BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF INFORMATION TECHNOLOGY
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF INFORMATION SYSTEMS
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

OPTIMIZATION OF TASK DISTRIBUTION IN FITCRACK SYSTEM

OPTIMALIZACE DISTRIBUCE ÚLOH V SYSTÉMU FITCRACK

MASTER'S THESIS
DIPLOMOVÁ PRÁCE

AUTHOR **Bc. TOMÁŠ ŽENČÁK**
AUTOR PRÁCE

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

BRNO 2020

Vysoké učení technické v Brně
Fakulta informačních technologií

Ústav informačních systémů (UIFS) Akademický rok 2019/2020

Zadání diplomové práce

Student: **Ženčák Tomáš, Bc.**
Program: Informační technologie Obor: Bezpečnost informačních technologií
Název: **Optimalizace distribuce úloh v systému Fitcrack**
Optimization of Task Distribution in Fitcrack System
Kategorie: Paralelní a distribuované výpočty

Zadání:

1. Seznamte se s architekturou a implementací systému Fitcrack.
2. Analyzujte algoritmus adaptivního plánování a strategie použité pro distribuci jednotlivých typů úloh. Identifikujte kritická místa, která přidávají režii do výpočetního procesu.
3. Po konzultaci s vedoucím navrhněte zdokonalení současného způsobu distribuce úloh, aby došlo k lepšímu rozdělení práce mezi připojené uzly a minimalizaci související režie.
4. Navržení řešení implementujte.
5. Navrhněte sadu experimentů, které ověří přínos vašeho řešení oproti původnímu stavu.
6. Experimenty realizujte a zhodnoťte dosažené výsledky.

Literatura:

- HRANICKÝ Radek, ZOBAL Lukáš, RYŠAVÝ Ondřej and KOLÁŘ Dušan. Distributed Password Cracking with BOINC and Hashcat. *Digital Investigation*, vol. 2019, no. 30, pp. 161-172. ISSN 1742-2876.
- HRANICKÝ Radek, ZOBAL Lukáš, VEČEŘA Vojtěch a MÚČKA Matúš. The Architecture of Fitcrack Distributed Password Cracking System. FIT-TR-2018-03, Brno, 2018.
- Další dle dohody s vedoucím

Při obhajobě semestrální části projektu je požadováno:

- Body 1 až 3, část bodu 4

Podrobné závazné pokyny pro vypracování práce viz https://www.fit.vut.cz/study/theses/

Vedoucí práce: **Hranický Radek, Ing.**
Vedoucí ústavu: Kolář Dušan, doc. Dr. Ing.
Datum zadání: 1. listopadu 2019
Datum odevzdání: 20. května 2020
Datum schválení: 29. října 2019

Abstract

The goal of this thesis is the optimization of task distribution in the Fitcrack system. The improvement is reached by way of increasing the accuracy of the estimation of the computational power of worker nodes, and the prevention of the creation of extremely small tasks, as well as increasing the efficiency of the transfer of the tasks to the worker nodes. In this thesis, the current state of the Fitcrack system is described, tested, and evaluated. This thesis then describes the weak points of the current implementation, proposes ways of remediating them and describes, tests and evaluates the implementation of those proposals.

Abstrakt

Cílem této práce je optimalizace rozdělování a distribuce úloh v systému Fitcrack, a to jak pomocí zpřesnění odhadů výpočetního výkonu jednotlivých uzlů a zamezení zbytečnému omezování velikosti jednotlivých úloh, tak pomocí zefektivnění přenosu zadání výkonným uzlům. V této práci je popsán, otestován a zhodnocen současný stav systému Fitcrack, jsou popsány jeho nedostatky, a jsou navrženy, implementovány, otestovány a zhodnoceny postupy, jak tyto nedostatky odstranit.

Keywords

Hashcat, Fitcrack, password recovery, password cracking, distributed computing

Klíčová slova

Hashcat, Fitrcack, obnova hesel, lámání hesel, distribuované výpočty

Reference

ŽENČÁK, Tomáš. *Optimization of Task Distribution in Fitcrack System.* Brno, 2020. Master's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor Ing. Radek Hranický,

Optimization of Task Distribution in Fitcrack System

Declaration

Hereby I declare that this bachelor's thesis was prepared as an original author's work under the supervision of Ing. Radek Hranický from Faculty of Information Technology, Brno University of Technology. All the relevant information sources, which were used during preparation of this thesis, are properly cited and included in the list of references.

.....
Tomáš Ženčák
July 30, 2020

Acknowledgements

I would like to express my gratitude to Ing. Radek Hranický for his invaluable advice and guidance during the writing of this thesis, as well as for supplying me with sources of information relevant to its topic.

Contents

Chapter 1

Introduction

In today's world, it is becoming much easier to protect data with a password. Such protections enable individuals to keep their data confidential even in the event the device they are stored on is stolen or otherwise compromised. However criminals can also encrypt data that may otherwise be used by law enforcement agencies as evidence of their crimes. It is therefore sometimes desirable to break password protection. To do so, a great amount of computing resources are needed, and one way of cracking passwords is using a supercomputer. This is, however, very expensive. Another alternative is distributed computing, where the computing task is split into small pieces, and those pieces are then sent to many ordinary computers to solve [7].

