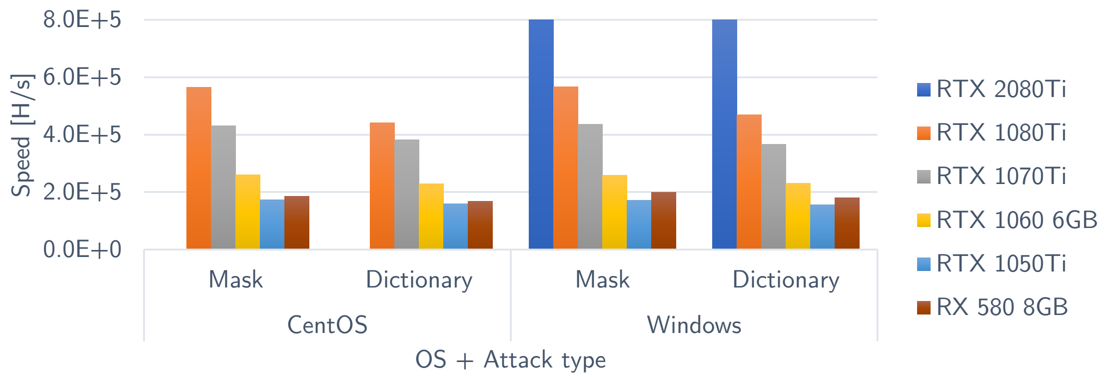

Figure B.19: Detail of dictionary attack average speeds on single GPU targeting *scrypt*.

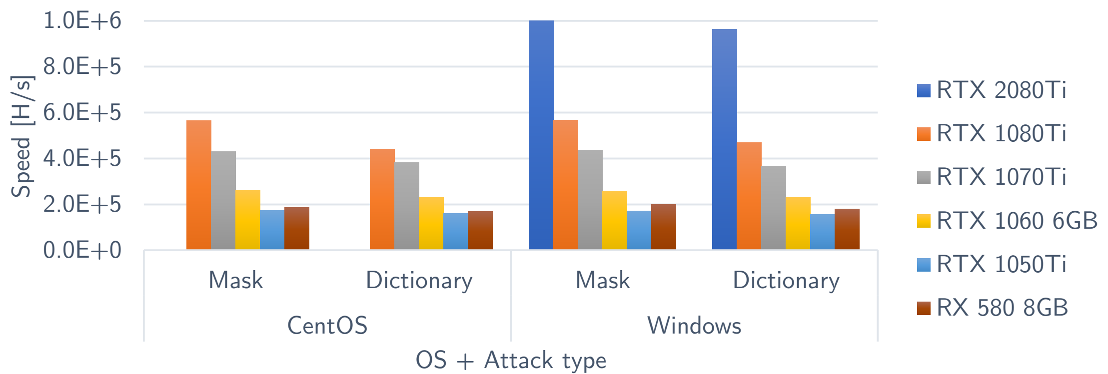

Figure B.20: Detail of mask attack average speeds on single GPU targeting *scrypt*.

B.5 RAR5

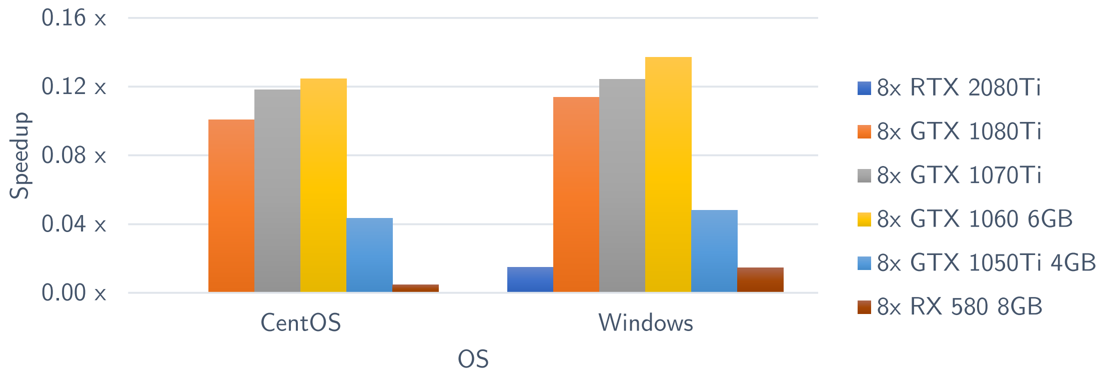

Figure B.21: Speed gain of mask attack compared to dictionary attack targeting *RAR5*.

