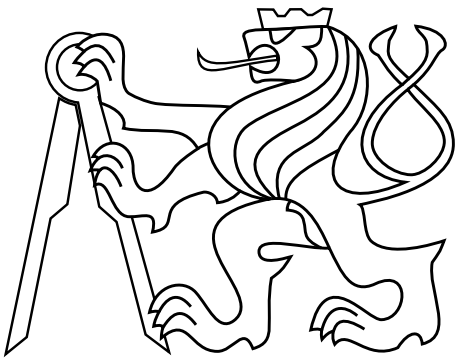
Master Thesis

# Password recovery job scheduling for online deep file analysis

Bc. Petr Kubelka
Supervisor: Mgr. Lucie Mohelníková

May 2020



Department of Computer Science
Faculty of Electrical Engineering
Czech Technical University in Prague

# MASTER'S THESIS ASSIGNMENT

## I. Personal and study details

Student's name: **Kubelka Petr** Personal ID number: **457022**

Faculty / Institute: **Faculty of Electrical Engineering**

Department / Institute: **Department of Computer Science**

Study program: **Open Informatics**

Specialisation: **Software Engineering**

## II. Master's thesis details

Master's thesis title in English:

**Password recovery job scheduling for online deep file analysis**

Master's thesis title in Czech:

**Online rozvrhování úkolů pro obnovu hesel**

Guidelines:

Many malicious files are spread via password-protected documents or compressed archives that can be, for example, sent via email by attackers. Password protection prevents these files from being analyzed for the presence of malicious code. In order to enable detection of this malicious code, the password protecting the file must be recovered. The password recovery must be done for a stream of incoming files in a timely fashion. Thus, an appropriate scheduling method has to be deployed to achieve the best performance. The main goal of the thesis is to (1) formalize the scheduling problem for this task, (2) review the existing approaches that are similar to the problem, (3) select and implement the chosen scheduling method, (4) experimentally analyze the advantages and disadvantages of the chosen method in a setting corresponding to a real-world application (number of files, distribution over time).

Bibliography / sources:

[1] Hranický R, Zobal L, Ryšavý O, Kolář D. Distributed password cracking with BOINC and hashcat. Digital Investigation. 2019 Sep 1;30:161-72.
[2] Pinedo M. Scheduling. New York: Springer; 2012.

Name and workplace of master's thesis supervisor:

**Mgr. Lucie Mohelníková, Department of Computer Science, FEE**

Name and workplace of second master's thesis supervisor or consultant:

Date of master's thesis assignment: **14.02.2020** Deadline for master's thesis submission: **22.05.2020**

Assignment valid until: **30.09.2021**

| Mgr. Lucie Mohelníková<br>Supervisor's signature | Head of department's signature | prof. Mgr. Petr Páta, Ph.D.<br>Dean's signature |
|---|---|---|

## III. Assignment receipt

The student acknowledges that the master's thesis is an individual work. The student must produce his thesis without the assistance of others, with the exception of provided consultations. Within the master's thesis, the author must state the names of consultants and include a list of references.

| Date of assignment receipt | Student's signature |
|---|---|

**Abstract**

Many malicious files are spread via password-protected documents or archives that can be, for example, sent via e-mail by attackers. Password protection prevents these files from being analyzed for the presence of malicious code. To enable the detection of malicious code, the password, which is protecting the file or archive must be recovered. The password recovery for the stream of incoming files must be completed in a timely fashion. To achieve the best performance, an appropriate scheduling method has to be deployed. Thanks to the rise of the computational power of graphics cards and the password cracking programs supporting them, password cracking became once again faster, thus enabling us to crack passwords in a reasonable time and for a broad spectrum of encrypting formats.

**Keywords:** Password cracking, hashing, Hashcat, John the Ripper, software, scheduling algorithms, Docker.

**Abstrakt**

Mnoho infikovaných souborů je šířeno pomocí heslem chráněných dokumentů nebo archívů, které mohou být poslány útočníkem přes e-mail. Tím, že jsou tyto soubory chráněny heslem, není možné, aby byl jejich obsah analyzován, zda obsahuje škodlivý kód. Abychom byli schopní tento škodlivý kód detekovat, musíme získat heslo chránící daný soubor. Obnovování hesel musí být provedeno v rozumném čase pro proud příchozích souborů, a tak musí být zvolen patřičný rozhodovací algoritmus pro dosažení nejlepšího výkonu. Vzhledem ke zvýšení výpočetního výkonu grafických karet a existenci programů obnovující hesla, které podporují grafické karty, se obnova hesel opět stala jednodušší. Díky tomu máme možnost obnovovat hesla v rozumném čase a pro více typů šifrovacích algoritmů.

**Klíčová slova:** Obnovování hesel, hašování, Hashcat, John the Ripper, software, plánovací algoritmy, Docker.

# Author statement for graduate thesis:

I declare that the presented work was developed independently and that I have listed all sources of information used within it in accordance with the methodical instructions for observing the ethical principles in the preparation of university theses.

Prague, date ..... .....

signature

# Acknowledgements

I want to thank my supervisor Mgr. Lucie Mohelníková for immense support, time and advice and to my girlfriend Zdenka for proof-reading and support, my friends Tomáš, Petr, Daniel and Anita for proof-reading and spell-checking and finally to my family for helping me with all encountered obstacles.

# Contents


**1 Introduction** **13**
1.1 Introduction . . . 13

**2 Related work** **15**
2.1 First Come First Served (FCFS) . . . 15
2.2 Shortest Job First (SJF) . . . 15
2.3 Round-Robin (RR) . . . 15
2.4 The List Scheduling (LIST) algorithm . . . 16
2.5 Min-Min and Max-Min . . . 16
2.6 Online scheduling . . . 16
2.7 Stochastic scheduling . . . 17
2.8 Robust optimization . . . 17

**3 Related password cracking engines** **19**
3.1 Hashcat . . . 19
3.1.1 Hashtopus . . . 19
3.1.2 Hashtopolis . . . 19
3.1.3 Fitcrack . . . 19
3.1.4 Terahash . . . 20
3.2 John the Ripper . . . 20
3.2.1 John the Ripper MPI and OpenMPI . . . 20

**4 Introduction to password cracking** **21**
4.1 Terminology . . . 21
4.2 Historical context . . . 21
4.3 Encryption . . . 22
4.3.1 Rivest Cipher (RC4) . . . 22
4.3.2 Advanced Encryption Standard (AES) . . . 22
4.4 Cryptographic hash function . . . 22
4.4.1 Message-Digest family . . . 22
4.4.2 Secure Hash Algorithms family . . . 23
4.5 File formats of our interest . . . 24
4.5.1 Office 95 . . . 24
4.5.2 Office 97 / 2000 . . . 24
4.5.3 Office XP / 2003 . . . 25
4.5.4 Office 2007 . . . 25
4.5.5 Office 2010 . . . 25
4.5.6 Office 2013, 2016 . . . 25
4.5.7 ZIP . . . 25
4.5.8 Portable Document Format (PDF) . . . 25
4.5.9 Roshal Archive(RAR) . . . 25

4.5.10 7Z . . . 25
4.6 Attack types . . . 25

**5 Formalization** **27**
5.1 Non-preemptive formalization . . . 27
5.2 Preemptive relaxation . . . 29
5.3 Heuristic approaches . . . 29
5.3.1 First Come First Served (FCFS) algorithm . . . 30
5.3.2 Round-Robin (RR) algorithm . . . 30
5.3.3 Dynamic Rebalancing algorithm . . . 30

**6 Implementation** **31**
6.1 Technological stack . . . 32
6.1.1 Python . . . 32
6.1.2 Docker . . . 32
6.1.3 Docker Compose . . . 32
6.1.4 S3 . . . 32
6.1.5 PostgreSQL . . . 32
6.1.6 RabbitMQ . . . 32
6.1.7 Redis . . . 33
6.1.8 Celery . . . 33
6.1.9 Grafana . . . 33
6.1.10 Kibana . . . 33
6.2 Implementation . . . 33
6.2.1 XtoHashcat . . . 33
6.2.2 Shovel . . . 35
6.3 Files generator . . . 36
6.4 Cracker . . . 37
6.4.1 Hashcat in detail . . . 37
6.4.2 John the Ripper in detail . . . 38
6.4.3 Cracker . . . 38
6.4.4 All wrapped to Docker . . . 39
6.5 Scheduler . . . 40
6.5.1 Scheduling implementations . . . 40
6.5.2 First Come First Served algorithm . . . 41
6.5.3 Round-Robin algorithm . . . 41
6.5.4 Dynamic Rebalancing algorithm . . . 42
6.6 Unpacker . . . 44

**7 Evaluation** **45**
7.1 Benchmark . . . 45
7.1.1 Hashcat . . . 45
7.1.2 John the Ripper . . . 46
7.1.3 Comparison of cracking tools . . . 47
7.1.4 Incoming data analysis . . . 47
7.1.5 The best-case benchmark . . . 48
7.1.6 The worst-case benchmark . . . 49
7.2 Artificial data . . . 50
7.2.1 Weight for the Dynamic Rebalancing algorithm . . . 51
7.2.2 15/15 scenario . . . 51
7.2.3 3/27 scenario . . . 53
7.2.4 First Come First Served vs. Round-Robin like implementations . . . 55

7.3 Real data . . . . . 56
7.3.1 Hand-picked real data . . . . . 56
7.3.2 Data in real-time . . . . . 58
7.3.3 Peak measurement . . . . . 59
7.3.4 Analysis of cracked passwords from real data . . . . . 61
7.3.5 Experiments conclusion . . . . . 61

**8 Conclusion and future work** **63**

**Bibliography** **65**

**A Attached files** **71**

# Acronyms

**7Z** 7-Zip files archive format.

**AES** Advanced Encryption Standard.

**ALS** Adaptive List Scheduling algorithm.

**AMMS** Adaptive Min-Min Scheduling algorithm.

**AMQP** Advanced Message Queuing Protocol.

**API** Application Program Interface.

**BOINC** Berkeley Open Infrastructure for Network Computing.

**c/s** Crypts per second.

**CPU** Central Processing Unit.

**CUDA** Compute Unified Device Architecture.

**DABRR** Dynamic Average Burst Round-Robin algorithm.

**DNS server** Domain Name System server.

**ELK Stack** Elasticsearch, Logstash, and Kibana.

**EML** Ecological Metadata Language.

**FCFS** First Come First Served algorithm.

**FPGA** Field Programmable Gate Array.

**GPU** Graphics Processing Unit.

**H/s** Hashes per second.

**HEFT** Heterogeneous Earliest Finish Time algorithm.

**ILP** Integer Linear Programming.

**KSA** Key-Scheduling Algorithm.

**LBIMM** Load Balance Improved Min-Min algorithm.

**LIST** List Scheduling algorithm.

**LPT** Longest Processing Time algorithm.

**MD** Message-Digest algorithm.

**MILP** Mixed-Integer Linear Programming.

**MPI** Message Passing Interface.

**NT LAN** New Technology Local Area Network.

**OpenCL** Open Computing Language.

**OpenMP** Open Multi-Processing.

**OS** Operating System.

**PA-LBIMM** User-priority Awared Load improved Min-Min algorithm.

**PDF** Portable Document Format.

**PHP** PHP: Hypertext Preprocessor.

**PoC** Proof of Concept.

**PostgreSQL** Postgre Structured Query Language.

**PSO** Particle Swarm Optimization.

**RAR** Roshal Archive.

**RC4** Rivest Cipher 4.

**REDIS** REmote DIctionary Server.

**RR** Round-Robin algorithm.

**S3** Amazon Simple Storage Service.

**SHA** Secure Hash Algorithm.

**SJF** Shortest Job First algorithm.

**TCP** Transmission Control Protocol.

**WEP** Wired Equivalent Privacy.

**WPA** Wi-Fi Protected Access.

**YAML** YAML Ain't Markup Language.

**ZIP** Compressed File format.

# Chapter 1

# Introduction

## 1.1 Introduction

The primary objective of password cracking is to retrieve a password in a plain text from an encrypted form as fast as possible. The passwords are used to hide sensitive data by and from governments, armies, normal users, but unfortunately, even hackers.

Hackers and attackers are nowadays more clever and send malicious software not as a standard file but as an encrypted file instead. It is meant to be decrypted later by the victim or by the attacker itself. The encrypted file might be sent to the victim, for example, via e-mail, and the password might be provided in the e-mail message body. This makes it impossible for antivirus companies to detect an infected file and for malware analysts to examine the virus. To solve this issue we implement the **Password cracking service**.

The form of user's passwords changed over time. First recommendations from (Morris and Thompson, 1979) suggested either a more than six-letter password comprising lower-case ASCII symbols or a five-letter password with both lower and upper case symbols. (NIST, 2017) suggested the use of at least eight-letters long passwords with no repetitive letters or common patterns and also encouraged users to use the *state-of-the-art* among password creation - passphrases. Besides being easy to remember, passphrases are also sufficiently long since they consist of at least four words, and are hard to guess by dictionary attack since recommendations mandate that the words are not connected to each other in any way. The article from (Microsoft, 2017) advises users to set up a password of the length of eight characters which meets the complexity requirements such as password must contain three out of five of the following different symbols: an uppercase, or a lowercase letter, a number, a non-alphanumeric case or any Unicode character not mentioned earlier.

These are examples that are still presented to users to make their password strong enough. However, users were taught by these guidelines to compose passwords that are hard to memorize yet easy to crack, with the exception of passphrases. Six-letter passwords are crackable for the majority of formats, and eight-letter passwords are nowadays considered to be easy to break as well if the encryption is weak or if only numbers are used. The authors of the article from (Hart et al., 2019) use eight NVIDIA GTX 2080Ti graphics cards to prove that it is possible to crack an eight-character Microsoft Windows NT LAN Manager password, which is still used as an authentication method in Active Directory environments, under 2.5 hours.

Today, the progress in computational power enables us to crack files even on commodity hardware. Lately, password cracking is done using not CPUs but GPUs instead (Hranickỳ et al., 2016), which brings speed-ups by up to two orders of magnitude. If the computational power of GPUs in one server is not enough, one can use botnets (infected

devices, lately IoT based, mostly acquired by hackers) or *Berkeley Open Infrastructure for Network Computing* (BOINC) (Anderson, 2004). They benefit from the fact that users share their commodity hardware for cracking passwords, performing medical research or other scientific research such as the search for extraterrestrial intelligence performed by (Anderson et al., 2002).

Another factor making our task easier is that users and hackers frequently use easy-to-guess, short passwords, usually already cracked and stored in the database from (Hunt, 2019), creator of `https://haveibeenpwned.com/` and well-known web security expert. Cloudflare offers an API to check whether a hash was found in (Hunt, 2019) database. According to (Avast, 2019) 83 % of the citizens of the United States use weak passwords, and only 47 % of the users use different passwords for different services. They also often use numbers from their birthday or their pet's name as passwords.

The main goal of our thesis is to incrementally crack a stream of incoming encrypted files. To process the files in a reasonable timespan, we need to select an appropriate scheduling algorithm to maximize resources utilization. We examine the possibilities of such algorithms and evaluate their performance. Our research focuses on an automatic processing of encrypted files, efficient cracking of their passwords, and uncovering their contents.

We divided our thesis into chapters according to our focus areas. In Chapter 2, we describe the background for the thesis, reviewing in particular the related works in scheduling algorithms. Chapter 3 introduces the most important cracking engines emphasising their benefits and disadvantages. In Chapter 4, we present the terminology, the history behind ciphers, some relevant encryption and cryptographic hashing algorithms used in file formats we want to crack. In Chapter 5, we formally define the problem. In Chapter 6, we describe each library and framework which is essential for our implementation and we describe the creation of our 3 microservices. Our implementations are tested in Chapter 7. The cracking and scheduling efficiency is evaluated through the use of both artificial and real-world data. Then we explore the ability of our selected algorithm in the production setting. Finally, Chapter 8, summarizes our thesis and suggests potential future work to continue exploring password cracking, potential use of different attack types, and algorithms to generate better dictionaries.

# Chapter 2

# Related work

In our case, the problem differs from traditional scheduling problems since in offline scheduling, the number of jobs, processing times (times to complete the process), release dates (times when jobs become available), due dates (times when jobs should be completed), deadlines (times when jobs must be completed), etc. are known in advance. First, we describe the *First Come First Served* (FCFS), *Shortest Job First* (SJF) and *Round-Robin* (RR) algorithms and their improvements. FCFS and RR algorithms are well-suited for load balancing since they do not need prior knowledge about the job. Afterwards, we review the *List Scheduling* (LIST) algorithm that creates an ordered list of tasks and assigns them to a free worker, *Min-Min* and *Max-Min* algorithms to see heuristic used in these algorithms. At the end of this chapter, we review work based on stochastic scheduling and robust optimization that is later used to formalize our problem.

## 2.1 First Come First Served (FCFS)

FCFS (Salot, 2013), also known as *First In First Out* is a simple algorithm that takes a job on the top of the queue and serves it to a free worker so it schedules the work in exact order from the queue. It is a non-preemptive algorithm (meaning that a job cannot be stopped and subsequently resumed) and Altmeyer et al. (2016) states, that it is categorized as a soft real-time algorithm (meaning that the jobs can exceed the deadlines). Since we are not pressured by deadlines, we use this algorithm as a baseline.

## 2.2 Shortest Job First (SJF)

Shortest Job First is an algorithm that takes the process with the shortest execution time and schedules it for execution.

## 2.3 Round-Robin (RR)

Round-Robin is often used in cloud computing. According to Wang and Casale (2014), modifications of RR are used by Heroku and Azure. RR is also used for handling requests to DNS servers (Hong et al., 2006). RR emits jobs in a First Come First Served manner but it is limited by a fixed time window. It is a preemptive algorithm fair in load distribution. The efficiency of RR lies solely in the choice of time given to each job (time quantum). RR executes each job for a fixed time quantum, then pauses the job and starts another job giving all jobs the same time quantum. The algorithm is widely used even outside the field of cloud computing. Dash et al. (2016) improved Round-Robin and called it *Dynamic Average Burst Round Robin* (DABRR). DABRR, in contrast with RR, dynamically

changes the time quantum. The quantum is computed from burst times (time required by a job for executing on a machine) of remaining times in the queue. Yadav et al. (2010) modified RR to use the SJF algorithm. Mohanty et al. (2011) propose a priority-based dynamic RR which changes time quantum for every round of execution. They prioritize tasks based on user's preference. We implement Round-Robin algorithm in Chapter 6 and later use dynamic improvement to decide the priority order of tasks.

## 2.4 The List Scheduling (LIST) algorithm

Another example of a simple scheduling algorithm is List Scheduling. We have a list of jobs, and every time a worker is free, we assign it a job from the head of the list (Michael, 2018). The list can be ordered using several heuristics, such as *Heterogeneous Earliest Finish time* presented by Topcuoglu et al. (1999). This heuristic focuses on heterogeneous systems and utilizes the fact that communication takes some time. Another commonly used heuristic is the *Longest Processing Time* (LPT). Li et al. (2010) present an adaptive resource allocation mechanism for preemptable jobs in cloud systems with two adaptive scheduling algorithms. One mechanism presented by Li et al. (2010) is called *Adaptive List Scheduling* (ALS), and the other is called *Adaptive Min-Min Scheduling* (AMMS).

## 2.5 Min-Min and Max-Min

Etminani and Naghibzadeh (2007) review the Min-Min and Max-Min algorithms. Min-Min takes the minimum completion time for each job found for all machines. The job with the minimum completion time from all jobs is assigned to the corresponding machine, then it is deleted from the set of tasks and the algorithm is running again until there are no unassigned tasks. Max-Min algorithm is almost identical to Min-Min, except in the second phase, the Max-Min selects the job where the completion time is the maximum. Chen et al. (2013) for load balancing in cloud computing. They improve upon the Min-Min algorithm by introducing two new algorithms: *Load Balance Improved Min-Min* (LBIMM) and *User-priority Awared Load improved Min-Min* (PA-LBIMM). LBIMM takes into consideration the workload of each resource and ensures no resource is being still. Since neither Min-Min nor LBIMM considers user-priority, PA-LBIMM is needed. The tasks are divided into VIP and non-VIP groups where VIP group is first distributed to resources (using Min-Min) and then the rest of the tasks are distributed by Min-Min again.