Fitcrack is a software tool developed at FIT BUT which allows its users to leverage distributed computing for password cracking in a manner similar to projects such as SETI@home or Folding@home. It is based on the Berkeley Open Infrastructure for Network Computing (BOINC) and it uses an open-source tool called Hashcat to execute the actual password cracking [9].

The goal of this thesis is to optimize the way Fitcrack distributes tasks to participating computers. There are several areas in which Fitcrack's performance is lacking so far, such as the estimation of the appropriate size for the first chunks of work assigned to new computers and overlaying communication with computation. Fitcrack also splits the work into undesirably small chunks in the beginning and near the end of a cracking task.

This thesis analyzes these problems, proposes improvements, and describes the implementation of these improvements. It also experimentally verifies that the implemented changes have successfully improved the investigated characteristics of the system.

This thesis is structured as follows. Chapter 2 describes the field of password recovery, chapter 3 describes relevant parts of BOINC, and chapter 4 provides a high-level overview of the way Fitcrack works. Chapter 5 then describes some problems the current version of Fitcrack has, while chapter 6 describes ways I have proposed to fix these problems and chapter 7 discusses the implementation of these proposals. Chapter 8 describes experiments I have done to assess how the implementation of my proposals has affected the problematic aspects of Fitcrack.

Chapter 2

Password recovery

Password recovery, or alternatively password cracking, is the process of finding an unknown password in order to access data or systems protected by that password. This process can generally be described as the repetition of two distinct procedures, which should be independent of each other [11]:

- generating candidate passwords, and
- verifying the correctness of the generated passwords.

The algorithms used to implement the first procedures are, in Hashcat's terminology, called **attack modes** [1]. Probably the simplest example of the first procedure is the brute-force attack. In this attack mode, the attacker systematically generates all possible passwords. Another is the dictionary attack, in which the output is taken from a preexisting list of passwords. These are the most common attack modes [15], but not all of them. Other attack modes will be described in section 2.1. The attack mode can be chosen arbitrarily and does not depend on what the password protects.

The verification process, on the other hand, is dictated by the format chosen by the protected content or service, as the attacker must verify the correctness of a candidate password using the same process the legitimate application uses. For example, when a server needs to authenticate users using a password, the best practice is to only store a cryptographic hash of the password, and when a user attempts authentication, the server hashes the provided password and then compares the result with a hash it has stored in its user database [3]. If an attacker obtains this database, they must then verify each candidate password in the same way.

Another example is attempting to verify a password used to access encrypted data. This is typically done by using the password as input to a function which transforms it into a cryptographic key used in an encryption algorithm, decrypting (part of) the encrypted (meta)data using the generated key and checking whether the plaintext has the expected structure [14].

2.1 Attack modes

This section will describe the various attack modes, focusing on the attack modes supported by Hashcat and using Hashcat's terminology except where noted.

2.1.1 Dictionary attack

In this attack mode, the user supplies a list of candidate passwords to be tried. Optionally, some transformations (called „rules“) can be applied to each candidate password, and the results of these transformations can be used instead of the original password. This can increase the keyspace size. In Hashcat's terminology, this attack mode is called „Straight attack“.

2.1.2 Combination attack

This attack mode is like the dictionary attack, but there are two dictionaries, which I will call the „left“ and „right“ dictionary. Candidate passwords are formed by appending each password from the „right“ dictionary to each password from the „left“ dictionary. The size of the resulting keyspace is the product of the sizes of the keyspaces of the dictionaries. There can also be a single rule applied to each dictionary.

2.1.3 Mask attack

This attack mode generates passwords from a so-called „mask“. This is a string that works somewhat like a regular expression. It is composed of placeholders, which are either literal characters, or a charset variable. With a charset variable, that position of the generated string can contain any one character from a set of characters. The keyspace represented by a mask then contains all possible strings which match the mask. Hashcat also calls this attack mode the „Brute-force attack“ in some places, since it is fairly straightforward to use the mask attack to implement a brute-force attack and there are no real disadvantages [2].

2.1.4 Hybrid attack

This attack mode is a combination of the Combination attack and the mask attack. It works like the Combination attack, but one of the dictionaries is replaced by a mask.

2.2 Existing password cracking software

There are several software tools whose purpose is password cracking. Examples are John the Ripper[1] or Hashcat[2], which has been used to win the Crack Me If You Can contest several times.[3] Both of these are used only on a single computer, but there are also tools which leverage Hashcat to execute distributed password cracking using many computers, such as Hashtopolis[4] or Fitcrack,[5] which is the focus of this thesis.

[1] https://www.openwall.com/john/
[2] https://hashcat.net/hashcat/
[3] https://contest.korelogic.com/
[4] https://github.com/s3inlc/hashtopolis
[5] https://fitcrack.fit.vutbr.cz/

Chapter 3

BOINC

BOINC is an open-source framework for distributed computing using volunteered resources, whose development started at University of California, Berkeley [4]. It allows one to create a custom project by means of implementing several daemons and supplying a client-side application that carries out the computation itself. In this chapter, I will not describe the entirety of BOINC. Instead, this chapter is intended to be a high-level overview of BOINC as it is used by Fitcrack.