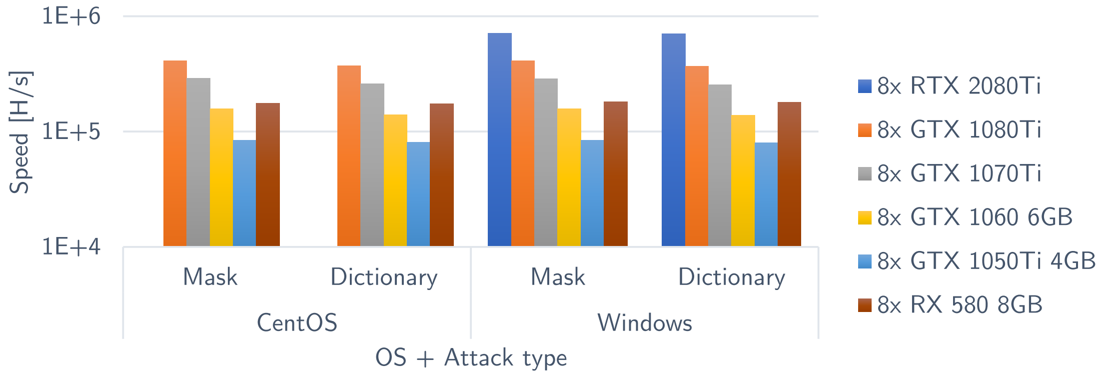

Figure B.22: Average speeds of mask and dictionary attacks on eight GPUs targeting *RAR5*.

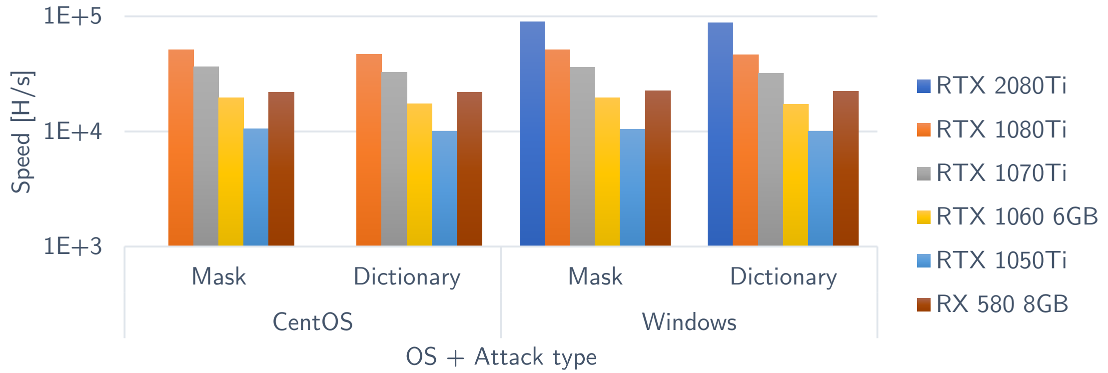

Figure B.23: Average speeds of mask and dictionary attacks on single GPU targeting *RAR5*.

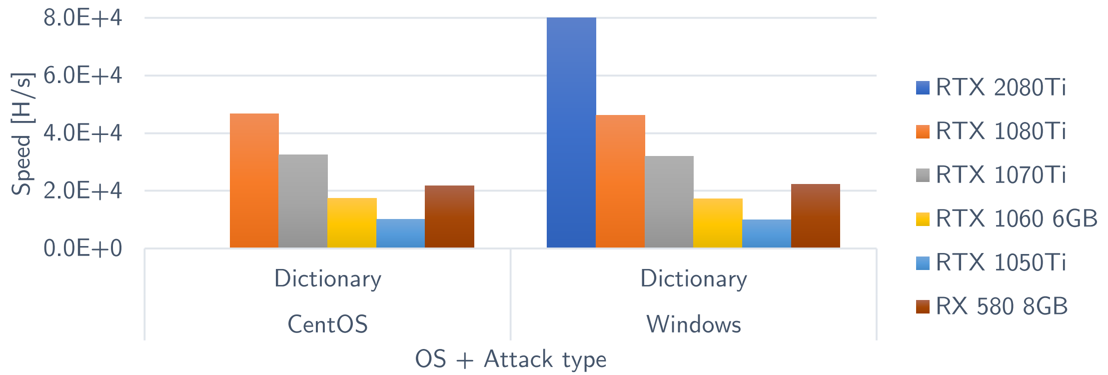

Figure B.24: Detail of dictionary attack average speeds on single GPU targeting *RAR5*.