## 2.6 Online scheduling

In online scheduling, jobs arrive at unknown times. Processing times are not known in advance until the jobs are finished. Wen and Du (1998) present a problem with two uniform processors, each having different speed. The objective is to minimize the makespan (the time between the start of the first job and end time of the last job) as an objective. Vestjens (1998) focuses on scheduling $n$ jobs on $m$ machines while minimizing the makespan. However, the processing time $p$ is known upon arrival, unlike our case. Megow and Vredeveld (2006) use the so-called *Gitting indices*, which are values that represent rewards for a given stochastic process, to determine the lower bound for each job in preemptive stochastic scheduling to minimize the sum of weighted completion times. Chen and Shen (2007) state that, for weighted jobs with unknown processing time, generic non-delay algorithms, such as First Come First Served, perform very well for a large number of jobs.

## 2.7 Stochastic scheduling

In stochastic scheduling, the processing times are usually not known in advance. However, a probability distribution over time is used to give an estimate of a processing time. The actual processing time remains unknown until the task ends. Stochastic scheduling is widely used in an online environment to describe the nature of the problem. Lee et al. (2012) schedule workers to machines and minimize makespan on uniform parallel-machine ($m$ machines, where each machine can have different speed). They introduce two heuristic algorithms, one based on a genetic algorithm, and the other using the *Particle swarm optimization* method, which is inspired by the collective behavior of animals, such as a flock of birds. The paper by Liu et al. (2019) assume that only the probability distribution is known (or can be at least estimated). They present two approaches, one based on an average approximation of a sample and the other a hierarchical approach based on mixed-integer second-order cone programming.

## 2.8 Robust optimization

Robust optimization does involve use stochastic techniques, but makes the processing times robust instead. Xu et al. (2013) solve a robust identical (each machine has the same speed) parallel-machine scheduling problem and minimize the maximum makespan in a manufacturing setting. They avoid the estimation of a probability distribution by using a robust *Min-Max regret model with minimal maximum deviation* from all possible scenarios. Seo and Do Chung (2014) apply robust optimization to the identical parallel-machine with unknown processing time. They define a robust schedule and study the trade-off between robustness and conservativeness.

# Chapter 3

# Related password cracking engines

First, we present two primary password cracking engines to the reader to get familiar with the problematique; *Hashcat* (Steube, 2009) and *John the Ripper* (Openwall, 1996). Then, we discuss distributed solutions and projects built on Hashcat and John the Ripper. At the time of writing, only two actively maintained open-source solutions for distributed computation exists for Hashcat – *Hashtopolis* (s3inlc, 2020) and *Fitcrack* (Hranickỳ et al., 2019). There are currently two outdated purely parallel implementations for John the Ripper supported by the community (Openwall, 2014).

## 3.1 Hashcat

Hashcat is released as open-source software under the MIT license. Hashcat claims to be the "World's fastest and most advanced password recovery utility". This version is a merge of two products, previous CPU-based Hashcat (now called hashcat-legacy) and GPU-based *oclHashcat*. It is capable of cracking passwords using CPUs, GPUs, custom FPGA, and many other hardware capable of Open Computing Language (OpenCL) runtime. It has currently 6,900 stars on GitHub and an active and healthy community. Hashcat supports over 300 different hash formats.

### 3.1.1 Hashtopus

*Hashtopus* by (Curlyboi, 2020) was an open-source project distributed as a GPU wrapper around Hashcat but the project was abandoned. It was written in C♯ and PHP.

### 3.1.2 Hashtopolis

Hashtopolis is a multi-platform client-server tool for distributing Hashcat tasks to multiple computers and one of the two actively maintained open-source projects of this kind. Its main disadvantage is that we need to set the Hashcat parameters (a hash or a list of hashes, password dictionaries, and an attack type) by ourselves, and thus it is not completely standalone. Hashtopolis handles communication with nodes and divides the work (keyspace distribution) automatically.

### 3.1.3 Fitcrack

Fitcrack is a distributed password cracking system. It has the capability of cracking different documents, files, and raw hashes. It leverages Hashcat as its core component that is capable of using OpenCL and thanks to this fact, Fitcrack is capable of running on CPUs, GPUs, and many other types of processing units.

Fitcrack uses an adaptive scheduling algorithm implementation since it is based on the *Berkeley Open Infrastructure for Network Computing* (BOINC) computational framework (Anderson, 2004). This algorithm works in two modes – *non-targeted* and *targeted*. In the former, a job goes to any free worker. In the latter, a job goes to a specific worker selected by the user. Fitcrack uses the latter mode.

This application uses Python script `XtoHashcat.py` from (Zobal and Hranický, 2018) that we will later use for extracting the hashes from files for further cracking using *Hashcat*.

### 3.1.4 Terahash

(Terahash, 2020) is the self-proclaimed leader in distributed password cracking. In April 2020, *Terahash* acquired a company called *L0phtCrack* that provides auditing software for users and teams. Terahash provides both software and hardware needed for cracking and can be an easy to use solution for auditing companies.

## 3.2 John the Ripper

John the Ripper is a free password cracking software. Its disadvantage is its limited support for the GPU. This causes a major slowdown, and for this reason, we have selected Hashcat to be our main cracking tool. However, John the Ripper supports certain formats that Hashcat does not (e.g., RAR3, ZIP2 with long hash size). John the Ripper also comes with a few useful tools, such as zip-to-hash or rar-to-hash utilities. These utilities later presented in Subsection 6.2.1 enable us to obtain the hash of a file that we use in both Hashcat and John the Ripper.

### 3.2.1 John the Ripper MPI and OpenMPI

John the Ripper has implementations depending either on MPI or OpenMP, however, both are outdated. There are 15 implementations of the parallel approach mentioned by (Openwall, 2014). However, all of them are outdated as well and are only processor-based relying mostly on *OpenMP* and *MPI*. (Lim, 2004) implemented the first parallel version of John the Ripper using MPI, where the keyspace is equally divided to processors that come back for more when they exceed the keyspace. (Pippin et al., 2006) expanded this solution using a dictionary attack and giving each node different hashes, while each node contains the same dictionary.

# Chapter 4

# Introduction to password cracking

In this chapter, we make the reader familiar with the basic terminology of password cracking, encryption, and hashing. These terms are later used in the description of formats of our interest and give us an intuition of the speed and security of each format.

## 4.1 Terminology

At the very beginning, we need to explain the difference between encryption and hashing in order to be consistent with the terminology. The operation that encodes the message using a chosen key is called **encryption** and is said to be a two-way function. The encoded message can be later decoded using the proper key. The operation where we use a one-way mathematical function to transform the message into a fixed-size so-called hash. This algorithm is called a **cryptographic hash function**, and in contrast with encryption, a cryptographic hash function is meant to be irreversible and has unique mapping.

Since this master thesis is focused on the recovery of passwords from files, our main interest is **cryptographic hashing**.

## 4.2 Historical context

Humans have had secrets since ancient times. In order to store them, the written form of the secret has to be illegible for the unauthorized reader. One of the first attempts to hide people's secrets was simply writing it as a text since a lot of people could not read. A slightly advanced version of this approach was the use of hieroglyphs since only readers belonging to kings could read.

Another famous way of encryption was introduced by the Greeks. The method is called *Caesar Shift Cipher*. To encrypt the message, the letters are simply shifted by an agreed number (usually 3, so the 'a' becomes 'c', b becomes 'd', etc.), and the mapping from original is made to the shifted alphabet. To decrypt the alphabet, we simply shift the alphabet in reverse order.

Later in the 16th century, the first cipher with encryption key was made. This cipher is called *Vigenère cipher*. First, we need to pick the key, then the key has to be duplicated, until the length reaches the same size as the text, which is meant to be encoded. Then we create a 2D table with the size of 26 x 26, where the rows are letters corresponding to the key and the columns are plain text characters. The table is filled with alphabet, where each row is shifted by one to the left. The intersections of rows and columns give us the encrypted character. One of the most famous ways of encrypting messages was *Enigma* during World War II (Damico, 2009).

## 4.3 Encryption

In this section, we describe stream cipher *Rivest Cipher 4* (RC4) (Rivest, 1992) and *Advanced Encryption Standard* (AES) (Rijmen and Daemen, 2001). These encryption functions are later used in file formats of our interest.

### 4.3.1 Rivest Cipher (RC4)

RC4 was invented by Ron Rivest in 1987. An interesting fact is that the implementation was a trade secret until it was leaked (Fluhrer et al., 2001). It is one of the most used stream ciphers since it is used in TCP, WEP, WPA, BitTorrent, PDF, and Kerberos, and its modifications are used in other ciphers. RC4 initializes an array of 256 bytes and runs the *Key-Scheduling Algorithm* (KSA) to create a permutation of that array. Now we need to run a pseudorandom generation algorithm to KSA output. The last step is to XOR the data with the keystream.

### 4.3.2 Advanced Encryption Standard (AES)

AES is a block cipher. AES was an answer to old *Data Encryption Standard* (DES), which is broken. AES has the length of the block of 128 bits and 3 options of the size of the encryption key (128, 192, and 256 bits), which interfere with the number of rounds (10, 12, 14). The input key is expanded, and from the result, the round keys are computed. Almost every round consists of operations *SubBytes*, *ShiftRows*, *MixColumns*, and *AddRoundKey*. Only in the last round, the operation *MixColumns* is not performed since it does not improve the safety of the cipher.

## 4.4 Cryptographic hash function

As we mentioned above in the Section 4.1, cryptographic hashing is a mathematical function that maps message of any size to a message of a fixed size, usually called a **hash**. Hashing, in contrast with encryption, is a one-way function (Al-Kuwari et al., 2011). A one-way function should not be possible to revert. However, that depends on the selected hashing algorithm, and computational power.

The ideal cryptographic hash function must fulfil the following requirements:

1. For the same input message, it produces same result; thus it is deterministic.
2. It calculates the cryptographic hash for every input message fast.
3. It is next to impossible to revert the hash and get the original message.
4. It is not possible to find two messages with the same hash that vary from each other. Even a small change in the message causes a tremendous change in the hash value. This is called the avalanche effect.

### 4.4.1 Message-Digest family

#### Message-Digest 2 (MD2)

The first *Message-Digest* algorithm was named MD2 and was invented in 1989 by Ronald Rivest and described by Kaliski (1992). The plain text message is padded. Padding is

an extension of a message in order to modify its length to make it divisible by 16 without a remainder. This extended message of the length **n** is then appended by 16-byte checksum.

To compute the hash itself, we need an S-table and auxiliary block. S-table has 256 bytes (table of size 16 x 16) generated from random permutation of $\pi$. The auxiliary block is defined as a 48-byte buffer initialized to 0 and is permuted 16 times for every 16 bytes from the input bytes. After the iterations ends, the first block of the auxiliary block becomes the hash value.

#### Message-Digest 5 (MD5)

MD5 is one of the most used hash functions. It is an improvement of MD2 and was developed in 1992. MD5 can not be considered safe since Wang et al. (2004) found a collision for MD5 and should be used solely as a fingerprint (e.g., software packages, etc.). The MD5 hash is also fast to compute even for commodity hardware, which makes it dangerous to use as shown in Table 7.1.

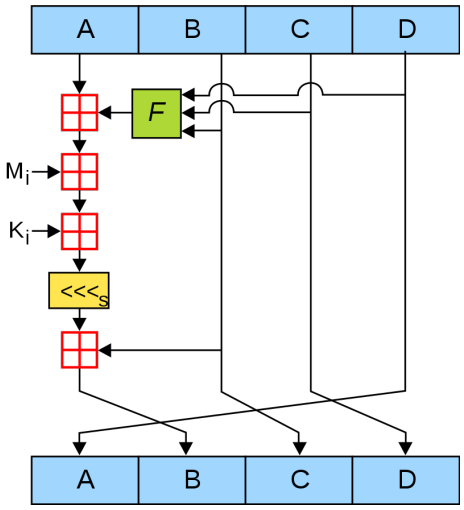



Figure 4.1: In this picture, the message divided into the block of 512-bits blocks. If the message is shorter than 512 bits, it is padded the same way as for the MD2 algorithm. Then the message is divided into 4 parts. For each part, similar operations such as left addition or modulo are used. Picture by Crypto (2007), distributed under a CC-BY-SA 2.5 license.

### 4.4.2 Secure Hash Algorithms family

This family of algorithms was invented by the National Security Agency of the United States of America. It is a Federal Information Processing Standard. It is one of the most used hash algorithms.

#### Secure Hash Algorithm 1 (SHA1)

SHA1 was invented in 1993. This hashing algorithm is also not considered safe since 2005 because a possible attack was found. A successful collision attack was performed by (Stevens et al., 2017). However, it is still required to be used in several U.S. government applications. SHA1 makes a fingerprint of 160-bit length. In addition to collision attack, it is easy to compute the SHA1 hash as can be seen in Table 7.1.

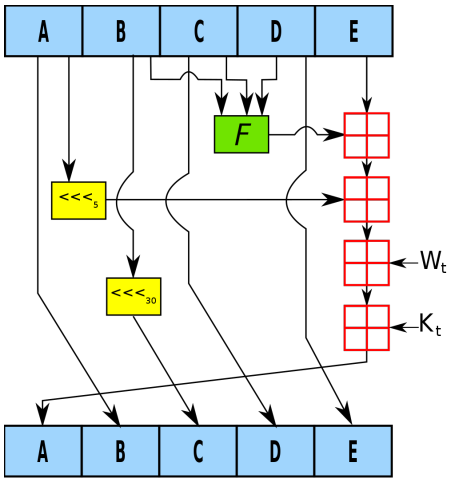



Figure 4.2: SHA-1 algorithm. Picture by Crypto (2006), distributed under a CC BY-SA 2.5 license.

#### Secure Hash Algorithm 2 (SHA2)

The SHA2 is an improvement of SHA1. Here we have 6 hash functions instead of 4. There is currently no known collision for a full round run of the algorithm. However, the variant of SHA2 that is the most commonly used, SHA256 is advised not to use for cryptographic purposes.

#### Secure Hash Algorithm 3 (SHA3)

SHA3 algorithm is completely new, introduced by (Dworkin, 2015) and different from the previous SHA algorithms. The origin of the algorithm comes from *Keccak algorithm* family. This is currently the hashing standard.

## 4.5 File formats of our interest

In the previous section, we described various encryption methods. These functions are used heavily in file formats we are interested in. The goal of our thesis is to be able to decrypt these formats; thus, it is beneficial to know their strength of encryption since the speed of cracking them is based on it. We choose to cover these formats since they are the most commonly seen sources of malicious content.

### 4.5.1 Office 95

The first encryption used in Office 95 was a simple *XOR cipher* algorithm (Wu, 2005). This algorithm is more an obfuscation method rather than true encryption. Thus we consider it unsecure since it is easy to crack it using frequency analysis or by known-plaintext attack.

### 4.5.2 Office 97 / 2000

Since Office 97 / 2000, RC4 for encryption and MD5 hashing algorithm were used. These two are also not considered to be safe as stated above since we can easily brute-force billions of hashes per second.

### 4.5.3 Office XP / 2003

Here the Office XP / 2003 encryption is the same as it is with Office 97 / 2000. But users can choose to use a custom encryption algorithm.

### 4.5.4 Office 2007

Starting with the 2007 version, Office employs AES encryption. Password verification is made using 50000 rounds of SHA1.

### 4.5.5 Office 2010

Office 2010 uses AES, and password verification is made using 100000 rounds of SHA1.

### 4.5.6 Office 2013, 2016

Office 2013 and 2016 use AES as well, but key length increased from 128-bit to 256-bit. Password verification is made using 100000 rounds of SHA1.

### 4.5.7 ZIP

According to (WinZip, 2020), the ZIP is encrypted using 128-bit or 256-bit AES encryption in a new format zip 2.0, and legacy encryption in old ZIP 1.1 format.

### 4.5.8 Portable Document Format (PDF)

According to Adobe (2020), Acrobat 6.0 and later (PDF 1.5) uses 128-bit RC4 encryption, Acrobat 7.0 and later (PDF 1.6) uses AES encryption algorithm with a 128-bit key size. Acrobat X and later (PDF 1.7) uses 256-bit AES encryption.

### 4.5.9 Roshal Archive(RAR)

Roshal (2020) invented *Roshal Archive* (RAR). According to Roshal (2020) WinRAR uses 128/256 Bit AES for RAR3, RAR4, and RAR5.

### 4.5.10 7Z

The 7Z format created by Pavlov (2020) uses AES encryption with a 256-bit key. The key for the archive is hashed to SHA-256, where it is executed $2^{18}$ (262144) times; thus, both encryption and decryption take considerable time.

## 4.6 Attack types

To crack the password, we have several techniques at our disposal. The first that comes to the mind of every person is to try every possible combination of letters and numbers for each position of given length - **brute-force attack**.

In order to decrease the number of password candidates for each hash, we can leverage lists of most used passwords (dictionaries, thus **dictionary attack**), usually obtained from leaked databases of various service providers such as well-known database from ***RockYou*** (security Skull, 2020), a software company creating third party software for social media services, ***MySpace*** (security Skull, 2020), social media networking site, ***Ashley Madison***, a dating site, ***LinkedIn***, American business and employment-oriented service. Hunt (2019), the creator of well-known site `haveibeenpwned.com`, shares the biggest leaked password databases yet. It aggregates all the aforementioned and many more

databases, allowing users to check their e-mail address or login credentials against the ever-growing dataset. An interesting dictionary is provided by Hornby (2011), where the dictionary was build in order to raise awareness between users with weak password encryption. Brute-force is quicker than dictionary attack in terms of tried combinations per second because modern cracking tools do not load the password candidates from a hard-drive but store the candidate in the memory of the processor and change the appearance, thus password candidate by only one bit. However, the search space is much larger thus carefully picked dictionary might suit the problem better.

The dictionary attack might be further improved by using rules and masks; thus, combined attacks are carried out. The **rule-based attack** is a dictionary attack that applies rules on each word from the dictionary using specific language defined by a cracking tool. The two most well-known cracking tools use one language expressiveness to describe the word modification. The ability of rules is huge. The user can append, concatenate, multiply, replace any character in word he wishes to. **Mask attack** is the same as the brute-force attack with the exception that it does not try all combinations from the search space but only given domain specified by mask (e. g. mask ?l?l?l?l?l?d?d represents all 6 lower-key letters with 2 numbers at the end).

# Chapter 5

# Formalization

Our goal is to solve the scheduling problem on identical machines with high utilization of resources in an online environment. The release times and processing times of each job are not known beforehand. First, we formalize the task as a non-preemptive (a job cannot be stopped) *mixed-integer linear programming* (MILP) in an offline environment. Later we propose preemptive (a job can be stopped and resumed) formalization to show that it is possible to describe the problem using preemption and allows us to switch between jobs to clear out the easiest cases where the solution of cracking is trivial, which is later used in heuristic algorithms. Afterward, we present a heuristic approach to the problem because the cost (regarding CPU and GPU time) is prohibitively high for larger instances. Also, many ILP solvers are not available for free for business entities. We formulate our task as parallel processing unit makespan minimization with the addition of the instruments of robust optimization. Using these instruments, we are able to hold certain guarantees regarding the makespan even in the worst-case scenario.

Employing *robust optimization*, we are able to find the best worst-case solution, which is a solution that performs well even in a worst-case scenario and is later used in Chapter 7 to give us the optimum makespan value, to compare our heuristics algorithms with.