In BOINC, the basic unit of computation is called a workunit [4]. Each workunit is executed using a so-called application,[1] which is a collection of files along with a configuration file, which describes what to do with each file and identifies the program that carries out the computation itself. An app can have multiple versions as well as different implementations for different platforms. The app is common to multiple workunits and is not deleted from the client. Once an app is added to BOINC, it is immutable and if it is to be changed, a new version of it must be created.[2]

The workunit itself also has various files specific for it, generally containing the data to be processed in the given workunit. These are described using so-called templates, which specify the names that the files should be presented under on the client.[3] Unlike apps, which also specify the content of the constituent files, the templates only specify what the files will be called, and the contents must be specified separately during the creation of each workunit.[4]

3.1 Server-side subsystems

The server of a BOINC system generally runs at least the following daemons:

- Trickler, which receives so-called trickle messages from the client. These can be used to report progress, information about the current load or temperatures, and so on. If a project wants to send custom trickle messages, part of the daemon has to be implemented by the project in order to handle any trickle messages the project wants to use.[5]

[1] https://boinc.berkeley.edu/trac/wiki/JobIn#Workunitattributes
[2] https://boinc.berkeley.edu/trac/wiki/AppVersion
[3] https://boinc.berkeley.edu/trac/wiki/JobTemplates
[4] https://boinc.berkeley.edu/trac/wiki/JobSubmission
[5] https://boinc.berkeley.edu/trac/wiki/TrickleMessages

- Validator, which validates the results incoming from the clients. This is important as BOINC is meant to enable anybody to participate, which however enables bad actors to get work and send fabricated results, among other things. Validator can be used to prevent this. In order to work for that, the project needs to implement its own logic into it, but there is the option of using the default BOINC validator, which only checks the syntax of the results.[6]

- Assimilator, which processes the results incoming from the clients and modifies the database accordingly. Part of it has to be implemented by the project as BOINC is meant to handle arbitrary computation, and so there cannot be uniform handling of the results.[7]

- Scheduler, which handles some of the communication with the client computers. They send requests for work, and scheduler sends information about workunits to be processed in case they exist and the host does not already have enough workunits being processed. It is entirely implemented by BOINC.

- Feeder, which facilitates communication between the database and some other daemons. It is entirely implemented by BOINC.

- Transitioner, which handles state transitions of workunits. It is entirely implemented by BOINC.[8]

- File deleter, which periodically deletes files that were generated by the other daemons are no longer needed. The default BOINC implementation is often sufficient.[9]

- Generator, which was the main focus of this thesis, as it is responsible for creating the workunits which are later sent to clients. It is entirely implemented by the project.[10]

3.2 Client-side subsystems

The only program running on the client in every BOINC project is the BOINC client. It is responsible for communicating with the server, asking for workunits, and possibly controlling the execution of the project-specific program.[11] When a workunit is available and the client is not already processing more workunits than the server is configured to give to a client at a time, the server sends information about the workunit and the locations of the necessary files. The BOINC client then downloads the files and runs the one that is designated as the main executable in the configuration file of the workunit's app.

This can be an arbitrary program, but the BOINC project also contains libraries which can be used to communicate with the BOINC client and let it do things such as pausing or canceling the computation. They also enable sending data to the BOINC client, which can then pass them to the server. This allows one to do things like sending trickle messages or reporting the final computation result. The libraries can also, among other things, take care of cleanup such as killing all subprocesses when the main process is canceled. [12]

[6] https://boinc.berkeley.edu/trac/wiki/ValidationIntro
[7] https://boinc.berkeley.edu/trac/wiki/AssimilateIntro
[8] https://boinc.berkeley.edu/trac/wiki/BackendPrograms
[9] https://boinc.berkeley.edu/trac/wiki/FileDeleter
[10] https://boinc.berkeley.edu/trac/wiki/WorkGeneration
[11] https://boinc.berkeley.edu/wiki/BOINC_Client
[12] https://boinc.berkeley.edu/trac/wiki/BasicApi

BOINC supplies several different wrapper programs,[13] which use those libraries and can be used to execute various tasks, but if that is insufficient, a project can implement its own application to be run by the BOINC client.

[13] https://boinc.berkeley.edu/trac/wiki/WrapperApp

Chapter 4

The Fitcrack system

The Fitcrack system is a system built using BOINC in order to leverage distributed computing for password cracking. It consists of a standard BOINC project and a web application called Fitcrack WebAdmin which is used by the system's administrators to add work and control what gets done and by which computers. There is also an application called `runner`, which is the application executed by the BOINC client and serves to run and coordinate the other processes involved in the cracking process [6]. The following text describes the concepts relevant to this thesis as well as the high level overview of the way the Fitcrack-specific parts of the system work. These are my observations from the source code.

4.1 Overview of jobs

To organize the work, Fitcrack uses so-called jobs. Each job consists of hashes that the administrator wants to be cracked, attack settings which dictate how the system should get the candidate passwords, and split the work among clients, as well as the list of hosts assigned to execute the job. Once the job is set to run, workunits are generated for the assigned hosts. Fitcrack uses a mechanism offered by BOINC which allows the server to specify which host will execute a given workunit in order to generate workunits whose size is tailored to the computing power of the host for which the workunit is generated.