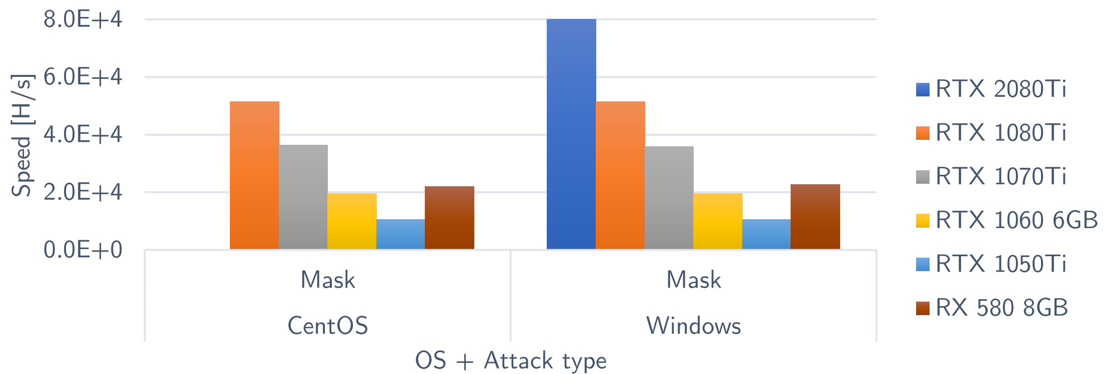

Figure B.25: Detail of mask attack average speeds on single GPU targeting *RAR5*.

Appendix C

Comparison of power consumptions

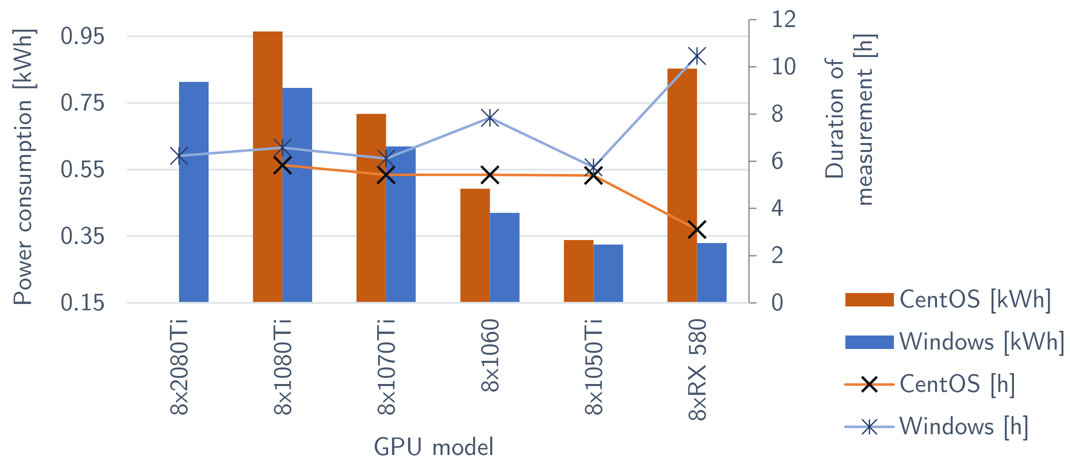

Figure C.1: Average power drain per one hour of measurements with effect of the length of measurement's duration.

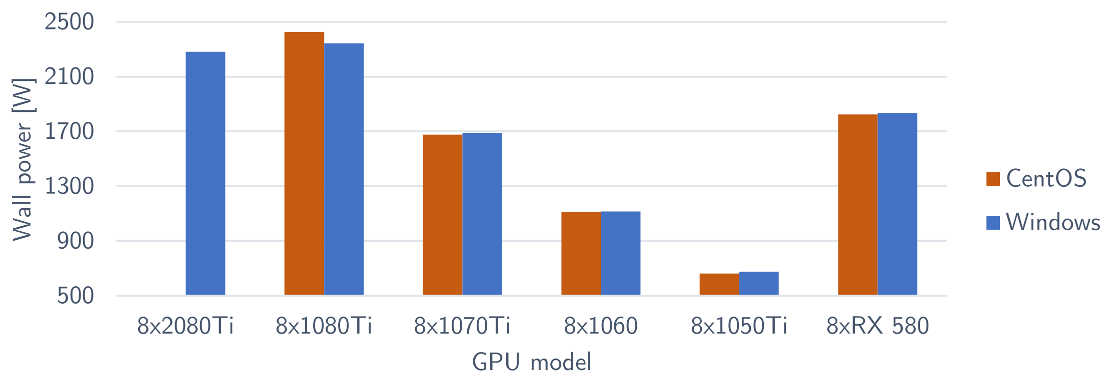

Figure C.2: Minimal power drain during measurements.