## 5.1 Non-preemptive formalization

Our MILP formalization is based on the work of Seo and Do Chung (2014). The goal is to distribute $N$ jobs denoted as $J_j : j \in \{1, \cdots, N\}$ on $M$ parallel processors denoted as $P_i : i \in \{1, \cdots, M\}$, where $i$ stands for i-th processor. One job corresponds to the cracking of one file and to ensuring that the criterion function is optimized and each job is scheduled only once.

Firstly, we define non-preemptive formalization. We have binary variable $x_{i,j} : i \in \{1, \cdots, N\}$, j $\in \{1, \cdots, M\}$ for the following holds:

$$x_{i,j} = \begin{cases} 1, & J_j \text{ is scheduled on processor } P_i \\ 0, & \text{otherwise} \end{cases} \tag{5.1}$$

We use symbol $p_j : j \in \{1, \cdots, N\}$ for processing time corresponding to job $J_j$. Lengths of these processing times are not known to us in advance, but only after the execution of the job has ended. The completion time of processing unit $i$ is characterized as:

$$C_i = \sum_{j=1}^{N} x_{i,j} \cdot p_j, \forall i \in \{1,\dots,M\} \tag{5.2}$$

We define the objective function we want to optimize as follows:

We want to minimize the maximum makespan (which is the time of the schedule which elapses from the start of the first scheduled job till the end of the last one). This allows us to distribute jobs to processors equally and maximally utilize them. This can be formally described as:

$$\text{Minimization of } C_{max} = max\{C_i\} = max_{i=1}^{M} \sum_{j=1}^{N} x_{i,j} \cdot p_j \tag{5.3}$$

With (5.1), (5.3) and the presumption that each job is scheduled only once, we characterize the MILP as:

$$\text{minimize:} \qquad C_{max} \tag{5.4}$$

$$\text{subject to:} \qquad \sum_{j=1}^{N} x_{i,j} \cdot p_j \leq C_{max}, \forall i \in \{1,\dots,M\} \tag{5.5}$$

$$\sum_{i=1}^{M} x_{i,j} = 1, \forall j \in \{1,\dots,N\} \tag{5.6}$$

$$\text{where:} \qquad x_{i,j} \in \{0,1\}, \forall i \in \{1,\dots,M\}, \forall j \in \{1,\dots,N\} \tag{5.7}$$

Equation (5.5) tells us that makespan on each processing unit is at most $C_{max}$, and equation (5.6) tells us, together with the fact that $x_{i,j}$ are binary variables, that every job is scheduled exactly once and on exactly one processing unit.

Since the processing time $p$ is not known until the cracking is finished, we introduce $\tilde{p}$ that is bounded by the minimum processing time $\underline{p}$ and maximum processing time $\overline{p}$, thus $\underline{p} \leq \tilde{p} \leq \overline{p}$. We can set these boundaries since we know the minimum processor time devoted to a job as well as we know the maximum time we want to allocate to the job. To obtain the feasible solution as the best worst-case scenario, we change (5.5) to:

$$\sum_{j=1}^{N} x_{i,j} \cdot \overline{p_j} \leq C_{max}, \forall i \in \{1,\dots,M\} \tag{5.8}$$

A less conservative approach towards the robust schedule inspired by Seo and Do Chung (2014) could be done by using the sum of $\underline{p}$ as the minimum required processing time and $\overline{p}$ - $\underline{p}$ multiplied by variable $r$, where $\overline{r_j} : j \in \{1,\dots,N\}$ belonging to $J_j$ denotes the probability that the job will take a longer time.

Rewriting (5.8) to:

$$\sum_{j=1}^{N} x_{i,j} \cdot (\underline{p_j} + (\overline{p_j} - \underline{p_j}) \cdot r_j) \leq C_{max}, \forall i \in \{1,\dots,M\} \tag{5.9}$$

$$\sum_{j=1}^{N} r_j \geq R \tag{5.10}$$

$$\text{where } r_j \in \{0,1\}, R \in [0,N], \forall j \in \{1,\dots,N\} \tag{5.11}$$

where $R$ in (5.10), which is the rate of robustness, gives us a bound for jobs and allows us to control additional processing time for each job. If R = 0, then we achieve the best-case scenario for each job, if R = N, we obtain the worst-case scenario.

## 5.2 Preemptive relaxation

Further, we allow preemption thus rewriting the problem to:

$$\text{minimize:} \quad C_{max} \tag{5.12}$$

$$\text{subject to:} \quad \sum_{t=1}^{T} c_t = C_{max} \tag{5.13}$$

$$\sum_{i=1}^{M} \sum_{t=1}^{T} x_{i,j,t} \geq p_j, \forall j \in \{1,\dots,N\} \tag{5.14}$$

$$\sum_{i=1}^{M} \sum_{j=1}^{N} \sum_{t_1=t}^{T} x_{i,j,t_1} \leq K \cdot c_t, \forall t \in \{1,\dots,T\} \tag{5.15}$$

$$\sum_{i=1}^{M} x_{i,j,t} \leq 1, \forall j \in \{1,\dots,N\}, \forall t \in \{1,\dots,T\} \tag{5.16}$$

$$\sum_{J=1}^{N} x_{i,j,t} \leq 1, \forall i \in \{1,\dots,M\}, \forall t \in \{1,\dots,T\} \tag{5.17}$$

$$\text{where:} \quad K \in \mathbb{N}, x_{i,j} \in \{0,1\}, \forall i \in \{1,\dots,M\}, \forall j \in \{1,\dots,N\}, \forall t \in \{1,\dots,T\} \tag{5.18}$$

$c_t$ is a working variable for which holds:

$$c_t = \begin{cases} 1 & \text{job J that is scheduled in time } t_1 \geq t \text{ exists} \\ 0 & \text{otherwise} \end{cases} \tag{5.19}$$

Now the variable $x_{i,j,t}$ has new meaning. We say that job $J_j$ is allocated on processor $P_i$ in time $t$ where the sum of $x_{i,j,t}$ adds into $p_j$. By adding equation (5.14) we say we use at least $p_j$ processing time for given job $J_j$.

Equation (5.15) says that if exists a time from $t$ to $T$ where in this time a job $J_j$ is scheduled, the $c_t$ has to be 1, otherwise 0. Variables $c_t$ create a sequence of 0 and 1 where, the last 1 is the last used time slot on any processing unit. This is what we minimize. At the same time, their sum corresponds to the index where the last job was scheduled, which is our $C_{max}$. $K$ is a constant corresponding to maximum time multiplied by the number of processing units.

Using equation (5.16) we say that one job $J_j$ can not run on multiple processing units at once. By (5.17) we ensure that each processing unit has only one job $J_j$.

## 5.3 Heuristic approaches

Since our main goal is to process a stream of incoming files in real-time, we decided to use a heuristic approach to job scheduling. We implemented three techniques that we later compare in Chapter 7 as well as with the non-preemptive MILP model implementation.

### 5.3.1 First Come First Served (FCFS) algorithm

At first we implemented a First Come First Served policy, where the jobs are scheduled based on their order in the queue. This algorithm is suitable where we have no prior knowledge about incoming jobs as shown by Chen and Shen (2007).

### 5.3.2 Round-Robin (RR) algorithm

As a second algorithm, we implement an algorithm that allows preemption, thus runs each job for fixed time windows unless the current cracking job has not ended or succeeded. In that case, job is enqueued at the end of the queue with the next cracking scenario if there is any scenario left. This enables us to schedule the jobs as soon as they ordered in the queue and not to block the resources with one cracking scenario job for a long period of time if it exceeds the fixed time.

### 5.3.3 Dynamic Rebalancing algorithm

Since a file cracking is a time-consuming effort, we employ two queues to contain the newly incoming jobs called *New Files* queue, and *In Progress* queue containing jobs moved from *New Files* queue after fixed time. If a new job is received, it is enqueued to the *New Files* queue. Jobs that take a long time to compute without success are moved to *In Progress* queue to be resolved for their remaining time. We propose a heuristic approach to choose the jobs based on the ratio between the sizes of the respective queues while giving a higher priority to the *New Files* queue. We utilize the processing units equally and using the preemption, we give a chance to any cracking attempt to be cracked sooner than without the preemption since it has a higher chance of success in the early stages of the cracking attempt. This is because first, we try dictionary attack, and later we switch to a brute-force attack. This is depicted in Subsection 7.2.4. After a fixed time the job is moved from *New Files* queue to *In Progress* queue and given a lower priority.

**Algorithm 1** *Dynamic Rebalancing* algorithm.

**Input:** New Files queue $Q1$, In Progress queue $Q2$, Weight $W$, Workers $\{x_1, \ldots, x_N\}$.

1: $Size1 \leftarrow len(Q1) \cdot W, Size2 \leftarrow len(Q2)$ ▷ Apply weight W
2: $amountPerResource = total\_size/len(W)$
3: $newFilesWorkersRatio = Size1/amountPerResource$
4: $inProgressWorkersRatio = Size2/amountPerResource$
5: **if** $newFilesWorkersRatio \geq inProgressWorkersRatio$ **then**
6: $doNewFilesJob()$ ▷ Allocates one job from New Files queue to resource
7: **else**
8: $doInProgressJob()$ ▷ Allocates one job from In Progress queue to resource
9: **end if**

# Chapter 6

# Implementation

We need two main parts: a password recovery software and an implementation of the appropriate scheduling algorithm. We are going to use *Hashcat* for cracking the files since it utilizes graphics cards for most of the formats, complemented by *John the Ripper* for the rest of the formats unsupported by Hashcat. We isolate each cracking tool instance in a separate container using *Docker* (Merkel, 2014). Isolating the instances enables us to run several cracking instances at one graphics card.

The Password cracking service comprises three microservices. The first microservice is *Shovel*, responsible for determining whether the file is encrypted, detecting the appropriate cracking tool, and extracting the words from e-mails, and expanding the wordlist used for cracking. Next, there is *Scheduler*, responsible for scheduling jobs on workers. The Scheduler executes the selected cracking tool. The last service is *Unpacker* which takes the file and its password and produces an encryption-free file.

This chapter also describes the technology stack we have used. Later, we describe each microservice and its implementation. Finally, we present the scheduling algorithms we implemented.

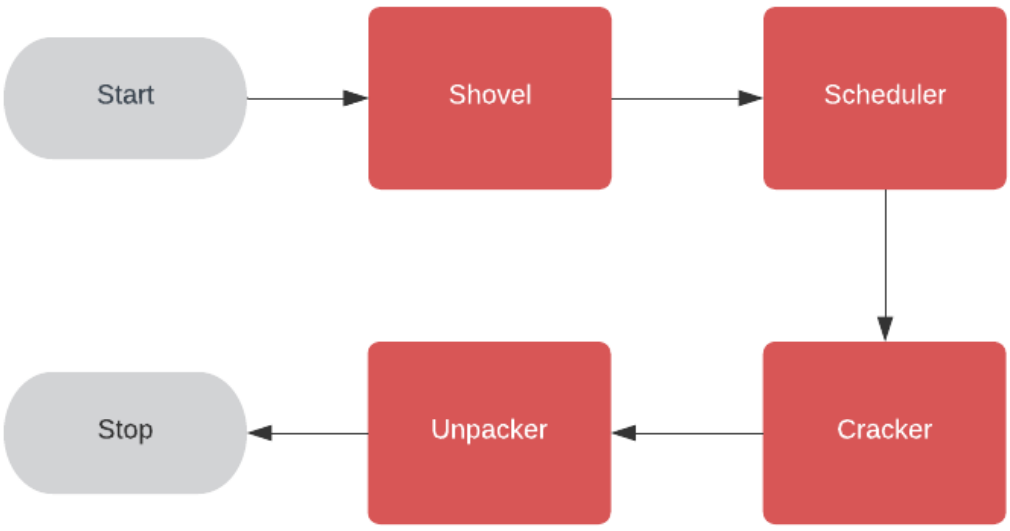



Figure 6.1: Microservices layout.

## 6.1 Technological stack

We have chosen state-of-the-art technologies which together make it easy to deploy, scale, and maintain the processing pipeline.

### 6.1.1 Python

As the core language, we have chosen *Python* (Rossum, 1995). It is an easy-to-understand yet sufficiently powerful programming language. All applications are written in an asynchronous way, which enables us to create and hold a connection with message broker and database without thread management.

### 6.1.2 Docker

Docker is open-source solution for isolation of applications in all well-known operating systems (Windows, Linux, macOS). Docker allows us to build lightweight microservices for all of our code because it does not use the whole OS since it shares the machine's OS system kernel. On top of that, it enables us to isolate each application from others, which became crucial for our use of Hashcat and John the Ripper since only one instance can be running in one OS.

Docker dramatically aids deployment and debugging. It provides a deterministic setup of the program. Once the Docker image is built on one computer it is no problem to transfer it to another computer and build it just the same. We use Docker to install all dependencies, such as Python, Python libraries, binaries corresponding to Hashcat and John the Ripper, *CUDA* - an API from NVIDIA company used as a parallel computing platform and NVIDIA drivers for GPU.

We also need *NVIDIA Container Runtime* from NVIDIA (2018), which enables utilizing GPUs within Docker containers.

### 6.1.3 Docker Compose

*Docker Compose* is a tool that enables us to run multi-container applications. The structure of the containers is defined in YAML file. Docker Compose provides commands to start, stop and restart our services as well as view and log the status of services.

### 6.1.4 S3

*Amazon Simple Storage Service* (S3) is object storage from Amazon (2015). It is used as our main storage for files.

### 6.1.5 PostgreSQL

To store the results of password cracking and temporary data, we use *PostgreSQL* developed by PostgreSQL Global Development (1996). It is an open-source relational database.

### 6.1.6 RabbitMQ

*RabbitMQ* (Pivotal, 2020) is an open-source message broker implementing *Advanced Message Queuing Protocol* (AMQP) and many other protocols. Our microservices are using RabbitMQ to communicate with each other. It is also utilized by the *Celery* (Celery, 2020) framework as its backing broker (The work is distributed using RabbitMQ messages) between the *Scheduler* and workers.

### 6.1.7 Redis

*Redis* (Sanfilippo, 2020) is an open-source in-memory key-value store.

### 6.1.8 Celery

We use Celery in our Scheduler to orchestrate the cracking jobs. Celery is an asynchronous task queue, which we configured with RabbitMQ as the backing broker, and Redis.

### 6.1.9 Grafana

*Grafana* (Grafana, 2020) is open-source for monitoring, analyzing, and visualizing the metrics from our microservices. Grafana is also capable of using alerts, that can be set for various situations based on metrics values. In our case, we monitor if the number of incoming messages is steady, and if the Scheduler is not idle. In that case, Grafana sends a message via e-mail or other communication channels. The graphs from Grafana can be seen in Subsection 7.3.3.

### 6.1.10 Kibana

We use the so-called *ELK Stack* (Banon, 2020). ELK is the acronym for *Elasticsearch*, *Logstash*, and *Kibana*. Elasticsearch is a search engine that relies on *Apache Lucene* query syntax language, which allows us to search the logs. It uses the Logstash as a processing pipeline and that sends the logs in Elasticsearch. Kibana is a frontend dashboard for visualization and search in our logs aggregated from all the microservices and a standard solution for log management.

## 6.2 Implementation

First, we create a microservice called *Shovel*. We begin with the creation of Shovel because we needed to analyze the stream of incoming files to know what we deal with. We needed a tool capable of distinguishing whether the file is encrypted or not. For this task, we used XtoHashcat incorporated in *Shovel* to learn what proportion of such files there is in the incoming file stream.

The second task was to get familiar with Hashcat and its commands and create an easy to use Docker image that wraps around Hashcat and its runtime dependencies. This image is called *Cracker*. Later we discovered that Hashcat does not support all formats we set out to support and would not be able to crack those. We decided to utilize John the Ripper to crack those few formats that Hashcat does not.

To be able to test Hashcat and John the Ripper, we created a tool that generates encrypted files.

After finishing *Cracker* for password cracking, we started to develop simple proof-of-concept *Scheduler* using the First Come First Served algorithm. Later we implemented the Round-Robin algorithm and improved the RR into the Dynamic Rebalancing scheduling algorithm.

To finalize the result, we created the *Unpacker* which, using different tools, strips the encryption, and produces its unencrypted form.

### 6.2.1 XtoHashcat

`XtoHashcat.py` is a script created by Zobal and Hranický (2018). It wraps the functionality of separated utilities responsible for extracting the password hash. Most of the

utilities come from John the Ripper. The password hash is later used in both Hashcat and John the Ripper. The password hash is a string that describes the encrypted file by properties such as: compression type, the checksum of the file, length of the file, the password hash, and other attributes.

The script uses the following source files and binaries:

1. `pdf2hashcat.py`

2. `zip2john`

3. `office2hashcat.py`

4. `rar2john`

5. `7z2john.pl`

For instance, using `pdf2hashcat.py` to obtain a password hash from a file looks like this:

```
root@6414b4ae84f2:/john-1.9.0-jumbo-1/run# ./pdf2john.pl pass.pdf
pass.pdf:$pdf$5*5*256*-1028*1*16*90a2af0e3948454a9435d3178ae75409*1
27*0ba02fceef968db364237a0308d4374c731257383e46b60e79647c88490c0ffb
5fba9b411bae85dde5af5ee97ea47ed700000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000*127*d77007
814b9e378a880597bc6c0918877d23de5d696f031ef2c9e5e1fd84d464958014998
a2de86c33a44741372466ce00000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000*32*d1ffabf549c91565
aa31a2ac0e75aa4f30108168305fa0a05b4dfa0bd07ecbbc*32*006c45e744093fc
4962b6333abcc71380e13eff198ce203bf0e68e431ed0df07
```

Figure 6.2: Pdf2john.pl call, used in XtoHashcat.

### 6.2.2 Shovel

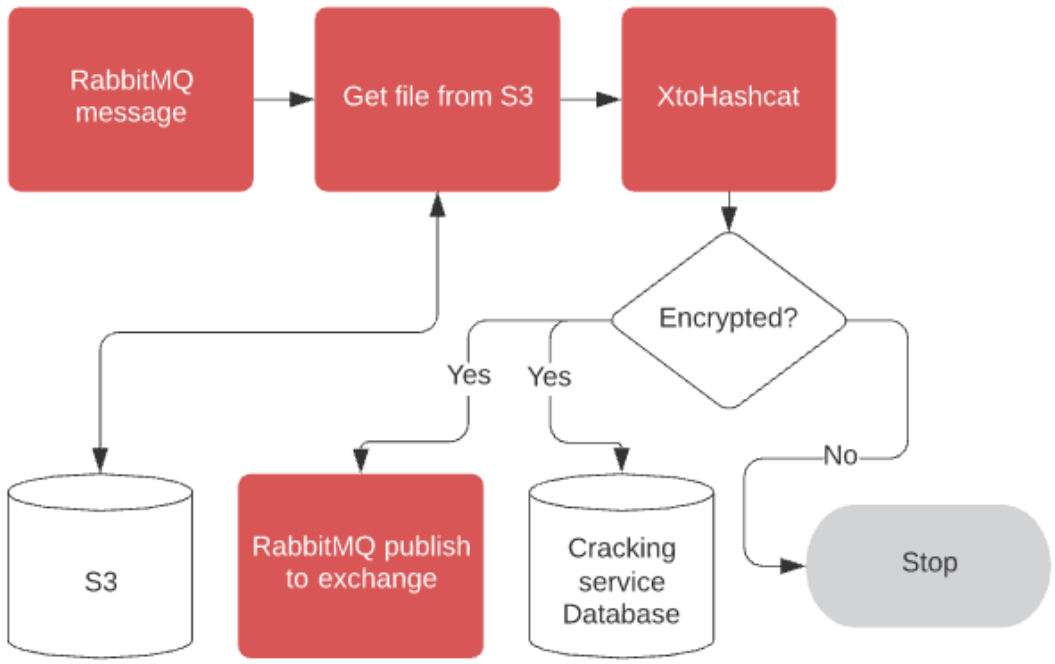



Figure 6.3: Shovel workflow.