The power is estimated based on the size of previous workunits and the time taken to complete them. Of course, before the host completes any work, there is no power estimate, and so one must be obtained. To do this, the first workunit generated in a job for every host is a so-called benchmark workunit, which does not do any useful computation and merely estimates the host's computing power.

Each job can be in one of several states. Each job starts in the **Ready** state. This state signifies that the job is ready to run. This is also the state in which a job is when it's paused. In case something goes wrong with the job, it is put into the **Malformed** state. The administrator can try to restart the job if this happens. The **Running** state signifies that workunits are currently being generated for the job. When transitioning from the Running state to any state other than the Malformed state, in which case the transition happens instantaneously, the job is first put into the **Finishing** state, which means that new workunits are no longer being generated, but there are still generated workunits which have not been processed by the client computers. From that state, depending on the reason for cessation of workunit generation, the job can transition into one of three states: If the generation has ceased because the job's configured end time has passed, the job transitions

into the **Timeout** state. Otherwise if the job was paused by the system administrator, the job is put back into the Ready state. If neither of the previous is true, either all hashes in the job have been cracked, in which case the job transitions into the **Finished** state, or the job's keyspace has been processed in its entirety without finding all the passwords. In that case, if at least one password has been found, the job is also put into the **Finished** state, and if no passwords have been found, the job is put into the **Exhausted** state.

4.2 Overview of the server-side systems

The server-side systems are a set of daemons that communicate using a MySQL database, as well as a webserver that is used to communicate with the clients. This part is common to all BOINC projects, but Fitcrack also has a web application that can be used by the administrators to control the system, add and manage jobs, and so on. Given that the web application was not relevant to the topic of this thesis, it will not be described here. The daemons relevant to this thesis and information about the logic created for them in the Fitcrack project are detailed below.

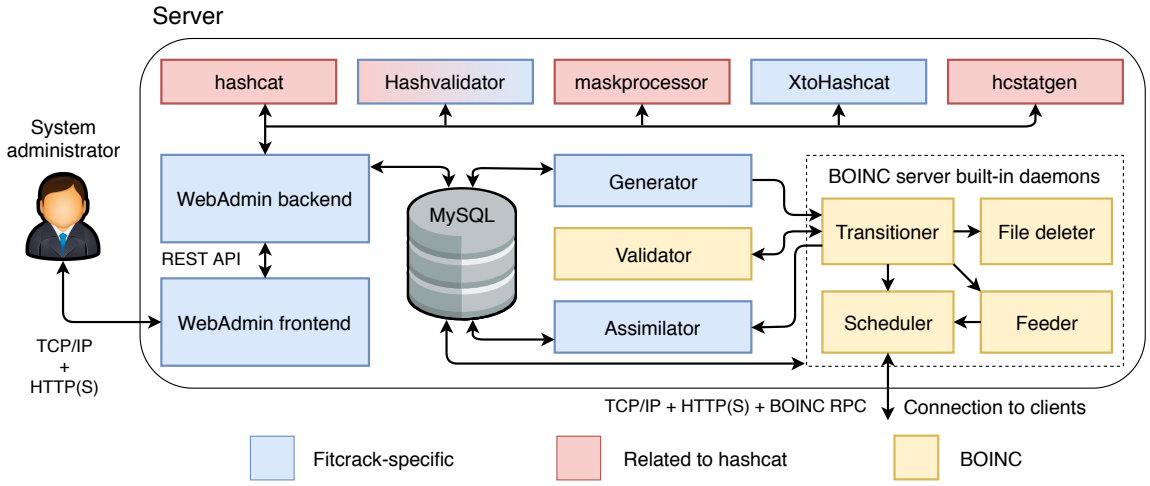


Figure 4.1: A diagram showing the general architecture of the Fitcrack server. Taken from [9].

4.2.1 Assimilator

`Assimilator` is mostly implemented by the BOINC project, with the Fitcrack-specific part being the implementation of a function called `assimilate_handler`. This function is called by BOINC with the workunit results received by the server. The code then has a separate branch for dealing with each of the three main workunit types: normal workunits, which actually attempt to crack hashes, benchmark workunits, which estimate the cracking power of the host they are assigned to for a specific job, and „bench all" workunits, which estimate the cracking power of the host for all supported algorithms.

The simplest of the three is the handling of the „bench all" workunit results. It simply iterates over each algorithm and its measured speed and adds it to the database, or updates it in case an estimate is already present (for example from a previously done normal benchmark).

When the result of a benchmark workunit is received, the raw power sent by the client is saved as the benchmarked power for the used algorithm. After that, further processing is done to create an estimate specific for the job to which the benchmark workunit belongs. For a mask attack, the power is divided by the ratio of the mask's real keyspace size to its keyspace size as reported by Hashcat and then further by a factor of three. For a dictionary attack, it is simply capped at one million. The result of those alterations is then used as the current estimated power of the host for the given job.

If the received result comes from a normal workunit, the result code is checked to see if the workunit cracked any hashes. If it did, the results are written to the database and `assimilator` checks whether there are any uncracked hashes left. If not, it changes the job's state to finished. Regardless of whether any hashes have been cracked or not, `assimilator` also gets the workunit's keyspace size from the database and divides it by the elapsed time to get the average hashrate of the workunit. If the job's attack mode is dictionary and it has rules, it also multiplies that hashrate by the number of rules in order to be consistent with `generator`. After that, it checks if the workunit was at least half the size that would be expected for the previous power estimate and the job's `seconds_per_workunit` setting, and if so, updates the host's power estimate for the job. The check is done in order to avoid the estimates being skewed towards lower numbers by excessively small workunits, in which the time taken to compute the workunit is dominated by overhead. These small workunits can be generated for example due to the fragmentation algorithm of the combinator attack.