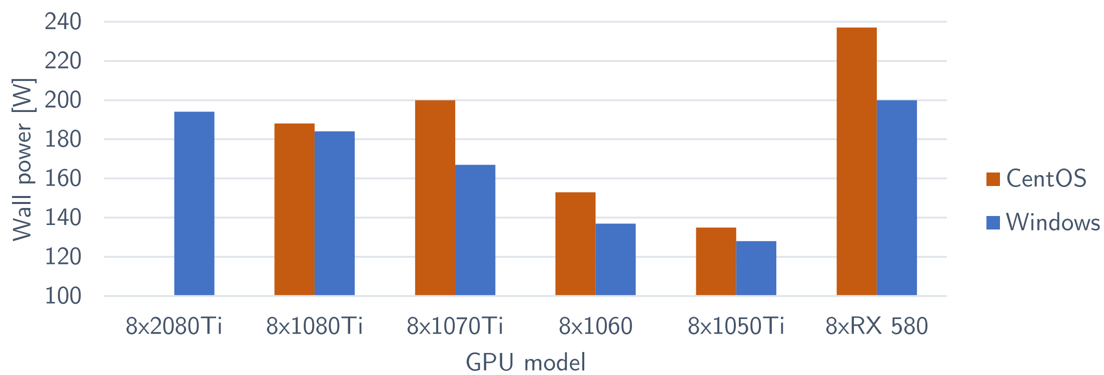

Figure C.3: Maximal power drain during measurements.

Appendix D

Long-term cost effectiveness of node configurations

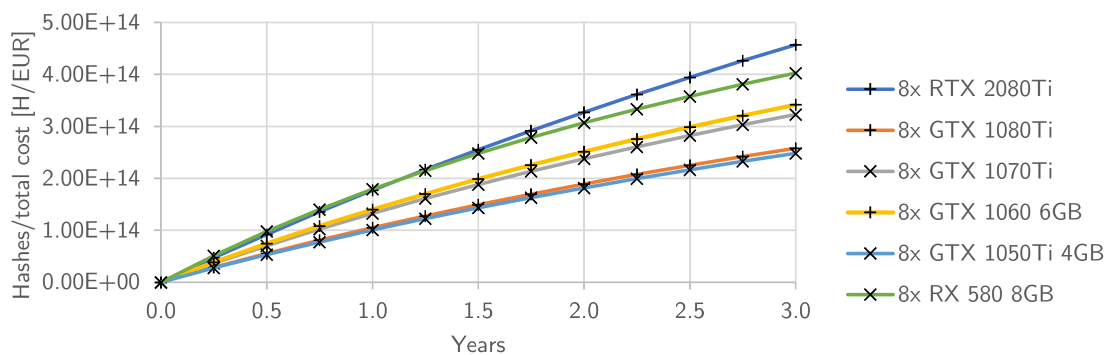

Figure D.1: Long-term cost effectiveness chart showing computed hashes per one euro of the total costs (purchase costs + power consumption costs) when targeting hash-mode *SHA1*.

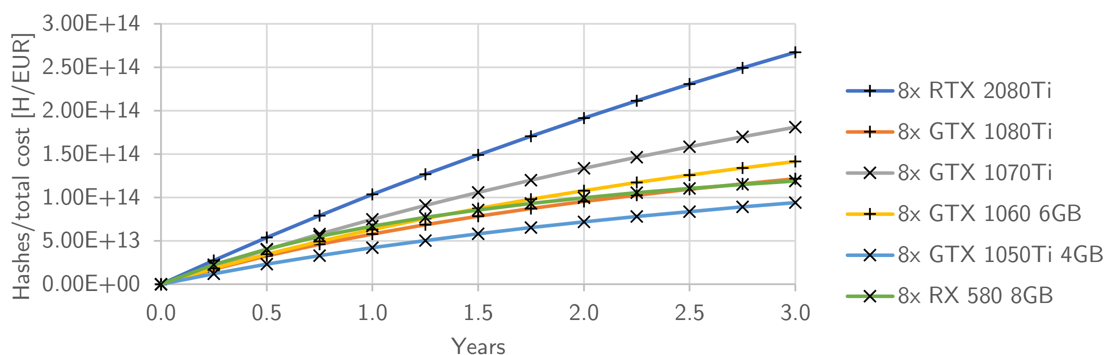

Figure D.2: Long-term cost effectiveness chart showing computed hashes per one euro of the total costs (purchase costs + power consumption costs) when targeting hash-mode *SHA256*.