The Shovel was constructed to read the stream of incoming files from a RabbitMQ exchange. The expected message looks like:

```
{
  "sha256": "5d5b09f6dcb2d53a5fffc6.....",
  "source": "...",
  "type": "pdf",
  "suggested_passwords": [
        "123",
        "password"
   ]
}
```

where:

1. **Sha256** - unique file identifier, SHA256 computed from the file content.
2. **Type** - file type.
3. **Source** - file source name, name of RabbitMQ message emitter.
4. **Suggested_passwords** - optionally, the user can specify passwords worth trying to crack.

First, we download the file from S3 storage, and then we use XtoHashcat. If XtoHashcat is able to acquire the password hash of the file, we check if we have prior knowledge about the particular file, such as an e-mail. The Shovel is able to parse corresponding EML source files and turn them into a dictionary using simple heuristics - dividing the words by space. This prior knowledge serves as a wordlist of potential passwords. Based

on the file's password hash, we also determine whether Hashcat or John the Ripper is used to crack the file. Finally, we forward the file for processing by the Scheduler, with a corresponding message:

```
{
  "sha256": "5d5b09f6dcb2d53a5fffc6.....",
  "type": "pdf",
  "cracking_tool":"hashcat",
  "start":True,
  "suggested_passwords": [
        "123",
        "password",
        "password_from_email"
   ]
}
```

1. **Sha256** - unique file identifier, SHA256 computed from the file content.
2. **Type** - file type.
3. **Start** - flag if the cracking starts for the first time.
4. **Suggested_passwords** - top passwords + words extracted from e-mails etc.

The decision whether the Scheduler uses Hashcat or John the Ripper is based on the file type and its password hash. We use John the Ripper to process RAR files of type 1 (password hashes starting with $RAR3$*1) since it is not implemented in Hashcat during the writing of our thesis and it is one of the most commonly used RAR encryption. ZIP version 2 is the second format that is handled only by John the Ripper. Unfortunately, Hashcat supports only cracking of short password hashes for ZIP (hashes starting with $zip2$) in terms of string length), so strings longer than 120 characters are sent to John the Ripper as well. The last format that is sent to John the Ripper is PKZIP2 since XtoHashcat does not correctly distinguish the format number for Hashcat, but John the Ripper detects the file type automatically and does not need this information.

Summary of files sent to John the Ripper:

| File format type | Start of the file password hash (signature) |
|---|---|
| RAR3 | $RAR3$*1 |
| ZIP2 (length >120 chars) | $zip2$ |
| PKZIP2 | $pkzip2$ |

The rest of the files are sent to Hashcat.

## 6.3 Files generator

To be able to test the data, we created a Python script that creates encrypted files in PDF, 7Z, and ZIP. We skipped the Office documents since there is not a proper command-line tool to encrypt Office documents and RAR formats since there is only proprietary software for creating RAR, however, to extract the RAR file an `unrar` command-line tool exists. We used Linux tools `qpdf` (Berkenbilt, 2013) for PDF files, `zip` (Gordon, 2020) for ZIP files, and `p7zip` (Trojette, 2016) for 7Z files to create the instances.

## 6.4 Cracker


Figure 6.4: Cracker workflow.

At first, we learn how to use Hashcat and John the Ripper. Afterward, we needed XtoHashcat to even obtain password hashes to test the functionality of both cracking tools. Finally, we connect all of these scripts with Python and make it as Docker image that can be given arguments to call different strategies we will later use from the Scheduler. The strategies are the same for both Hashcat and John the Ripper, however, the command is slightly different as shown below. Then we connect the parts into one microservice using Python.

### 6.4.1 Hashcat in detail

Firstly, we learned how to use Hashcat's command-line interface to be able to interact with it. One important feature of Hashcat is a flag `--machine-readable` which makes the output that can be parsed by computers. As a result of this convenient output, we track the progress of Hashcat and check whether the password was obtained.

Another feature that Hashcat offers is `--workload` flag - the user can specify how many of the available resources are to be allocated and utilized. The default value is 2 (economy), and we have chosen 3 to get the maximum output from each GPU, without rendering the machine overloaded and effectively unusable, which is a risk when choosing higher values

Hashcat is also capable of pausing the cracking job. The option to pause the cracking job becomes available after a `Restore.Point` is created, which is approximately 2 minutes from the start time after a certain keyspace of password candidates is searched. To make the restore file, we must give a name to session during a call of Hashcat command by using `hashcat --session cracking -m 0 -a 3 md5.hash`. We use the restore option later in Subsection 6.5.3 and Subsection 6.5.4, where we need to run the job for certain time window.

We use `hashcat --session --restore` to resume cracking. If no name for the session is provided, Hashcat uses standard name `hashcat.restore` otherwise the restore file is named after the session name.

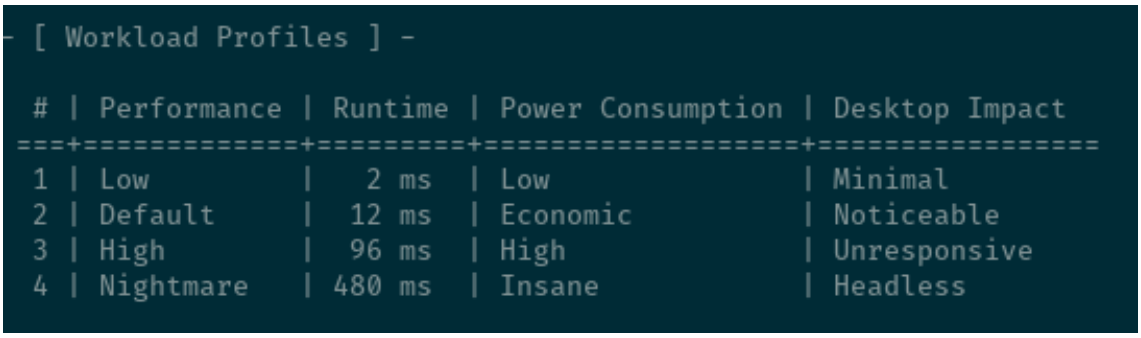



Figure 6.5: Hashcat workload options from official Hashcat help.

Hashcat has two main parameters, parameter a for attack type selection, where 0 corresponds to dictionary attack, and 3 that corresponds to a brute-force attack. The second parameter is `m`. Parameter `m` is the password hash type. Each password hash type is represented by the number mapped to the name of the hash type.

Currently, we are interested in a dictionary attack and brute-force attack. These attacks are called using these commands:

1. **Dictionary attack** - `hashcat -m 0 -a 0 md5.hash /path/to/dictionary`.
2. **Brute-force attack** - `hashcat -m 0 -a 3 1a1dc91c907325c69271ddf0c944bc72`.

### 6.4.2 John the Ripper in detail

John the Ripper has a fairly similar feature set and commands as Hashcat. However John the Ripper still comes with few differences. It does not provide machine-readable output thus we must parse the standard output. The restore file for John the Ripper is stored in `john.rec` file by default, but we can specify the exact name and location of the file. John the Ripper does not allow to change the workload profile and utilize all CPUs to the maximum. However as well as Hashcat, John the Ripper offer us to restore the cracking, which is crucial in the implementation of our scheduling algorithms.

We use two different attacks types for John the Ripper:

1. **Dictionary attack** - `john md5.hash --wordlist=/path/to/wordlist`.
2. **Brute-force attack** - `john md5.hash`.

Notice, that unlike Hashcat, we do not need to specify (but we can) the file hash type that should be cracked since John the Ripper is able to detect the format by itself.

### 6.4.3 Cracker

After we got familiar with all of the components we integrated all the essential pieces into a single Python script. The program starts by running `__main__.py`. The user has an option to specify several arguments

1. **Method_to_call** - the method we are interested in running.
2. **Sha256** - SHA256 of file.
3. **File_type** - the type of file.
4. **Timeout** - time after the container stops running.

5. **Session_name** - the name of the session, usually the same as SHA256.
6. **Task_in_progress** or dictionary - last parameter specifies either job in progress if we restore the cracking or custom words to try as dictionary.

The Python script operates as follows. First it downloads file based on its SHA256 from S3. Then we use XtoHashcat to identify the file password hash and persist the hash to file. In the case of WinZip and other archive formats the hash can take up to 1 MB size and immense string length, thus it is not feasible to pass it as an argument to shell call which is why we pass the file path with stored file hash in it as argument instead. Next, Hashcat or John the Ripper is called. Both Hashcat and John the Ripper calls spawn a shell subprocess and read the standard output and standard input of invoked script and act according to script output. For both cracking tools there are four possible cracking attempt outcomes:

1. **Cracked** - File is cracked.
2. **Timeout** - Cracking was ended due to a timeout.
3. **Fail** - Cracking failed due to hardware overheat, bad hash etc.
4. **Exhausted** - Unsuccessful cracking, we explored the whole keyspace.

The result is reported via standard output.

### 6.4.4 All wrapped to Docker

Finally we can wrap the final product into a Docker base image. As mentioned before in Subsection 6.1.2, our two cracking tools can run only in one instance in the OS. However we built a Docker image with both cracking tools, effectively allowing us to isolate their processes into standalone containers, which can be run in multiple instances concurrently on a single machine. The Dockerfile consists of our Python script, newest Hashcat, and John the Ripper binaries which are built inside the Dockerfile and supporting libraries for all the programs mentioned above. There are also two dictionaries included in the image, RockYou and CrackStation-human-only. Our base image is based on `nvidia/opencl:runtime-ubuntu18.04` parent image. It is an image based on the Linux operating system Ubuntu.

Listing 6.1: Example command for dictionary attack that runs for 120 seconds on GPU 2 using argument `--gpus-device`.

```
docker run --rm --env-file=secrets.env --gpus device=2 --label
    sha256=0337A97307D771C717075C595C3EB4DE3801FF127F2D193B042CE
    857F1EA99EF password-cracking-service:latest hashcat_
    dictionary_attack 0337A97307D771C717075C595C3EB4DE3801FF127F2
    D193B042CE857F1EA99EF doc 120 0337A97307D771C717075C595C3EB4
    DE3801FF127F2D193B042CE857F1EA99EF rock_you
```

## 6.5 Scheduler

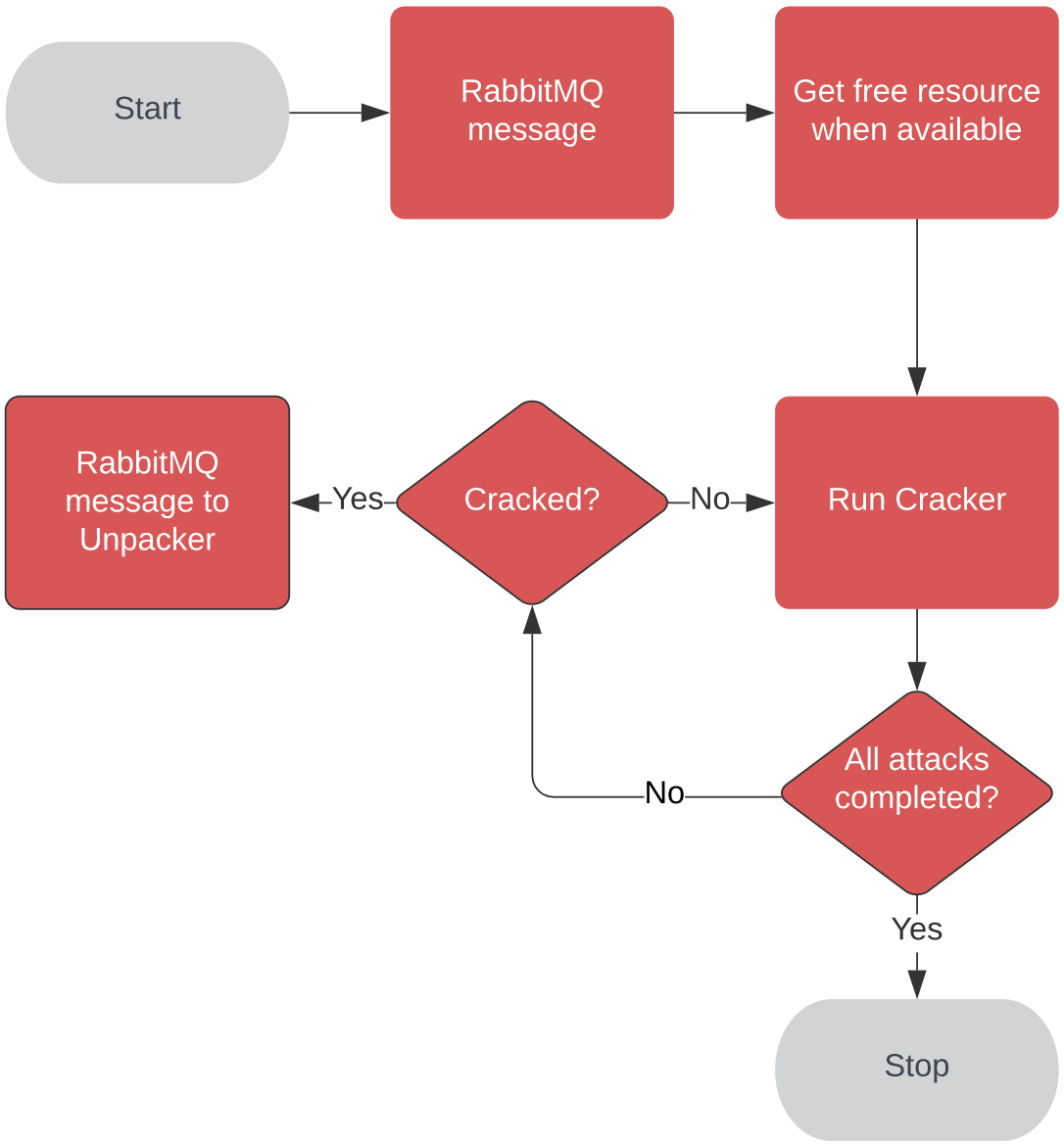



Figure 6.6: General workflow of Scheduler.

The *Scheduler* is responsible for distributing the job to the free resources, check the job status by tracking output from a running *Cracker* container, receiving jobs from *Shovel*, sending jobs to *Unpacker*, and storing the generated data. All this with different scheduling algorithms. We use *Docker Compose* to run all necessary images. The images are: *RabbitMQ* server, *Redis* database, *Celery* broker, *Celery* workers, and *Flower* - monitoring system for *Celery*.

### 6.5.1 Scheduling implementations

For each scheduling algorithm we use 3 types of attack and the cracking job use all of them if the given the limit is not exceeded. First we try a very low number of suggested words, these might be taken from the e-mail body or suggested by the user. Then we attempt a dictionary attack using the famous RockYou password list, and only if any

of the former is unsuccessful, we resort to a brute-force attack for a fixed time limit as described in Section 4.6. We believe that majority of passwords of our interest can be cracked since the attackers are oftentimes neglectful and do not use strong passwords as demonstrated by Hýža (2014).

### 6.5.2 First Come First Served algorithm

At the very beginning we created a Proof of Concept (PoC) to make a working solution of *Scheduler*. As a scheduling algorithm, we choose FCFS - where jobs are run one after another as soon as the machine is free.

When a job from RabbitMQ is acquired we send it to the available worker using Celery. The Scheduler starts the cracking attempt for a given cracking tool with the first cracking scenario, suggested passwords, then we try the RockYou dictionary and if not successful, we run a brute-force attack for the rest of the time.

Each job is run from Scheduler broker container in own separated Cracker container called via command mentioned in Listing 6.1.

Unfortunately, this scheduling algorithm occupies the resource for a long period of time with a single job only, thus we decided to preemptively run the jobs, since there is a chance to clear out easy to process jobs.

### 6.5.3 Round-Robin algorithm

We decided to implement the Round-Robin algorithm, where we do not use all scenarios at once for each file but we divide the cracking scenarios into separated cracking jobs. Thus if we start cracking $N$ jobs, all jobs come through suggested password cracking, then switch to dictionary attack and finish with brute-force attack.

Given a brute-force attack (and rarely a dictionary attack) may take a long time we do not make this algorithm preemptive only in terms of dividing the cracking of file to its own each scenario but we employ a fixed time window duration for each job run. The time window is determined to be 120 seconds - the minimum time for Hashcat to save the existing cracking progress. Thus we need an option to pause and resume the cracking. This is fortunately implemented in Hashcat and John the Ripper as described in Subsection 6.4.1 and Subsection 6.4.2.

To run the container for a fixed time we start the container and when there is no remaining time, we extract the restore (Hashcat) or rec (John the Ripper) file from container and store it to the database for later use. When the job is ready to be executed again, we download the restore file from database, create a new container with the restore file copied into its isolated filesystem.

We name the container after SHA256 of the file that is being cracked, allowing us to correctly address the specific container when controlling it.

To get the file from the container we use:

**Algorithm 2** Get the restore file from container.

**Input:** sha256 of file *sha*256,

1: docker cp container_id:/path/to/restore-file file ▷ Copy the restore file from the container to scheduler

2: save_to_db(sha256) ▷ Save it to the Cracking Service DB

To inject the file to a container we use the following code:

**Algorithm 3** Get restore file to rhe container.

**Input:** sha256 of file *sha*256,

1: docker create –name sha_name_tmp password-cracking-service:latest ▷ Create a new image from original image
2: file = get_from_db(sha256) ▷ Save from DB to file
3: docker cp file sha_name_tmp:/path/to/restore/file ▷ Save the restore file to proper position
4: docker commit sha_name_tmp sha_name ▷ Commit changes to container
5: docker rm sha_name_tmp ▷ Delete temporary container

Now, after we inject the restore file to the container, we can simply run the corresponding container created from Cracker with argument `restore_hashcat` or `restore_john`.

### 6.5.4 Dynamic Rebalancing algorithm

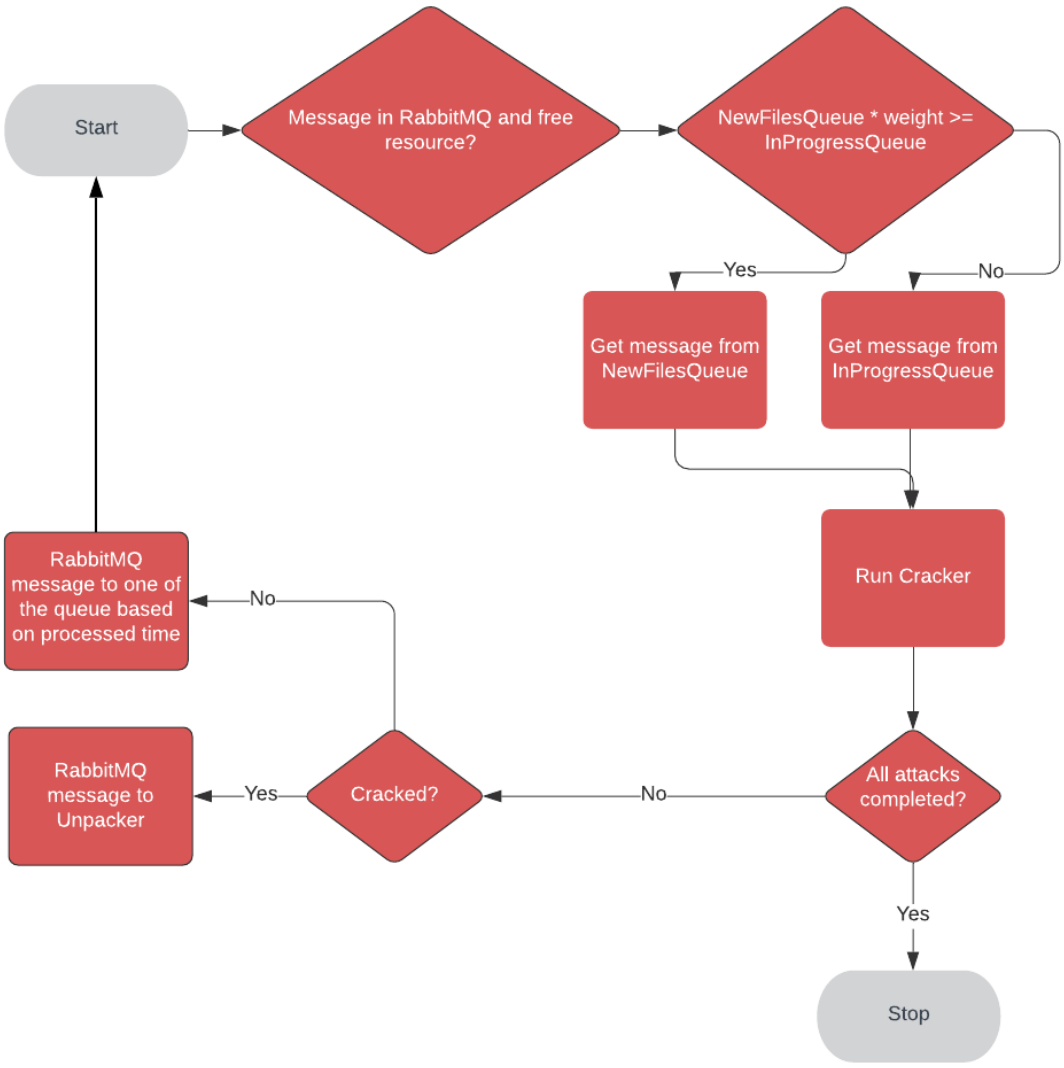



Figure 6.7: Dynamic Rebalancing algorithm workflow.