4.2.2 Trickler

Similarly to `assimilator`, `trickler` is also mostly implemented by the BOINC project with the Fitcrack specific part being an implementation of a single function, `handle_trickle`. It simply parses the workunit identification and the current progress from the trickle message and updates the database accordingly.

4.2.3 Generator

Generator is essentially a simple infinite loop. In it, `generator` loops over each active job and each host assigned to said job, checking the number of unfinished workunits. If the host has not yet finished the initial benchmark in the current job and does not have an unfinished benchmark workunit, one is created for it. If the host is already benchmarked, `generator` checks whether it has less than two unfinished workunits, and if so, generates one for it.

It also takes care of changing the job's state should the conditions for the change be satisfied. If a job has a set end time and that time has already passed, the job is put into the finishing state. If a job is already in the finishing state, then its state is changed according to the logic described in section 4.1.

As for the workunit generation process, most of it is implemented in class `CAttackMode` and its subclasses. There is generally at least one subclass of `CAttackMode` for each specific attack mode, though there can be more. For example, there is a subclass for a dictionary attack with rules, and another subclass for a dictionary attack without them.

First, the generator loads the resources necessary for the attack such as hashes, masks, dictionaries, and so on. Then it calculates the time the workunit should take. This calculation has several constraints. The calculated time is bounded from above by the time per workunit that has been set for the job, and from below by the minimum that's set as a compile-time constant in `generator`, whose current value is 60 seconds. Other than that, it

is no higher than the elapsed time of the job, which is responsible for the ramp-up period. It is also limited to a certain fraction of the estimated remaining time, which is responsible for the ramp-down period [5].

After that, one of the attack mode classes is instantiated based on the job's attack mode. Then the database is checked for the presence of failed workunits. If any are found, the smallest one is taken and instead of a new workunit which would continue processing new candidate passwords, the failed workunit's parameters (most importantly the start index and the size) are supplied to the attack mode class. Afterward, the `makeWorkunit` method of the attack mode class is invoked.

This method first checks if the new workunit's parameters have already been set. If they have not, the start index is set from the current progress of the job, and the size is determined by multiplying the host's power by the desired workunit time. Then the method generates the input files for the workunit based on the set parameters, registers the workunit with BOINC, and restricts it to the target host.

4.3 Overview of the client-side systems

The Fitcrack-specific programs that run on the client consist of Hashcat, which performs the actual password cracking, external password generators whose output can be piped into Hashcat (at the time of the writing there is a generator for PCFG attacks [15] and a generator for the PRINCE attack, both implemented outside Fitcrack), and lastly, there is `runner`, which serves as the main executable ran by the BOINC client.

4.3.1 Runner

`Runner` has several tasks. The most important is the execution of Hashcat with parameters according to the configuration sent by the server, getting the results, and sending them to the server. This involves launching a subprocess, reading and parsing its output, and then transforming it into a format that the server understands. For the attacks which use an external generator, it also involves creating another subprocess and ensuring that its output is used as input of the Hashcat subprocess. It also sends progress reports to the server using the BOINC trickle message mechanism.

Runner is compiled for 64-bit Windows and Linux. It is compiled with all libraries (including system libraries on Linux) linked statically so as to minimize or eliminate dependencies on the client system. Unlike the server, which is written in C++11 [13], `runner` is written in C++98 [12]. The general operation of runner begins by unpacking the files required by Hashcat, as they are sent to the client as a ZIP archive. Then it reads the configuration file created for the workunit by `generator`. According to the contents of the file, it finds the necessary input files, creates the appropriate parameter lists for the subprocesses, and launches them. If an external generator is used, its output is piped directly into the Hashcat process without any processing done by `runner` itself.

While the computation is running, `runner` only checks the status messages periodically output by Hashcat, parses the current progress from them and sends them to the server as trickle messages. After Hashcat finishes running, its exit status and output files are checked to determine if the computation succeeded and whether any passwords were found. This information is then transformed into the format expected by the server and handed over to the BOINC client to be sent to the server.

Chapter 5

Current problems

The goal of this thesis is optimizing the task distribuiton in Fitcrack, and in this chapter I will describe several areas in which the current state of Fitcrack is not ideal and explain the associated problems.

5.1 Node power estimation

In order to get an estimate of the hashrate a node is capable of before it starts computing, Fitcrack currently runs Hashcat with the `--benchmark` argument. This, however, does not accept any inputs other than the hashing algorithm. This benchmark then gives an estimate of the hashrate of a node with the given algorithm, which is often inaccurate by orders of magnitude.