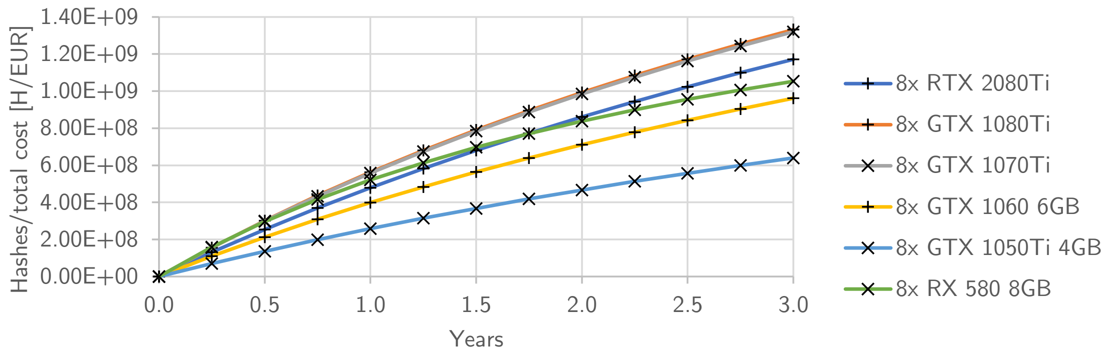

Figure D.3: Long-term cost effectiveness chart showing computed hashes per one euro of the total costs (purchase costs + power consumption costs) when targeting hash-mode *bcrypt*.

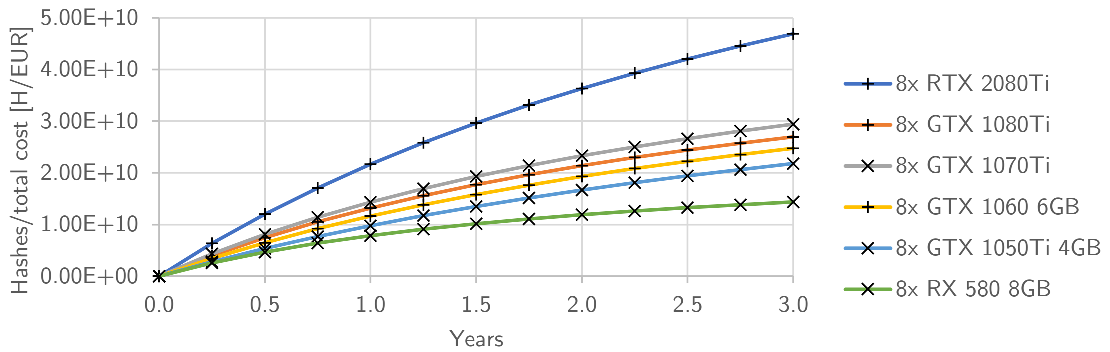

Figure D.4: Long-term cost effectiveness chart showing computed hashes per one euro of the total costs (purchase costs + power consumption costs) when targeting hash-mode *scrypt*.

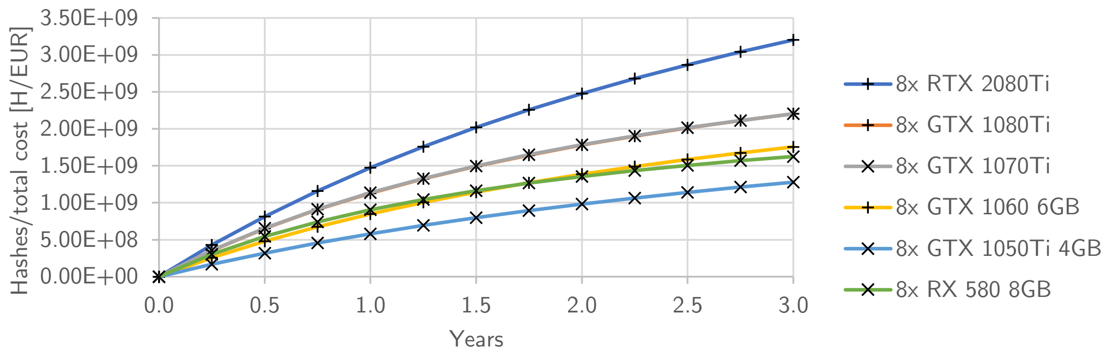

Figure D.5: Long-term cost effectiveness chart showing computed hashes per one euro of the total costs (purchase costs + power consumption costs) when targeting hash-mode *RAR5*.