Since we want to crack new files in a timely fashion we decided to have two RabbitMQ queues to work with. This algorithm is based on Round-Robin implementation but the switch was added to move the job to another queue after a constant time period elapsed. One queue consists of new files called *New Files* and the second queue contains files that are being cracked for more than fixed allotted time, called *In Progress*. We decided to prioritize the new files since the cracking scenario starts from the most promising suggested passwords attack, to dictionary attack ending with a last-effort brute-force attack, each with increasing chance of success. By enforcing to get messages more from *New Files* queue, we process new files that might have a password lying in the keyspace of suggested passwords or our dictionary attack, while the processing of jobs from *In Progress* queue, where brute-force attack attempt occurs, is postponed. The decision when to take a job from *New Files* queue is based on the ratio of queues lengths, where the *New Files* queue is weighted by fixed constant, the tuning of weight constant is shown in Subsection 7.2.1.

## 6.6 Unpacker

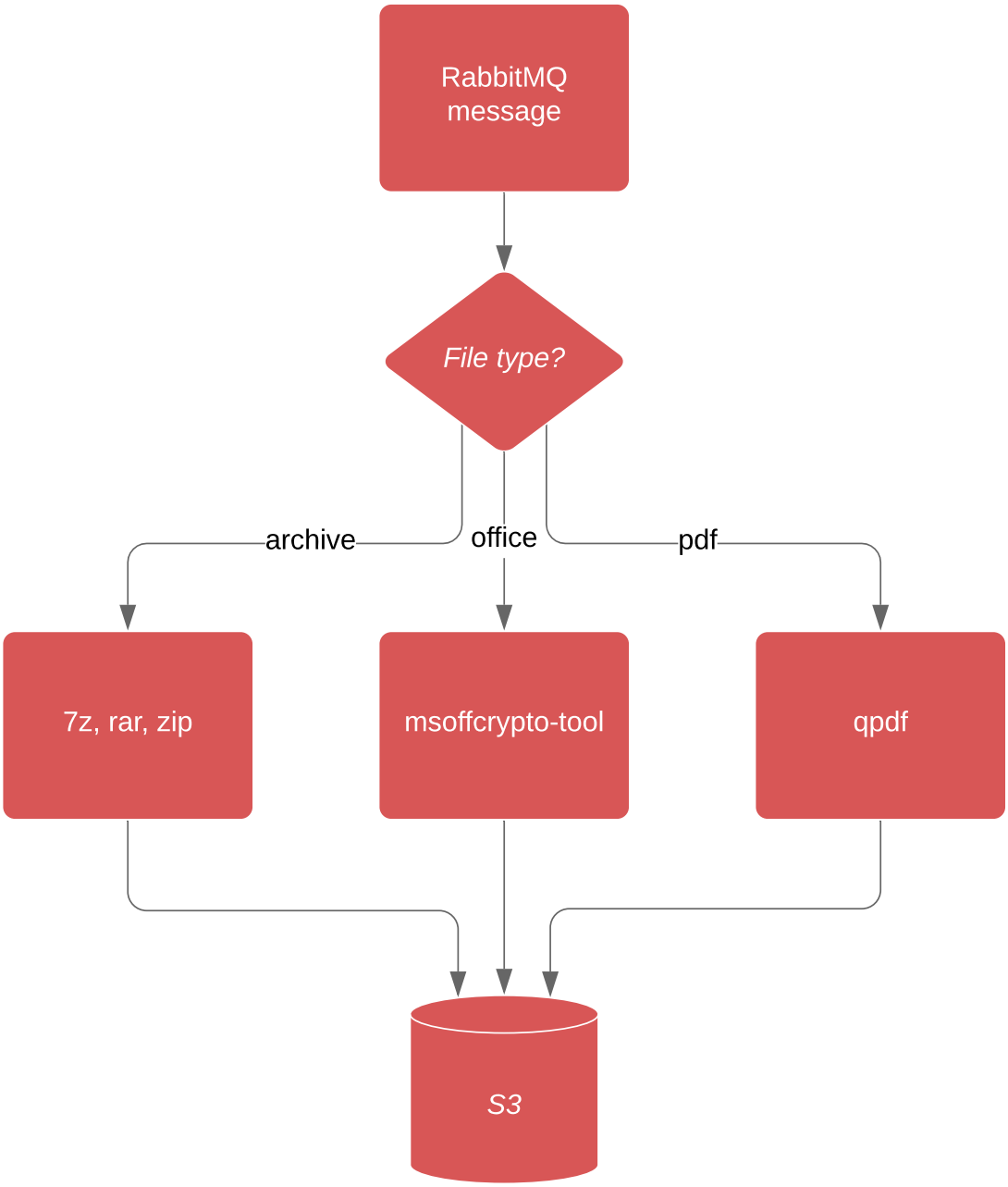



Figure 6.8: Unpacker service workflow.

Finally if the file is cracked, the message with the password is emitted to RabbitMQ, then Unpacker takes it from the queue and starts its job. The Unpacker is capable of using the password and extracting the files from 7Z, ZIP and RAR or retrieving a file without the password in case of PDF and Office documents. For a RAR file an `unrar` command-line tool exists. We used Linux tools `qpdf` (Berkenbilt, 2013) for PDF files, `zip` (Gordon, 2020) for ZIP files and `p7zip` (Trojette, 2016) for 7Z files, and `msoffcrypto-tool` (Nolze, 2020) for Office documents. These files are uploaded to S3 and processed by other systems.

# Chapter 7

# Evaluation

First, we show the speeds of cracking for file types of our interest in both Hashcat and John the Ripper. Then, we review the data that we work with in terms of file types and frequency. In Subsection 7.1.6 and Subsection 7.1.5, we show the worst-case and the best-case scenarios of password cracking using Scheduler. Later we determine the weight for the Dynamic Rebalancing algorithm which is later used in subsequent measurements. The performance of our algorithms is then shown and discussed on two artificial datasets made out of a subset corresponding to a real-life setting. The performance of our algorithms is compared with a Gurobi (Gurobi Optimization, 2015) model of non-preemptive implementation for comparison. In Subsection 7.2.4, we show the weakness of the FCFS algorithms non-preemptive approach in job distribution in comparison with Round-Robin and Dynamic Rebalancing algorithm where preemption is used.

Later on, we provide results of applying our methods to real data under a stress environment. We also discuss the results of cracking jobs.

During the testing, we experienced a GPU overheat, if the 7Z format or similar expensive jobs run at once. That resulted in making our Scheduler more resilient towards failures of hardware, as well as to connectivity issues. An interesting issue is also that protected (which means we can view source code of the file, but can not edit the PDF in any way) PDFs appear as encrypted, so the password hash is sent from Shovel to Scheduler that has to process it, but Hashcat returns empty password string.

## 7.1 Benchmark

The evaluation of our work is measured on 3x GeForce GTX Titan Z, where each GPU corresponds to 2 workers. This is due to the fact that the GPU has enough memory to run 2 jobs at once. The server is additionally equipped with an Intel(R) Core(TM) i7-3820 CPU @ 3.60GHz and 32GB DDR3 RAM and runs Centos 7.

The MILP model is solved using Gurobi Optimization (2015) solver computed on Dell Precision 3530 laptop equipped with 32 GB DDR4 RAM and Intel Core i7-8750H @ 2.20GHz.

### 7.1.1 Hashcat

We use the command `hashcat -m 0 -b`, where according to Hashcat (2020) parameter `-m` corresponds to a selected hash number and `hashcat -b` means *benchmark*. The computation is measured as if we tried to crack the file in brute-force fashion. The results are using unit H/s (hashes per s) with unit prefix in the metric system (mega, kilo).

Table 7.1: Speed of Hashcat.

| Format type number | Format name | Speed |
|---|---|---|
| 0 | MD5 | 6388.0 MH/s |
| 100 | SHA1 | 2185.4 MH/s |
| 1400 | SHA256 | 1071.9 MH/s |
| 10400 | PDF 1.1 - 1.3 (Acrobat 2 - 4) | 30371.6 kH/s |
| 10500 | PDF 1.4 - 1.6 (Acrobat 5 - 8) | 1501.8 kH/s |
| 10600 | PDF 1.7 Level 3 (Acrobat 9) | 1071.4 MH/s |
| 10700 | PDF 1.7 Level 8 (Acrobat 10 - 11) | 11814 H/s |
| 9400 | MS Office 2007 | 40598 H/s |
| 9500 | MS Office 2010 | 20294 H/s |
| 9600 | MS Office 2013 | 3231 H/s |
| 9700 | MS Office <- 2003 MD5 + RC4, oldoffice0, oldoffice1 | 21406.8 kH/s |
| 9800 | MS Office <- 2003 SHA1 + RC4, oldoffice3, oldoffice4 | 28276.5 kH/s |
| 11600 | 7-Zip | 126.5 kH/s |
| 13600 | WinZip | 888.9 kH/s |
| 12500 | RAR3-hp | 10286 H/s |
| 13000 | RAR5 | 13208 H/s |

### 7.1.2 John the Ripper

According to Openwall (1996) the metric c/s is called "crypts" (password hash or cipher computations) per second with unit prefix in the metric system (mega, kilo). Using argument `john --format=raw-md5 --test`, where we changed the format for each algorithm, we obtained:

Table 7.2: Speed of John the Ripper.

| Format name | Speed |
|---|---|
| MD5 | 43514 Kc/s |
| SHA256 | 2660 Kc/s |
| PDF | 10240 c/s |
| Office 2007/2010/2013 | 257 c/s |
| Zip, WinZip | 56219 c/s |
| RAR3 | 325 c/s |
| RAR5 | 691 c/s |

### 7.1.3 Comparison of cracking tools

As we can see, the Hashcat is superior to John the Ripper in terms of speed. This is due to the utilization of graphics cards. Hashcat, using graphics cards, is up to 100 times faster than John the Ripper. Thus we try to delegate all jobs to Hashcat, if the format is supported. The speed difference can be seen for example on the RAR3 and RAR5 formats, where Hashcat cracks RAR files with speed of 10286 H/s whereas John the Ripper cracks with the speed of 325 c/s. RAR5 is cracked by the speed of 13208 H/s by Hashcat and 691 c/s by John the Ripper. From the results of both benchmarks of cracking tools, we can see that the cracking of the new Office documents encryption from 2013 is much slower than the rest of the algorithms. Another example of a slow cracking are RAR files. These files are encrypted with AES 256, which was designed to make the encryption of one hash expensive, thus making cracking tools spend considerable resources to crack it.

### 7.1.4 Incoming data analysis

We receive approximately 30 encrypted files per hour. The Figure 7.1 shows that the most frequent encrypted format is PDF followed by Office documents. This data might be coming from phishing e-mails where viruses are often spread. Further, among the format types we distinguish for DOC files, PDF files and RAR files, we can see in Figure 7.4 that majority of our received files are legacy formats for both Microsoft Office documents and PDFs, which are main contributors to the encrypted file count, thus we should be able to crack them very fast. Based on the statistics of incoming files, we choose the maximum processing time $\overline{p}$ = 720 s, since we receive 30 files per hour on average, to 6 workers. Further in Subsection 7.1.5 we determine the value $\underline{p}$ = 15, since it is the mean of files computed in the best-case scenario. The time window for preemptive algorithms, RR and Dynamic Rebalancing is set to 120 s - the minimum time for Hashcat to save the current progress to restore file. In the case of Dynamic Rebalancing, the job is sent to the *In Progress* queue after $\overline{p}$ / 2 has elapsed. The values $\overline{p}$, $\underline{p}$, the constant for the job switch, and the time window hold for all the examples unless stated otherwise.

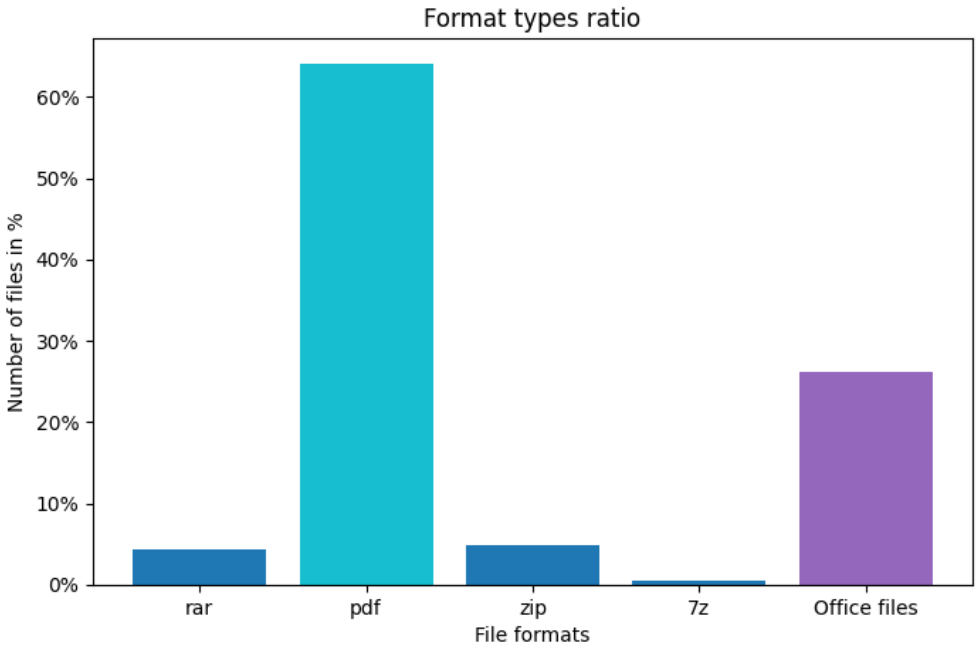



Figure 7.1: Format types ratio.

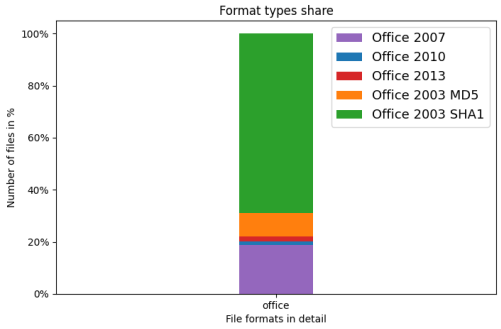



Figure 7.2: Office documents ratio in detail.

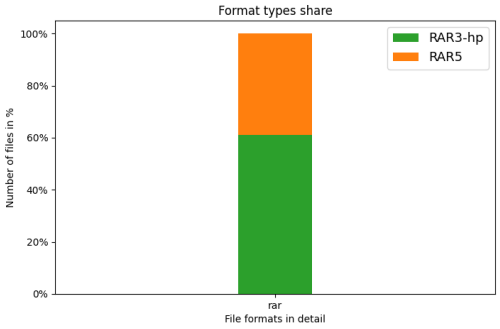



Figure 7.3: RAR format ration in detail.

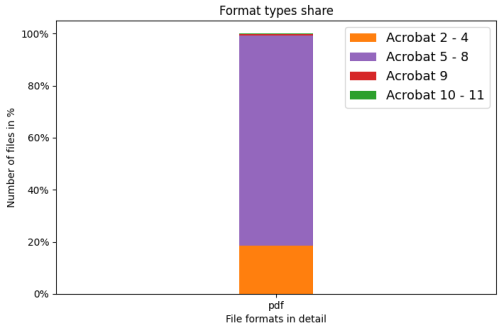



Figure 7.4: PDF format ration in detail.

### 7.1.5 The best-case benchmark

The best-case scenario occurs if we know the correct password for the file, so no other word candidates have to be tried, thus the minimum possible time should be required for password cracking.

Our implementation is capable of processing 1462 jobs in $C_{max}$ = 3596.97 s, whereas the MILP model, given the corresponding exact times for each job, gives optimum of $C_{max}$ = 3549.22 s and $C_{max}$ = 3660 s for robust model with $\underline{p}$ = 15. Since we provide only one password candidate and all jobs end after the password is correctly found, all algorithms behave in the same way and we compare the MILP-based solution with only one implementation. The corresponding Gurobi models and measurements can be found in `/evaluation/the_best_case/` folder.

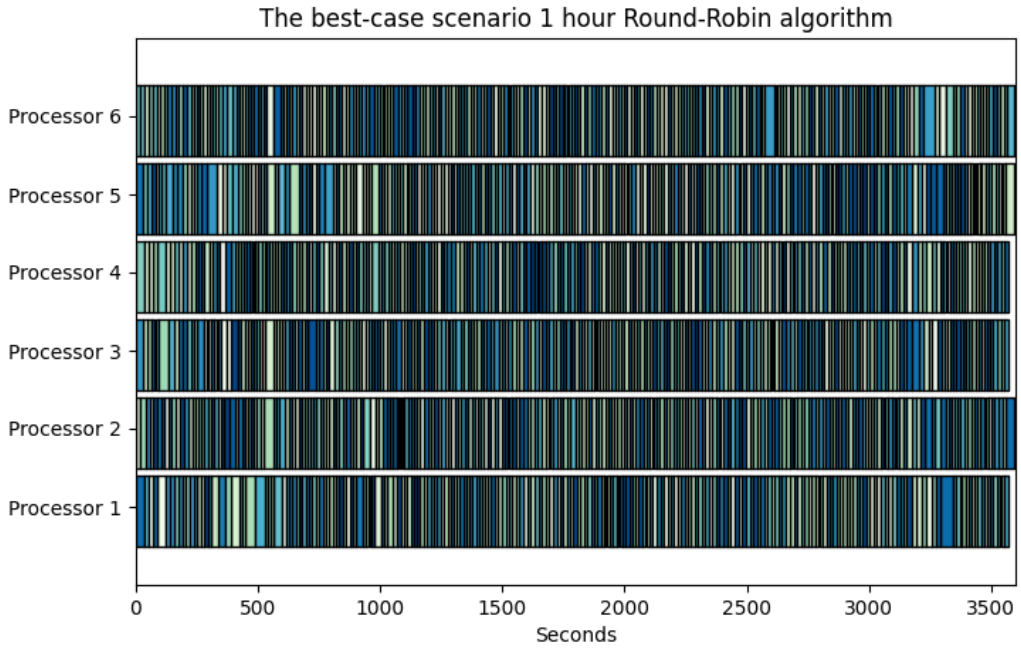



Figure 7.5: Schedule of best-case jobs.