This can be seen in Table 5.1), where **benchmark** is the hashrate obtained using the `--benchmark` argument, **mask** is the maximum hashrate achieved during a real mask attack, **dictionary** is the maximum hashrate achieved during a real dictionary attack, and **relative** is the percentage of **benchmark** in the preceding column. Aside from the percentages, the values given are in passwords/second.

hash algorithm	benchmark	mask	relative	dictionary	relative
7zip	6043	5921	97.98%	5279	87.35%
md5	1.6559e+10	3.5516e+09	21.44%	8.861e+08	5.35%
md5-salts	1.6559e+10	1.7966e+09	10.84%	6.4128e+08	3.87%
pbkdf2	2.8459e+05	2.6169e+05	91.95%	2.6308e+05	92.44%
pbkdf2-salts	2.8459e+05	2.8905e+05	101.56%	2.9074e+05	102.16%
sha512	6.772e+08	8.0008e+07	11.81%	5.7149e+07	8.43%
sha512-salts	6.772e+08	7.916e+07	11.68%	5.7139e+07	8.43%

Table 5.1: Comparison of benchmarked and achieved hashrate

5.2 The handling of rules and salts

Knowing the number of hashes per second that the computer can calculate each second is not, by itself, sufficient to create an appropriately sized workunit. The server can only control how many candidate passwords a workunit is supposed to process. However, one candidate password can require many hash operations to be computed in order to be fully

processed. If an attack involves applying multiple rules to each input string, then the number of actually processed candidate passwords is the number of input candidate passwords multiplied by the number of applied rules.

But this number is still not always the number of hash operations necessary to carry out an attack. If a job is attacking several salted hashes, each candidate password must be hashed with each salt in order to check if it matches. The number of hashing operations must therefore be further multiplied by the number of distinct salts being attacked. The overall formula for the number of hashes that must be computed is therefore $InputPasswordCount \cdot RuleCount \cdot SaltCount$.

Since `generator` can only control the number of input passwords, while it always sends all the rules and salts, it should have access to information that allows it to determine this number appropriately. Currently, however, Fitcrack stores a host's power in the database as a number whose meaning isn't entirely clear. This is caused by every attack treating it differently, and even `assimilator` has special cases in its benchmark handling code for several attacks. For mask attacks, `assimilator` divides the benchmark results by three (probably to deal with the benchmarks' general inaccuracy) and then further by the ratio of the job's real keyspace size to its keyspace size as it's reported by Hashcat. The results for dictionary attacks, on the other hand, are limited to one million, presumably because dictionary attacks can be limited by the rate at which passwords can be sent to the GPU. Curiously enough, attacks which get the candidate passwords from an external generator, and are thus equivalent to a dictionary attack in this respect, have no such limit.

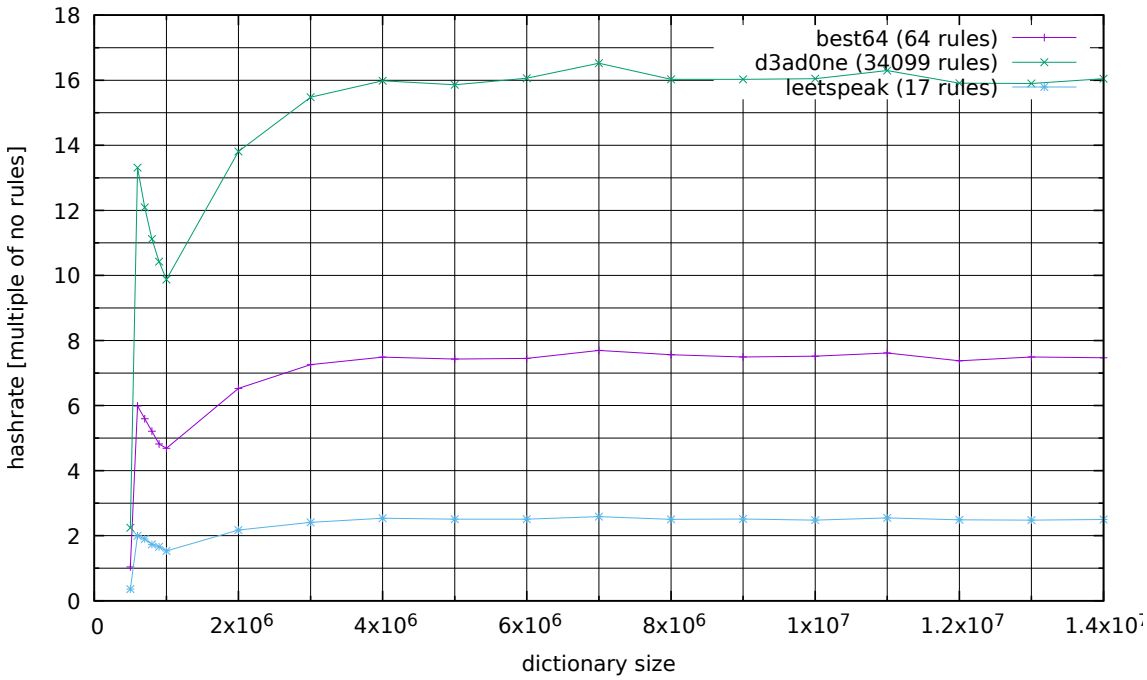


Figure 5.1: The factor by which hashrate is increased for different rulecounts and keyspaces in dictionary attacks.

.