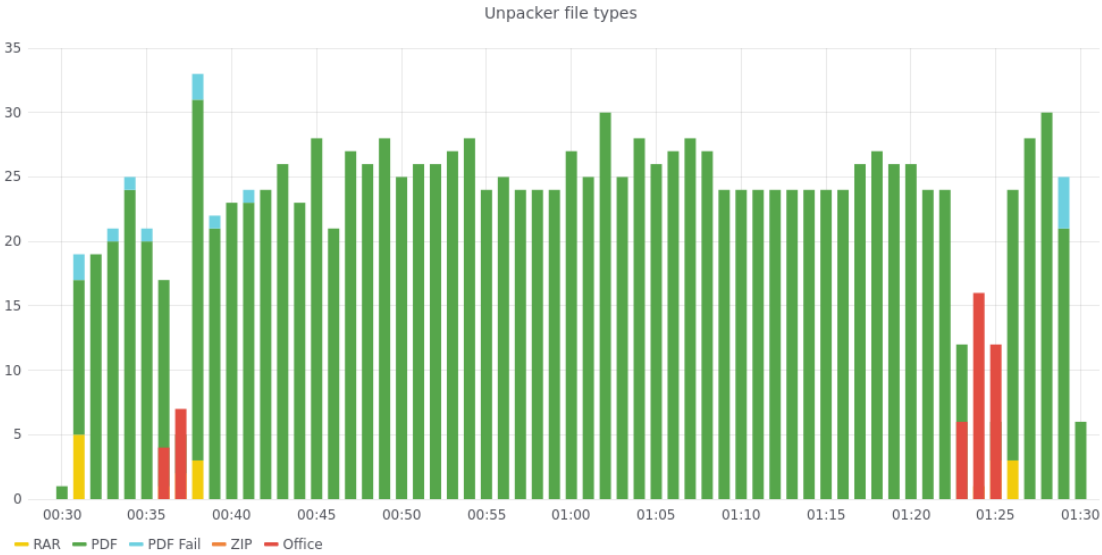
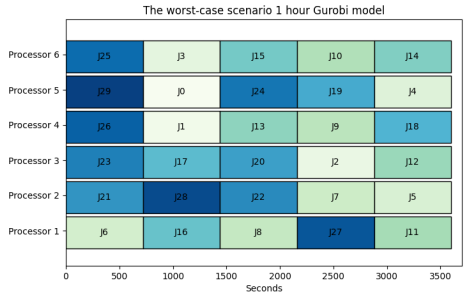


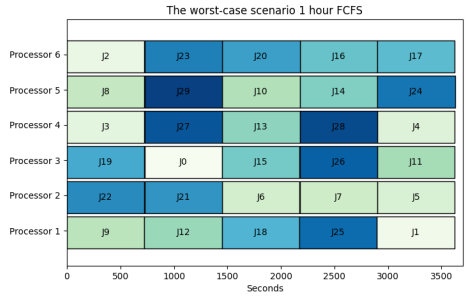

Figure 7.6: Unpacker work on cracked files visualized in Grafana.

### 7.1.6 The worst-case benchmark

The worst-case possibility is when all incoming data is encrypted with a strong password, thus we cannot decrypt the file and the entire time budget ($\overline{p}$) is spent on one file. The corresponding Gurobi model and data can be found in `/evaluation/the_worst_case/` folder. The colors and names represent the same job.

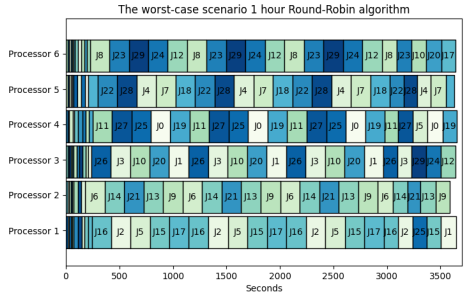



(a) The worst-case Gurobi schedule

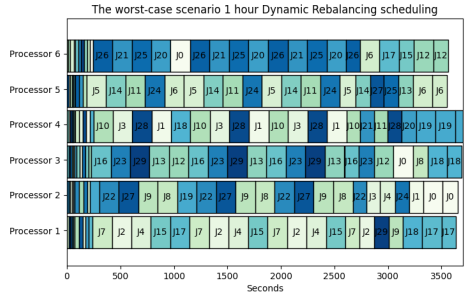



(b) The worst-case FCFS algorithm

Figure 7.7: Graph (a) depicts the optimal solution from Gurobi where the $C_{max}$ = 3600 s and picture (b) is our solution using First Come First Served algorithm. The graphs may appear similar, but the FCFS optimum is worse due to errors in measurements and burst time of each job making $C_{max}$ = 3626 s.




(a) The worst-case Round-Robin algorithm




(b) The worst-case Dynamic Rebalancing algorithm

Figure 7.8: Graph (a) depicts the solution presented by Round-Robin algorithm, where $C_{max}$ = 3649 s and picture (b) is our solution scheduled by Dynamic Rebalancing algorithm, where $C_{max}$ = 3703 s.

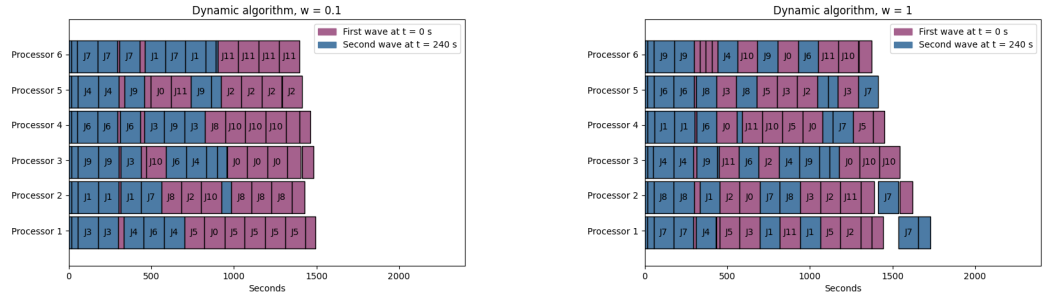
## 7.2 Artificial data

In this section, we evaluate our algorithms on artificial data created from the Files generator. We use mostly PDF and ZIP files since it is easy to generate them in any way we need, and we encounter these file types in real-world setting. We experiment with weight for the Dynamic Rebalancing algorithm as well as with the ratio of files which we are able to crack and which we are not able to crack.

### 7.2.1 Weight for the Dynamic Rebalancing algorithm

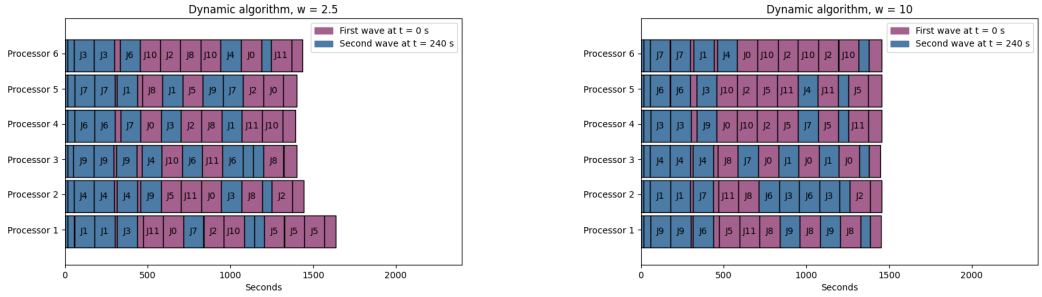
The Dynamic Rebalancing algorithm contains a constant weight. To show the behavior of Dynamic Rebalancing algorithm, we present a schedule where we used 4 different values for weight (0.1, 2.5, 5, 10). We simulated the behavior based on the value of constant weight by first sending 6 jobs to Scheduler, then waiting 4 minutes (which is the time when the jobs are moved from *New Files* queue to *In Progress* queue) and then sending another 6 jobs. The jobs in the first wave have a blue color, and jobs sent after 4 minutes have a purple color. The behavior is noticeable after the third column of jobs starts. If the weight is low, we consume only a few new jobs from the *New Files* queue and then we return to the *In Progress* queue, this holds for weight = 0.1 shown in Figure 7.9. The other extreme is weight = 10 depicted in Figure 7.10, where we prioritize the *New Files* queue heavily. We choose weight = 2.5 from Figure 7.10 as a compromise between these two extremes, where we prioritize the *New Files* queue but still use some resources to *In Progress* queue. The weight = 1 in Figure 7.9 shows the behavior in an unweighted case.

In further examples, we use weight = 2.5, as it is prioritizing the *New Files* queue but does not heavily limit the consumption of *In Progress* queue.




(a) w = 0.1 (b) w = 1

Figure 7.9: Dynamic Rebalancing algorithm.




(a) w = 2.5 (b) w = 10

Figure 7.10: Dynamic Rebalancing algorithm.

### 7.2.2 15/15 scenario

This artificial dataset has 15 files that we are able to crack and 15 files that we are not able to crack. Further, the dataset consists of 15 pdf files and 15 zip files. The dataset is made of 7 pdf files that are encrypted using 4 letters that are not part of the RockYou dictionary and 8 zip files encrypted using passwords from the RockYou dictionary. The remaining 15 files are encrypted using a 10-letter password which is uncrackable in the

allotted time frame, so the expected result is to successfully crack 15 out of the 30 files total.

First, we start the Scheduler and then we send the jobs all at once. This way we are able to compare the results with the schedule made by Gurobi and test the behavior during peak hours. For the Round-Robin and Dynamic Rebalancing algorithm, we have selected a time-window of 120 s.

We used the real processing time values computed from First Come First Served algorithm to non-preemptive implementation for Gurobi solver to see if it can be scheduled in a more simple way. We also computed the robust schedule for this given scenario. The optimal value for a robust non-preemptive schedule is $C_{max}$ = 2160 s, where R = 15, this value is the number of jobs that are expected to be not possible to crack, $\underline{p}$ = 15 s and $\overline{p}$ = 720 s. The schedule built from exact times from FCFS gives us the optimal value $C_{max}$ = 2171 s.

The best schedule was built by the RR algorithm, where the $C_{max}$ = 1982 s is shown in picture Figure 7.12. Our Dynamic Rebalancing algorithm in Figure 7.13 took $C_{max}$ = 2228 s and FCFS took $C_{max}$ = 2303 s as can be seen in Figure 7.11. We can see that the RR schedules the job equally and in a different order. The time window is set to 120 s - the minimum time for Hashcat to save the current progress to restore file. The order is determined by the order of messages in the queue, but not even the addition of more runs helped us to build a different schedule with better $C_{max}$. The colors and job names correspond to the same job in each graph. There is a trade-off between using the Dynamic Rebalancing algorithm, thus prioritizing the new files, and using RR where the load is equally distributed, thus obtaining a better schedule. The measurements can be found in `/evaluation/weight_parameter/` folder.

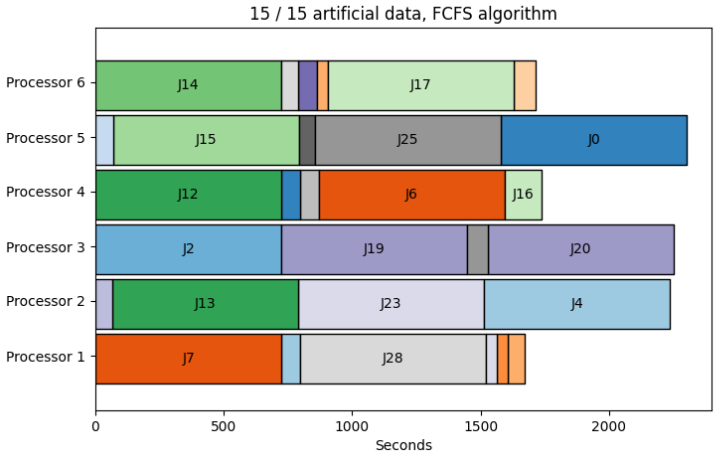



Figure 7.11: Schedule of FCFS algorithm 15/15 jobs took 2303 s.

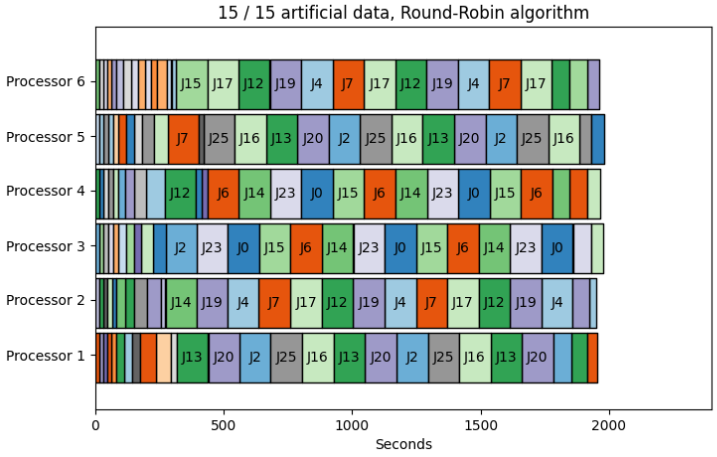



Figure 7.12: Schedule of Round-Robin algorithm 15/15 jobs took 1982 s.

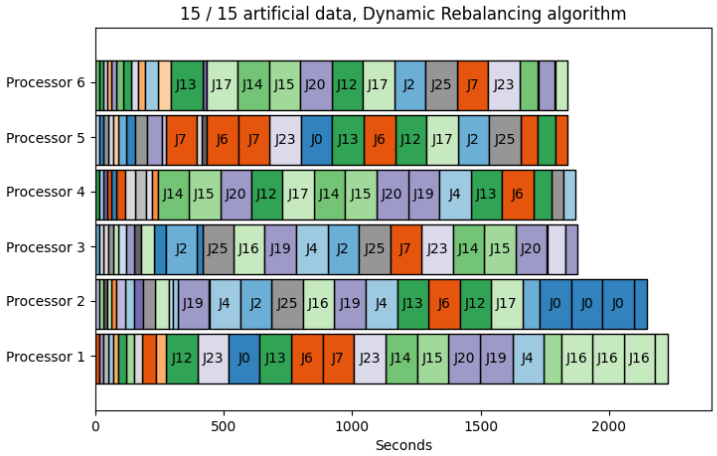



Figure 7.13: Schedule of Dynamic Rebalancing algorithm 15/15 jobs took 2228 s.

### 7.2.3 3/27 scenario

In this scenario, we created a dataset that has 3 files that we are able to crack. Each file is from our group of our attack (suggested password, dictionary attack, brute-force attack) which can be seen in Figure 7.14, where we have 3 distinct jobs with different completion times, in contrast with other jobs. This is due to the fact that the job with the suggested password was nearly instant, whereas the brute-force attack took the longest. As well as in Subsection 7.2.2, the RR algorithm creates a better schedule with $C_{max}$ = 3315 s, however without the prioritizing of new files. A slightly worse result of $C_{max}$ = 3460 s is achieved by the Dynamic Rebalancing algorithm and FCFS results in $C_{max}$ = 3644 s. The robust MILP optimum value $C_{max}$ = 3600 s and the exact solution constructed from data provided by FCFS gives $C_{max}$ = 3619 s. The corresponding Gurobi models and measurements can be found in `/evaluation/artificial_data_3_27/` folder.

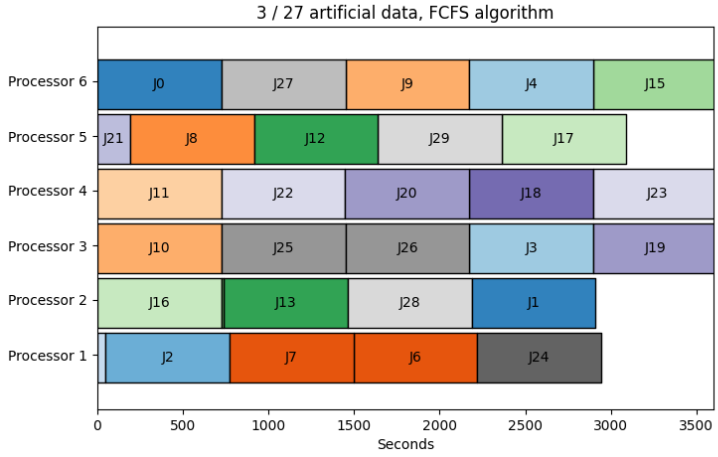



Figure 7.14: Schedule of FCFS algorithm 3/27 jobs took 3644 s.

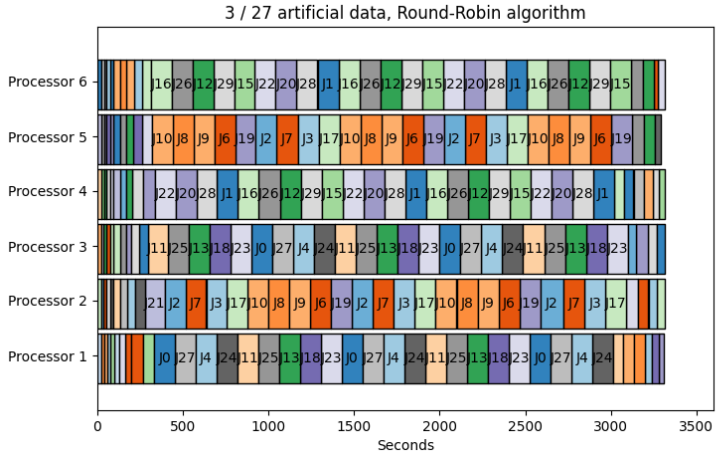



Figure 7.15: Schedule of Round-Robin algorithm 3/27 jobs took 3315 s.

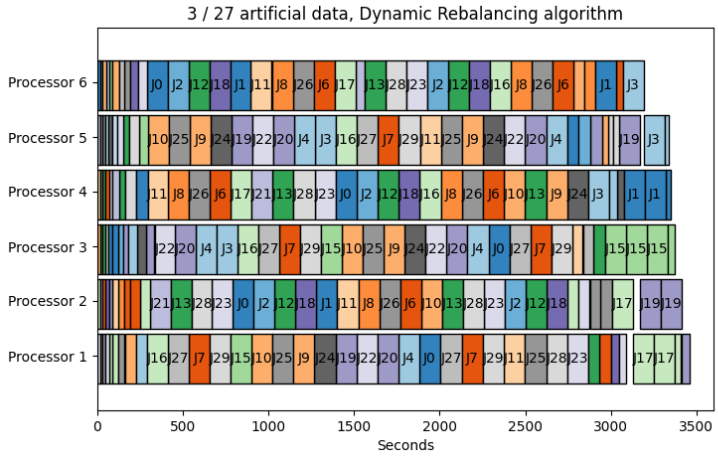



Figure 7.16: Schedule of Dynamic Rebalancing algorithm 3/27 jobs took 3315 s.

### 7.2.4 First Come First Served vs. Round-Robin like implementations

As mentioned in the last paragraph in Subsection 6.5.2, FCFS implementation blocks the processing unit with only one job and does not give another job the space to be solved. Beside our RR and Dynamic Rebalancing implementations are capable of making the schedule more equally distributed thus minimizing the maximal makespan, it comes with one more benefit. Given a scenario, where we have 7 files, 6 hard to crack and 1 easy to crack the FCFS will block the cracking through the stage where the 1 file could be cracked. RR like implementations do not suffer from this because we do all scenarios for each file at once. We want to show the comparison between FCFS and RR algorithms between the first successfully cracked file by this example. The corresponding Gurobi models and measurements can be found in `/evaluation/fcfs_vs_rr/` folder.

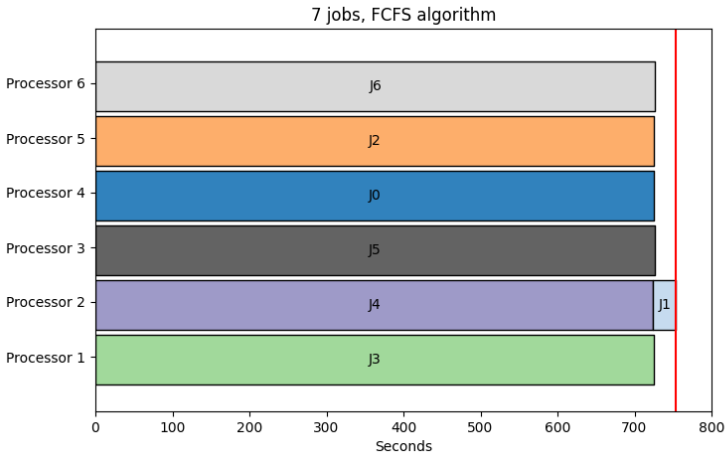



Figure 7.17: FCFS algorithm 7 jobs.

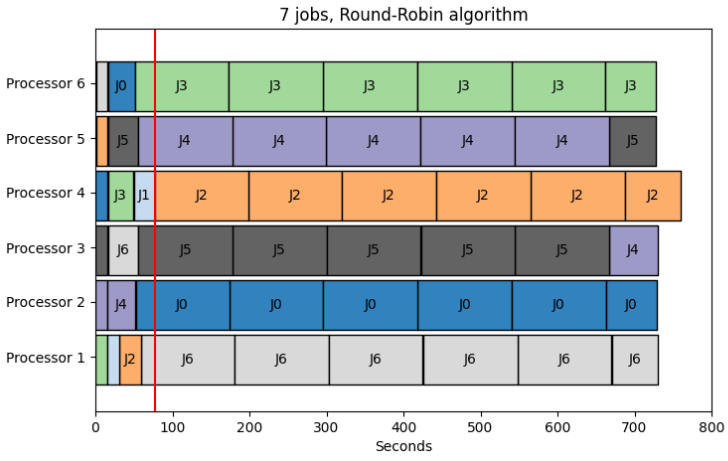



Figure 7.18: Round-Robin algorithm 7 jobs.

Since 6 jobs in Figure 7.17 occupy all the processing units in FCFS implementation, the job $J_1$ is cracked after 754 s from the start. However using RR implementation we crack the file after 78 s. Both these results are highlighted with the red line.

## 7.3 Real data

With real data, we do not know the original password to the given file. However, during our research, we processed over 25 thousand files and found the passwords to more than two thousand of them. When working with real data, we discovered many corrupted files. For example, PDF files can return a password hash even though it is not encrypted by password. The PDF is only protected from editing, which is enough for `pdf2john` to return some kind of password hash, that is later cracked resulting in an empty password. Also misspecification of a file format from the provider can happen, or the files are simply corrupted. It is also possible to extract a hash from encrypted archive divided into parts, however, we are not capable of extracting it without previous parts. We tried to enhance the cracking tactics by using a rule-based attack combined with RockYou dictionary, but we did not obtain any new passwords and outside of the scope of the normal RockYou dictionary. The rule-based attack also caused an inability to crack the files in time with the current hardware setup, since it overheats when the load is high. For this reason, we were unable to perform the experiment correctly.

### 7.3.1 Hand-picked real data

We created a dataset of 29 real files. It consists of 7 files that we are able to crack, 6 files which we are not able to crack, 3 files which has bad generated hash and 13 files with an empty password.

The R for our robust MILP is determined as follows:

1. Calculate the 1st and 3rd quartiles q1 and
2. Calculate the interquartile range IQR = q3 - q1.
3. Select x = 1.5 IQR, as is common when detecting distant values using boxplots.
4. Now the R is the number of values bigger than x.

This results x = 376, which results R = 7 and the optimal value of this schedule provided by a robust Gurobi solution is $C_{max}$ = 1440 s. The optimal value for this dataset is $C_{max}$ = 936 s provided by Dynamic Rebalancing algorithm followed by RR with $C_{max}$ = 959 s and FCFS, where $C_{max}$ = 1276 s. The results can be found in `/evaluation/real_data_29/` folder together with *Jupyter notebook* (Kluyver et al., 2016) where the cut-off value is determined.

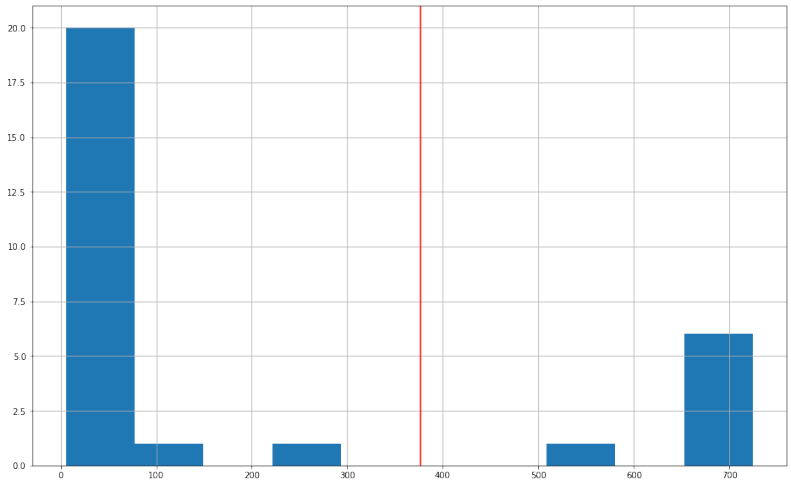



Figure 7.19: Cut-off, where the red-lines values is x = 376.

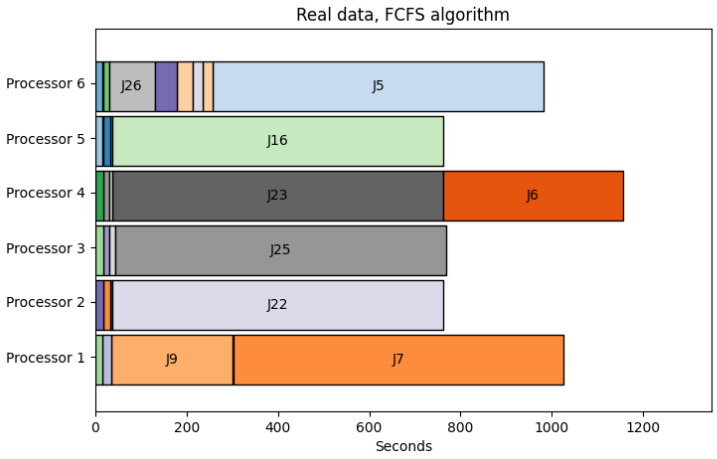



Figure 7.20: Real data, FCFS algorithm.

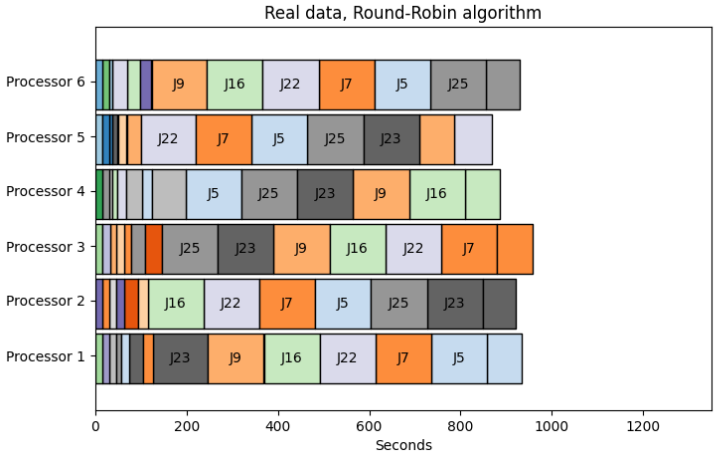



Figure 7.21: Real data, Round-Robin algorithm.

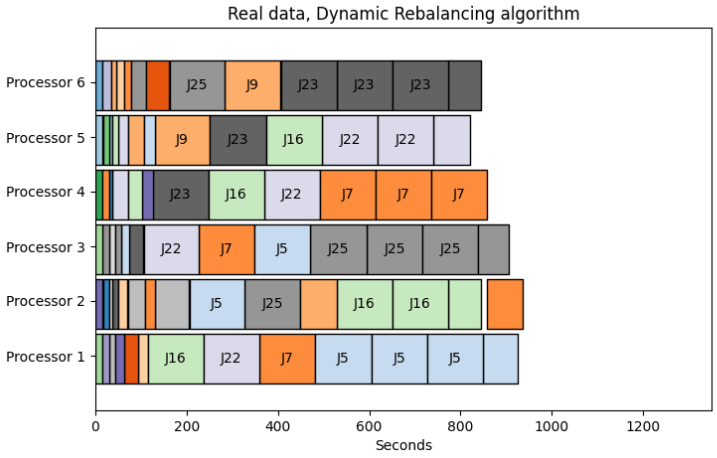



Figure 7.22: Real data, Dynamic Rebalancing algorithm.

### 7.3.2 Data in real-time

Here we recorded incoming files for approximately 30 minutes of incoming traffic and replay 30 minutes for each algorithm to compare its performance. As well as in previous real data measurement, the Dynamic Rebalancing algorithm has the shortest $C_{max} = 2012$ s followed by RR implementation with $C_{max} = 2046$ s, and finally, the FCFS algorithm ends after $C_{max} = 2174$ s. The measurements can be found in `/evaluation/data_in_real_time` folder.

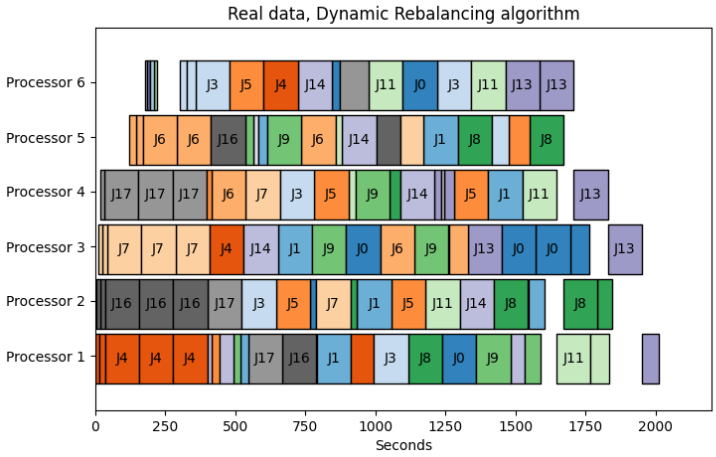



Figure 7.23: Real-time schedule of 30 minutes of jobs, FCFS algorithm.

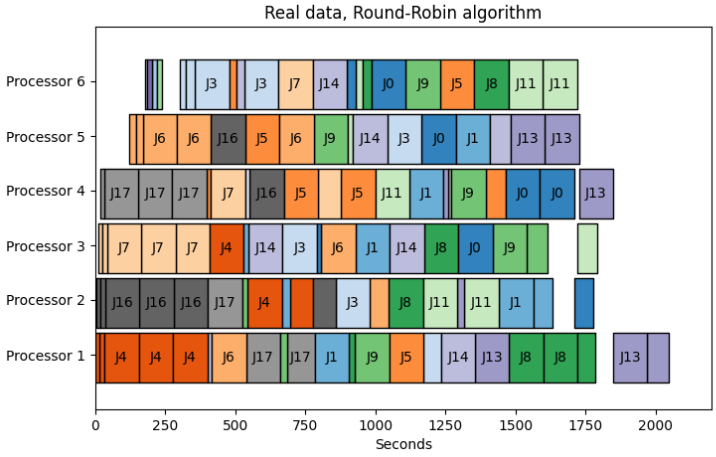



Figure 7.24: Real-time schedule of 30 minutes of jobs, Round-Robin algorithm.

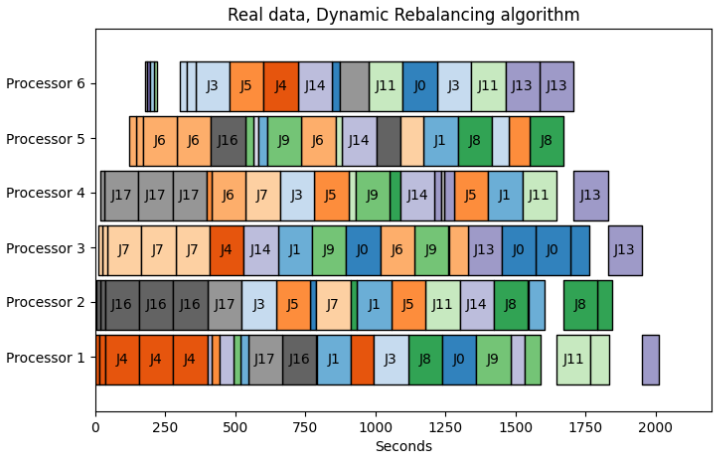



Figure 7.25: Real-time schedule of 30 minutes of jobs, Dynamic Rebalancing algorithm.

### 7.3.3 Peak measurement

Since it performed the best on real-life data, we use our Dynamic Rebalancing algorithm in the production setting. Further, we show, how Dynamic Rebalancing algorithm handles 3 hours of traffic with the weight = 2.5. The reason behind the selection of this particular window is being able to demonstrate it in on paper. We used weight = 2.5, $\overline{p}$ = 720 s and, job switch at $\overline{p}$ / 2 s, and 120 s time window. With these constants we can guarantee to process the files in real-time.

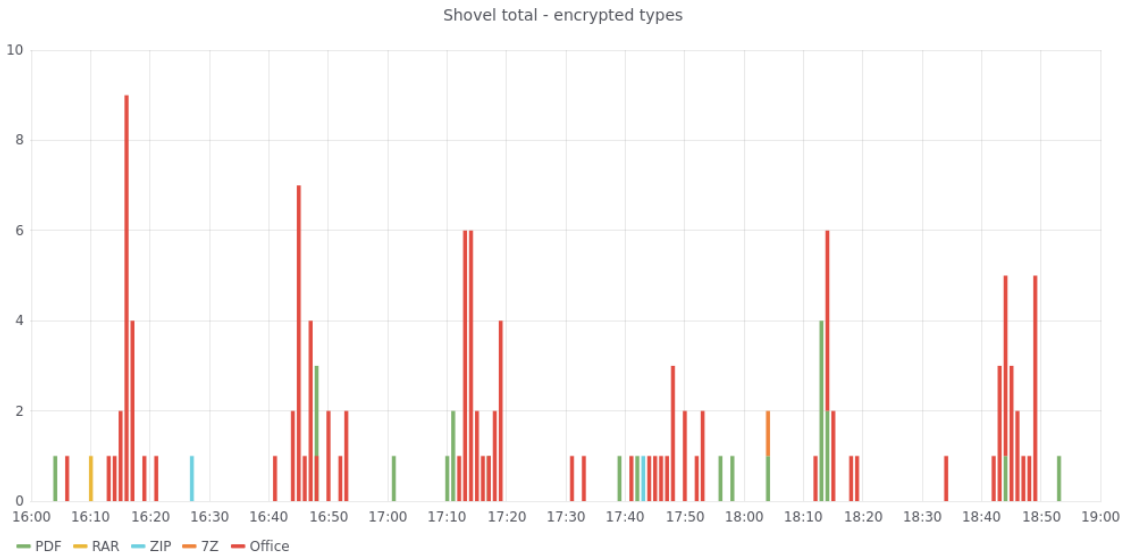



Figure 7.26: Visualisation of incoming encrypted files to Shovel.

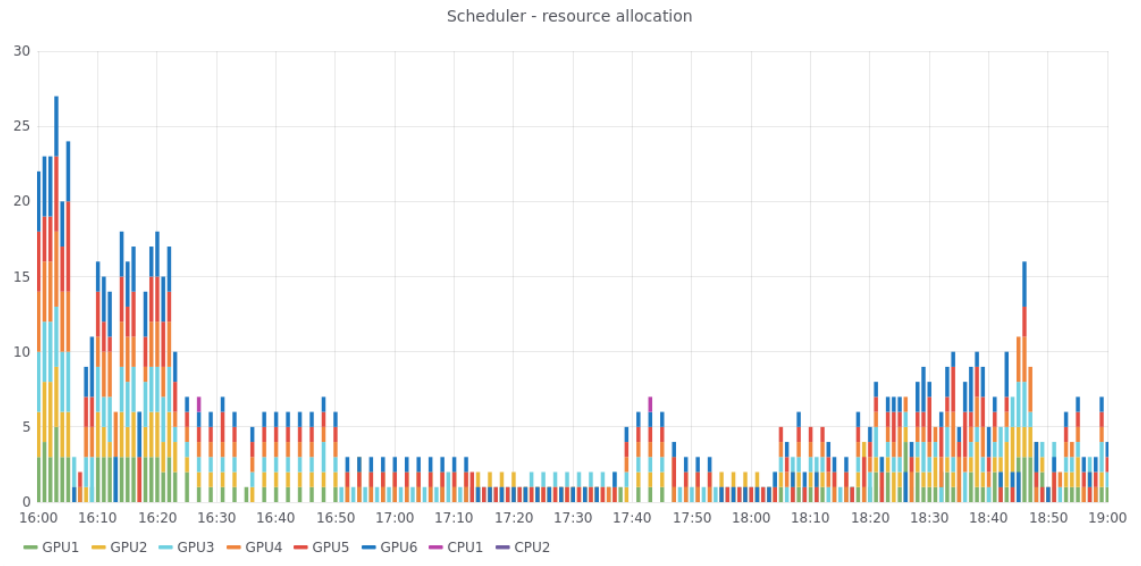



Figure 7.27: This graph demonstrates the resource allocation during password cracking. The bar chart on axis y shows the number of jobs per minute. It can be seen that 16:00 to 16:20 a lot of files came, thus a high number of suggested attacks were performed. These attacks were followed by fixed-window attacks of the RockYou dictionary attack and a brute-force attack. The frequency of allocated resources has decreased till 18:10. At 18:10, another wave of new files arrived, namely PDFs containing empty passwords, These files were cracked fast and as a result, the number of allocated resources raised.

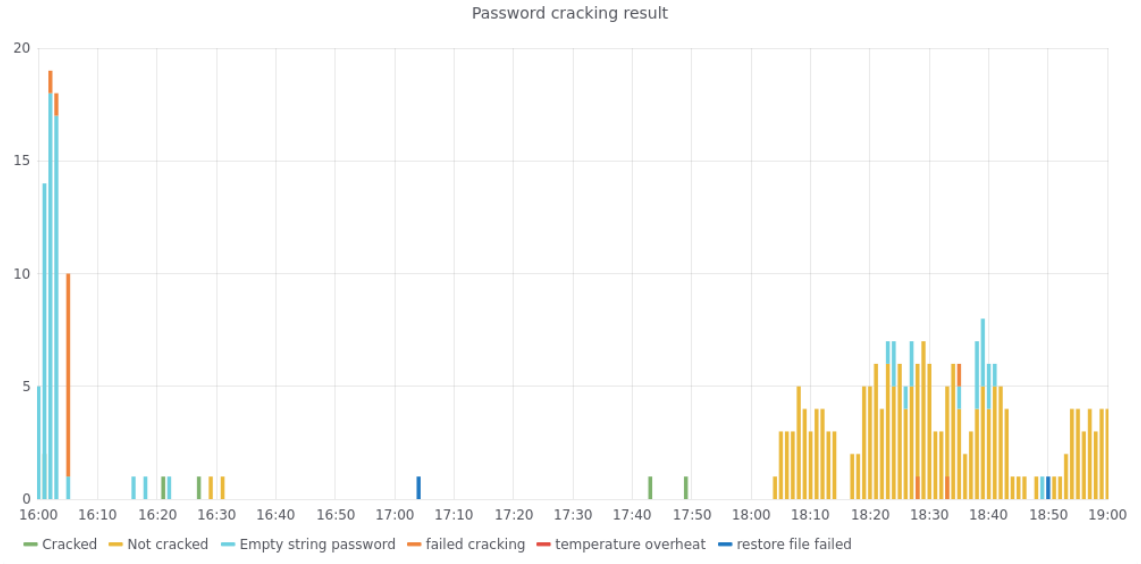



Figure 7.28: During this 3-hour session, we first detected the empty passwords, usually in PDF files. We also experienced a failed cracking, caused by XtoHashcat misclassification of the format type, resulting in failed start of Hashcat. After iterating over the suggested passwords attacks, we moved to cracking the files till 18:10 where we completed most of the cracking jobs and also cleared out the incoming empty hashes.