Further processing is then done by `generator`, which always multiplies this number by the desired number of seconds that the workunit should take, and then does further processing. For attacks with rules, it is divided by the number of rules. This is probably

based on the assumption that rules multiply the amount of work to be done, but don't affect the cracking speed, which is false as can be seen in Figure 6.6. Unfortunately, in the PCFG attack with rules, the number of passwords is first limited to not be larger than the PCFG keyspace, and only then is it divided by the rule count.

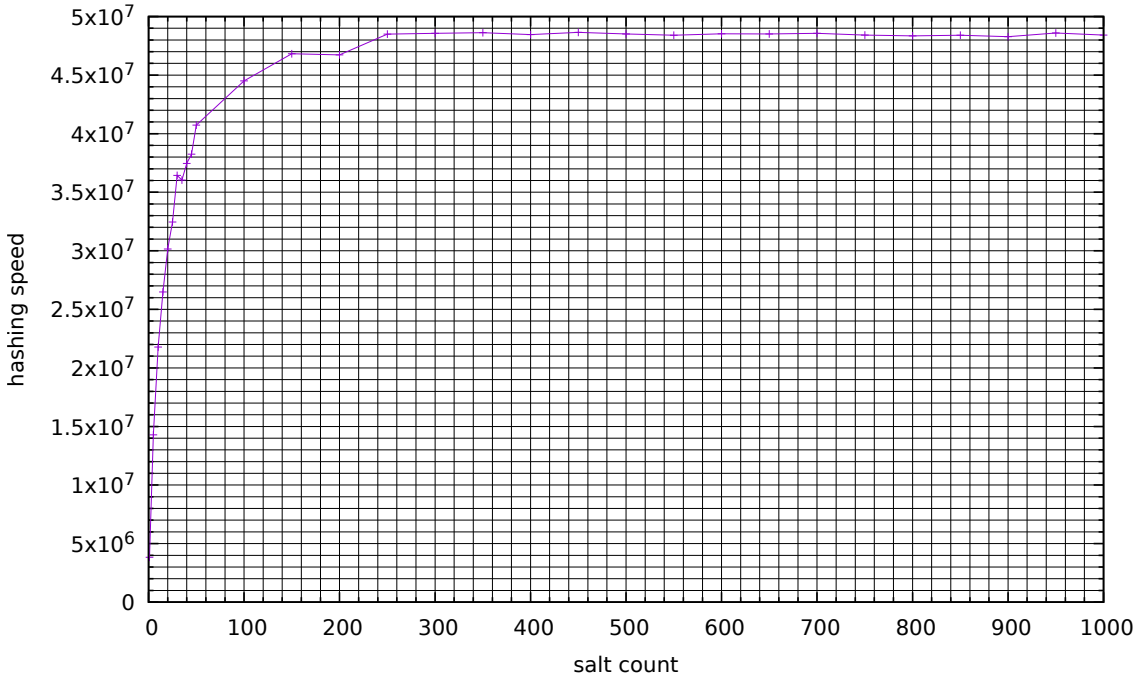


Figure 5.2: A plot of SHA-512 hashrate depending on the number of salts in the attack.

Another thing that influences the amount of work that has to be done by Hashcat, and possibly the cracking speed (as can be seen in Figure 5.2), is the number of salts in the attack. Currently, Fitcrack does not take the salt count into account at all.

5.3 Workunit pipelining

BOINC supports assigning multiple workunits to each host simultaneously. Unfortunately, running multiple instances of Hashcat at the same time doesn't really make much sense, as GPUs generally will not run two tasks at the same time. Running multiple instances of Hashctat would thus lead to switching between the two tasks at best, or to one of the tasks crashing due to insufficient resources at worst. Running multiple instances would, therefore, if successful at all, just degrade performance because of the overhead of task switching. It can, however, be useful to download the data for another workunit and prepare it to run while another workunit is being processed. This would lead to effectively negating the temporal overhead of downloading the workunit data. However, to achieve this, it is necessary to prevent the second workunit from running as soon as its data is received, and instead only having it run when the computation of the previous workunit is finished.

Currently, Fitcrack can use the BOINC facilities to have the BOINC client download multiple tasks, but only run one at a time. This, however, requires that a configuration file be created in a specific location on the worker node and loaded by its BOINC client [10].

This means that this option is only usable in an environment in which every connected worker node is administered by a person of at least moderate technical abilities (they have to create the configuration file and force the BOINC client to load it). This is a disadvantage compared to the default settings, where the server only sends the next task when the previous one is finished, and joining the project consists only of installing and running the BOINC client and then joining the project using the built-in wizard.

5.4 Hybrid and Combination attacks

Hashcat supports several attack modes that combine candidate password generation algorithms from more attack modes, specifically the Combination attack and two Hybrid attacks. These attack modes are described in section 2.1.

Unfortunately, it is impossible to achieve fine-grained control of the keyspace size using only Hashcat's native keyspace controlling mechanism (the `--skip` and `--limit` command-line flags) in these attacks. This is because in the combination attack mode, Hashcat keyspace of 1 corresponds to trying one password from the „left“ dictionary with *all* passwords from the „right“ dictionary. Therefore when the „right“ dictionary is large, it can become impossible to assign sufficiently small workunits to weaker workstations. Fitcrack deals with this by fragmenting the „right“ dictionary and, if further lowering of the keyspace size is needed, limiting the Hashcat keyspace. Specifically, Fitcrack has an estimate of how many passwords it should send to the worker node. Based on this desired real keyspace, there are two alternative approaches Fitcrack currently uses. For the algorithm see algorithm 1. The following paragraphs describe the algorithm and discuss its properties and weaknesses.