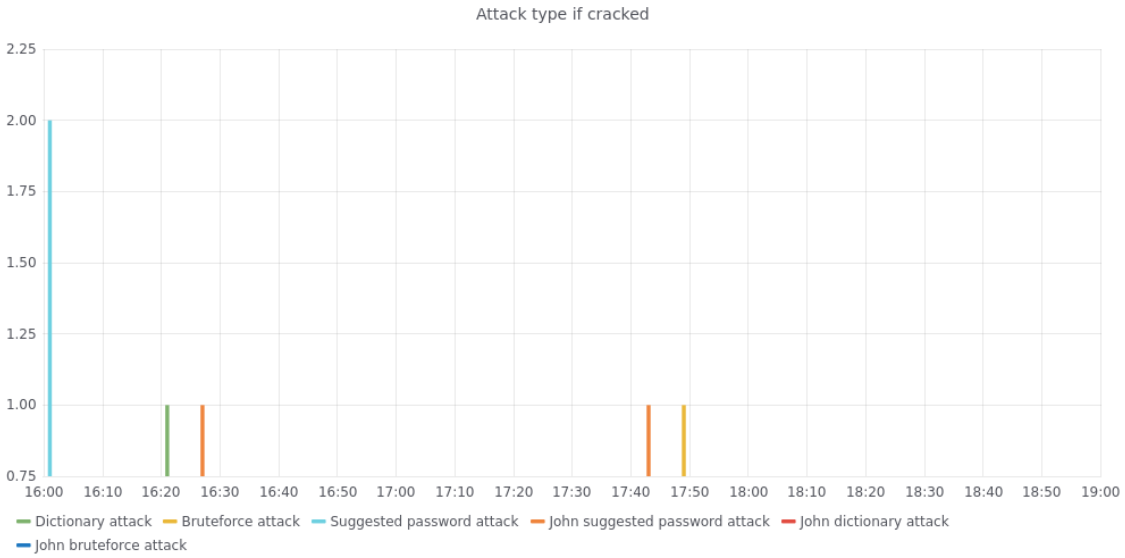



Figure 7.29: These are the results for successful password cracking attempts.

### 7.3.4 Analysis of cracked passwords from real data

We present the top 10 passwords obtained during our creation of our thesis. Interestingly, password *4534* was used to encrypt PDF files that contains malware so it might be most likely the favourite password among hackers. Next top malware passwords are *apr13*, *apr17* and *apr20* probably referring to month and number, however, these files were sent in different dates than April. Other favourite passwords are: *1qazzaq1*, *neo*, *Corona*, *virus*, *windows*, *Password1*, *qwer1234*.

The majority of our obtained passwords consist of numbers. The knowledge of passwords might be used to tag the virus according to its password.

From the total of 25812 files we were able to crack 5650 files from which 3932 are empty string passwords and 1718 files are truly encrypted which yields 6 % of cracked encrypted files. Most of the passwords that we cracked are easy to crack since they are made using only numbers. From the 1718 passwords 1583 are purely digits, 100 are only letters and 36 are a combination of both or use special characters.

| | Top 10 passwords |
|---|---|
| 1 | 1234 |
| 2 | 4532 |
| 3 | 123 |
| 4 | encrypted |
| 5 | infected |
| 6 | apr17 |
| 7 | view |
| 8 | 12345 |
| 9 | 1111 |
| 10 | 2222 |

### 7.3.5 Experiments conclusion

We tested our implementations on both artificial and real data. First, we tested the maximal throughput of our implementation as well as the behavior when the jobs used the maximum allocated time. These experiments helped us to determine the parameters $\underline{p}$

and $\overline{p}$. From the experiment in Subsection 7.2.1 we have chosen the constant weight later used in Dynamic Rebalancing algorithm.

The majority of the data we receive (Figure 7.4) is encrypted using legacy formats. Due to this fact we are able to search the passwords key space fast and even a brute-force attacks succeeds up to 6-letter passwords.

From the results of the experiments presented above, we can see that the RR algorithm exhibits the best on artificial data, since it equally distributes jobs to processing units. When we used the real-life data, the Dynamic Rebalancing algorithm outperforms Round-Robin and gives us a benefit of processing new incoming files in addition. Both RR and Dynamic Rebalancing algorithm allows us to process easy jobs and reduce the size of the queue as shown in Subsection 7.2.4 and Figure 7.28. The FCFS algorithm blocks the processing units for a long period of time and does not allow us to distribute the work more equally.

The top 10 passwords are not surprising as these passwords are similar to the top 10 worst passwords used by the users (Devillers, 2010), and hackers tend to use average passwords (Hýža, 2014).

# Chapter 8

# Conclusion and future work

This thesis presents the problem of cracking an incoming stream of encrypted files. To process the incoming files in a reasonable time, a proper scheduling algorithm had to be chosen, and a processing pipeline of microservices had to be created. To enable encrypted file detection and to read the stream of files, we created the microservice called Shovel. Shovel, in addition to its previously mentioned functions, adds prior knowledge, if available, about the particular file. Such information is e.g., the e-mail attachment, which is used to support the cracking process. The act of recovering the passwords is managed by Cracker, which wraps the functionality of both Hashcat and John the Ripper cracking tools in order to make it easier to call and control them. The Scheduler is responsible for distributing jobs to free resources, checking the job status by tracking output from a running Cracker container, receiving jobs from Shovel, sending jobs to Unpacker, and storing the generated data. First, we created Scheduler that used First Come First Served algorithm. Later we relaxed the task and allowed preemption, using Round-Robin algorithm, thus we are not stuck on one file for long period of time. Finally, we improved the Round-Robin algorithm to use two queues and called it Dynamic Rebalancing algorithm. We prioritize the queue with new files thus having a flow where new files are processed while we do not stuck on cracking that is not likely to succeed.

We tested these algorithms on both artificial and real data corresponding to real-life setting to choose the best candidate for production environment. The outcome of this thesis allows malware analysts to handle previously unseen files and make relations between newly obtained files. During the implementation of the thesis, we managed to process over 25 thousand files and find passwords to more than two thousand of them.

We have made an application that is currently run in production setting, which we can now tune as suggested in future work. Our implementation does not need any MILP solver and yet is still quite efficient. MILP solvers are expensive for business entities and our solution performs just as well.

In future work, we plan to increase the server capacity, thus allowing the extension of dedicated time for each job. Improving that capacity is a desirable step, as it enables the use of more complex password recovery strategies such as using rules-attack or masks attack. Detecting an empty string password of PDF files prior the cracking would also be beneficial for future applications. A drift to the use of PRINCE (PRobability INfinite Chained Elements) algorithm, created by the author of Hashcat (Steube, 2018), which combines password candidates from a dictionary to generate new password candidates, has the potential of adapting faster to changing conditions and to discover unseen passwords. The next logical step is to improve XtoHashcat to distinguish PKZIP2 format, since new Hashcat in beta version supports it and will eliminate the need to use John the Ripper for these files, allowing us to crack PKZIP2 faster.

# Bibliography


Adobe. Securing pdfs with passwords, adobe acrobat, 2020. URL https://helpx.adobe.com/acrobat/using/securing-pdfs-passwords.html.

S. Al-Kuwari, J. H. Davenport, and R. J. Bradford. Cryptographic hash functions: Recent design trends and security notions. Cryptology ePrint Archive, Report 2011/565, 2011. https://eprint.iacr.org/2011/565.

S. Altmeyer, S. M. Sundharam, and N. Navet. The case for fifo real-time scheduling. Technical report, University of Luxembourg, 2016.

E. Amazon. Amazon web services. *Available in: http://aws. amazon. com/es/ec2/(November 2012)*, 2015.

D. P. Anderson. Boinc: A system for public-resource computing and storage. In *proceedings of the 5th IEEE/ACM International Workshop on Grid Computing*, pages 4–10. IEEE Computer Society, 2004.

D. P. Anderson, J. Cobb, E. Korpela, M. Lebofsky, and D. Werthimer. Seti@ home: an experiment in public-resource computing. *Communications of the ACM*, 45(11):56–61, 2002.

Avast. 83% of americans are using weak passwords, 2019. URL https://press.avast.com/83-of-americans-are-using-weak-passwords.

S. Banon. Kibana: Explore, visualize, discover data, 2020. URL https://www.elastic.co/kibana.

J. Berkenbilt. Qpdf: A content-preserving pdf transformation system. *URL http://qpdf. sourceforge. net*, 2013.

Celery. Distributed task queue, 2020. URL http://www.celeryproject.org/.

G. Chen and Z.-J. M. Shen. Probabilistic asymptotic analysis of stochastic online scheduling problems. *IIE Transactions*, 39(5):525–538, 2007. doi: 10.1080/07408170600941623. URL https://doi.org/10.1080/07408170600941623.

H. Chen, F. Wang, N. Helian, and G. Akanmu. User-priority guided min-min scheduling algorithm for load balancing in cloud computing. In *2013 national conference on parallel computing technologies (PARCOMPTECH)*, pages 1–8. IEEE, 2013.

M. Crypto. File:sha-1.svg, 2006. URL https://commons.wikimedia.org/wiki/File:SHA-1.svg.

M. Crypto. File:md5 algorithm.svg, 2007. URL https://commons.wikimedia.org/wiki/File:MD5_algorithm.svg.

Curlyboi. curlyboi/hashtopus, Apr 2020. URL https://github.com/curlyboi/hashtopus.

T. M. Damico. A brief history of cryptography. *Inquiries Journal*, 1(11), 2009.

A. R. Dash, S. K. Samantra, et al. An optimized round robin cpu scheduling algorithm with dynamic time quantum. *arXiv preprint arXiv:1605.00362*, 2016.

M. M. Devillers. Analyzing password strength. *Radboud University Nijmegen, Tech. Rep*, 2, 2010.

M. J. Dworkin. SHA-3 standard: Permutation-based hash and extendable-output functions. *Federal Inf. Process. Stds.(NIST FIPS)-202*, 2015.

K. Etminani and M. Naghibzadeh. A min-min max-min selective algorihtm for grid task scheduling. In *2007 3rd IEEE/IFIP International Conference in Central Asia on Internet*, pages 1–7. IEEE, 2007.

S. Fluhrer, I. Mantin, and A. Shamir. Weaknesses in the key scheduling algorithm of rc4. In *International Workshop on Selected Areas in Cryptography*, pages 1–24. Springer, 2001.

E. Gordon. Zip home page, 2020. URL http://infozip.sourceforge.net/.

Grafana. grafana/grafana, May 2020. URL https://github.com/grafana/grafana.

I. Gurobi Optimization. Gurobi optimizer reference manual. *URL http://www. gurobi. com*, 2015.

N. Hart, A. Swift, and D. Higgins. Hashcat can now crack an eight-character windows ntlm password hash in under 2.5 hours., Feb 2019. URL https://www.informationsecuritybuzz.com/expert-comments/hashcat-can-now-crack-an-eight-character/.

Hashcat. hashcat advanced password recovery, 2020. URL https://hashcat.net/wiki/doku.php?id=frequently_asked_questions#how_can_i_perform_a_benchmark.

Y. S. Hong, J. No, and S. Kim. Dns-based load balancing in distributed web-server systems. In *The Fourth IEEE Workshop on Software Technologies for Future Embedded and Ubiquitous Systems, and the Second International Workshop on Collaborative Computing, Integration, and Assurance (SEUS-WCCIA'06)*, pages 4–pp. IEEE, 2006.

T. Hornby. Free password hash cracker, 2011. URL https://crackstation.net/.

R. Hranickỳ, M. Holkovič, and P. Matoušek. On efficiency of distributed password recovery. *Journal of Digital Forensics, Security and Law*, 11(2):5, 2016.

R. Hranickỳ, L. Zobal, O. Ryšavỳ, and D. Kolář. Distributed password cracking with boinc and hashcat. *Digital Investigation*, 30:161–172, 2019.

T. Hunt. Have i been pwned. *Last retrieved*, 23, 2019.

T. Hýža. Are hackers' passwords stronger than regular passwords?, 2014. URL https://blog.avast.com/2014/06/09/are-hackers-passwords-stronger-than-regular-passwords/.

Kaliski. The md2 message-digest algorithm, 1992. URL https://tools.ietf.org/html/rfc1319.

T. Kluyver, B. Ragan-Kelley, F. Pérez, B. Granger, M. Bussonnier, J. Frederic, K. Kelley, J. Hamrick, J. Grout, S. Corlay, P. Ivanov, D. Avila, S. Abdalla, and C. Willing. Jupyter notebooks – a publishing format for reproducible computational workflows. In F. Loizides and B. Schmidt, editors, *Positioning and Power in Academic Publishing: Players, Agents and Agendas*, pages 87 – 90. IOS Press, 2016.

W.-C. Lee, M.-C. Chuang, and W.-C. Yeh. Uniform parallel-machine scheduling to minimize makespan with position-based learning curves. *Computers & Industrial Engineering*, 63(4):813–818, 2012.

J. Li, M. Qiu, J.-W. Niu, Y. Chen, and Z. Ming. Adaptive resource allocation for preemptable jobs in cloud systems. In *2010 10th International Conference on Intelligent Systems Design and Applications*, pages 31–36. IEEE, 2010.

R. Lim. Parallelization of john the ripper (jtr) using mpi. *Nebraska: University of Nebraska*, 37, 2004.

M. Liu, X. Liu, F. Chu, F. Zheng, and C. Chu. Service-oriented robust parallel machine scheduling. *International Journal of Production Research*, 57(12):3814–3830, 2019.

N. Megow and T. Vredeveld. Approximation in preemptive stochastic online scheduling. In *European Symposium on Algorithms*, pages 516–527. Springer, 2006.

D. Merkel. Docker: lightweight linux containers for consistent development and deployment. *Linux journal*, 2014(239):2, 2014.

L. P. Michael. *Scheduling: theory, algorithms, and systems*. Springer, 2018.

D. Microsoft. Password must meet complexity requirements (windows 10) - windows security, Aug 2017. URL `https://docs.microsoft.com/en-us/windows/security/threat-protection/security-policy-settings/password-must-meet-complexity-requirements`.

R. Mohanty, H. S. Behera, K. Patwari, M. Dash, and M. L. Prasanna. Priority based dynamic round robin (pbdrr) algorithm with intelligent time slice for soft real time systems. *arXiv preprint arXiv:1105.1736*, 2011.

R. Morris and K. Thompson. Password security: A case history. *Communications of the ACM*, 22(11):594–597, 1979.

NIST. Nist special publication 800-63b, 2017. URL `https://pages.nist.gov/800-63-3/sp800-63b.html`.

Nolze. nolze/msoffcrypto-tool, Apr 2020. URL `https://github.com/nolze/msoffcrypto-tool`.

NVIDIA. Nvidia container runtime, Jun 2018. URL `https://developer.nvidia.com/nvidia-container-runtime`.

Openwall. John the ripper password cracker, 1996. URL `https://www.openwall.com/john/`.

Openwall. Parallel and distributed processing with john the ripper, Feb 2014. URL `https://openwall.info/wiki/john/parallelization`.

I. Pavlov, 2020. URL `https://www.7-zip.org/7z.html`.

A. Pippin, B. Hall, and W. Chen. Parallelization of john the ripper using mpi. *linux. kiev. ua*, 2006.

S. Pivotal. Rabbitmq, 2020. URL https://www.rabbitmq.com/getstarted.html.

G. PostgreSQL Global Development. The world's most advanced open source relational database, 1996. URL https://www.postgresql.org/.

V. Rijmen and J. Daemen. Advanced encryption standard. *Proceedings of Federal Information Processing Standards Publications, National Institute of Standards and Technology*, pages 19–22, 2001.

R. L. Rivest. The rc4 encryption algorithm. rsa data security. *Inc., March*, 12:9–2, 1992.

E. Roshal. Winrar encryption frequently asked question (faq), 2020. URL https://www.win-rar.com/encryption-faq.html.

G. Rossum. Python reference manual. 1995.

s3inlc. s3inlc/hashtopolis, May 2020. URL https://github.com/s3inlc/hashtopolis.

P. Salot. A survey of various scheduling algorithm in cloud computing environment. *International Journal of Research in Engineering and Technology*, 2(2):131–135, 2013.

S. Sanfilippo, 2020. URL https://redis.io/.

security Skull. Passwords, 2020. URL https://wiki.skullsecurity.org/Passwords.

K.-K. Seo and B. Do Chung. Robust optimization for identical parallel machine scheduling with uncertain processing time. *Journal of Advanced Mechanical Design, Systems, and Manufacturing*, 8(2):JAMDSM0015–JAMDSM0015, 2014.

J. Steube. Hashcat, 2009.

J. Steube. hashcat/princeprocessor, Jul 2018. URL https://github.com/hashcat/princeprocessor.

M. Stevens, E. Bursztein, P. Karpman, A. Albertini, Y. Markov, A. P. Bianco, and C. Baisse. Announcing the first sha1 collision. *Google Security Blog*, 2017.

Terahash. The inmanis™, 2020. URL https://terahash.com/.

H. Topcuoglu, S. Hariri, and M.-Y. Wu. Task scheduling algorithms for heterogeneous processors. In *Proceedings. Eighth Heterogeneous Computing Workshop (HCW'99)*, pages 3–14. IEEE, 1999.

M. A. Trojette. Welcome to the p7zip home, 2016. URL http://p7zip.sourceforge.net/.

A. P. Vestjens. Scheduling uniform machines on-line requires nondecreasing speed ratios. *Mathematical programming*, 82(1-2):225–234, 1998.

W. Wang and G. Casale. Evaluating weighted round robin load balancing for cloud web services. In *2014 16th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing*, pages 393–400. IEEE, 2014.

X. Wang, D. Feng, X. Lai, and H. Yu. Collisions for hash functions md4, md5, haval-128 and ripemd. *IACR Cryptology ePrint Archive*, 2004:199, 2004.

J. Wen and D. Du. Preemptive on-line scheduling for two uniform processors. *Operations Research Letters*, 23(3-5):113–116, 1998.

WinZip. How strong is winzip's encryption?, 2020. URL https://support.winzip.com/hc/en-us/articles/115011455167-How-strong-is-WinZip-s-encryption-.

H. Wu. The misuse of rc4 in microsoft word and excel. *IACR Cryptology ePrint Archive*, 2005:7, 2005.

X. Xu, W. Cui, J. Lin, and Y. Qian. Robust makespan minimisation in identical parallel machine scheduling problem with interval data. *International Journal of Production Research*, 51(12):3532–3548, 2013.

R. K. Y. Yadav, A. Mishra, N. Prakash, and H. Sharma. An improved round robin scheduling algorithm for cpu scheduling. *International Journal on Computer Science and Engineering*, 2, 07 2010.

P. B. L. Zobal and R. Hranický. Distribuovaná obnova hesel s využitím nástroje hashcat, 2018.

# Appendix A

# Attached files

These are the attached files that are presented in attachments. The attachment structure is specified in Table A.1.

| File | Description |
|---|---|
| /evaluation/artificial_data_3_27/ | Artificial data measurement for 3/27 scenario |
| /evaluation/artificial_data_15_15/ | Artificial data measurement for 15/15 scenario |
| /evaluation/date_in_real_time/ | Real-time measurement of traffic during 30 minutes |
| /evaluation/fcfs_vs_rr/ | Comparison between FCFS and RR |
| /evaluation/real_data_29/ | Comparison of work on real data |
| /evaluation/the_best_case/ | Measurement of the best-case scenario |
| /evaluation/the_worst_case/ | Measurement of the worst-case scenario |
| /evaluation/weight_parameter/ | Tuning with the weight constant |
| /implementation/FCFS/ | FCFS implementation |
| /implementation/RR_and_Dynamic/ | RR and Dynamic Rebalancing implementation |
| /implementation/Shovel/ | Shovel implementation |
| /implementation/files_generator/ | Artificial data files generator |
| /implementation/Unpacker/ | Unpacker implementation |

Table A.1: List of attached files.