Algorithm 1 describes the process `generator` currently uses to determine what parameters and data to send to the client in a given workunit. The only listed input is `desiredPasswords`, which is the number of passwords that should be sent to the client, which was calculated from the desired workunit computation time and the host's power. However, there are several other variables which are not listed as input, as they are kept as a persistent state in the Fitcrack database. These are `fragmentIndex`, which is used during fragmentation and stores the number of passwords processed from the left dictionary with the current password from the right dictionary, and `rightDictIndex`, which keeps track of how many passwords have already been fully processed in the right dictionary. As for other notation used in the algorithm, `leftDict` and `rightDict` are the left and right dictionaries treated as lists of strings, and `wu` is a structure containing members `dict1` and `dict2`, which represent the dictionaries sent to the client, as well as `skip` and `limit`, which represent the appropriate Hashcat parameters.

If the worker node's power is sufficiently large, Fitcrack sends one to N passwords from the „right“ dictionary and lets the worker node test all the passwords from the „left“ dictionary with the few sent passwords used as the „right“ dictionary. This can be seen in Algorithm 1 on lines 11 to 14. Then it continues in a similar fashion with the rest of the „right“ dictionary. This way Fitcrack cannot precisely control the number of passwords in the workunit, but this is offset by the simplicity of the algorithm itself, and the error is, in absolute terms, at most half of the size of the „left“ dictionary. Currently, `generator` stores the password index in the database, and, for each workunit, it gets to that index by reading the file until reading that many passwords.

Alternatively, if the worker node's power is insufficient to process the entire „left“ dictionary with even one password from the „right“ dictionary in a timely manner, Fitcrack

Algorithm 1: Fitcrack's current work distribution algorithm for combination attack

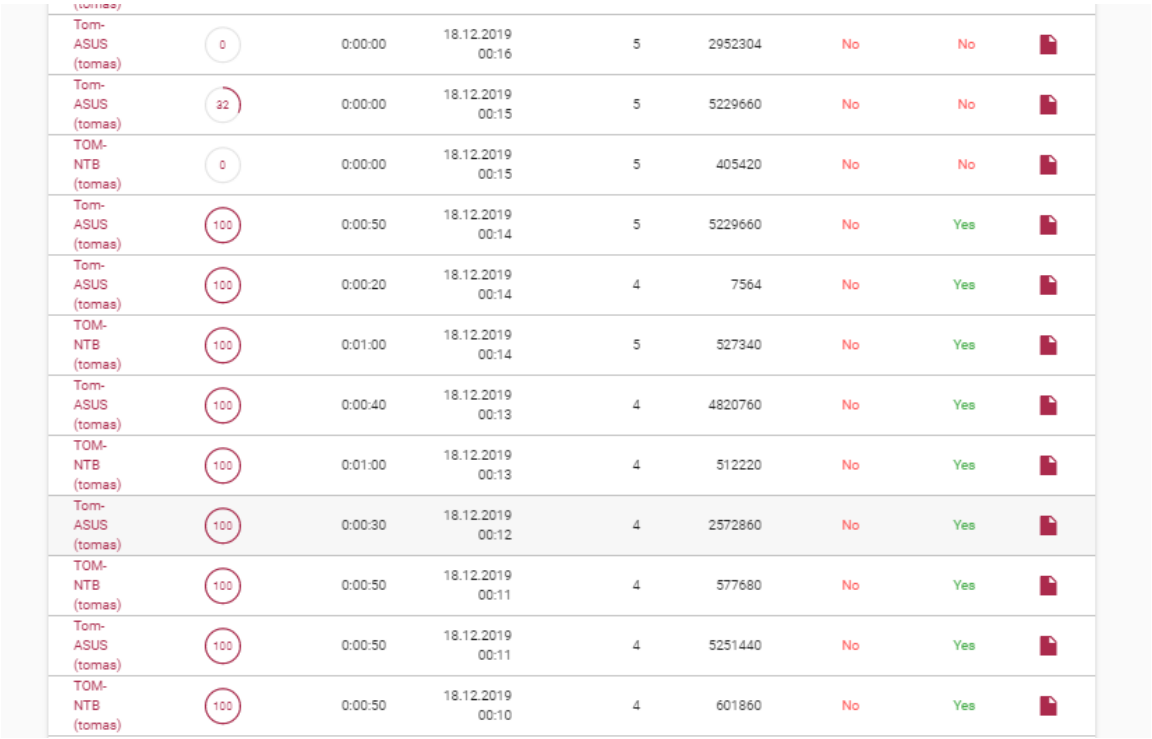
Input: $desiredPasswords$
Output: wu

```
1:  wu.dict1 = leftDict
2:  if fragmentIndex/ = 0 then
3:      wu.dict2 = [rightDict[rightDictIndex]]
4:      wu.skip = fragmentIndex
5:      if desiredPasswords < |leftDict| − fragmentIndex then
6:          wu.limit = desiredPasswords
7:          fragmentIndex+ = desiredPasswords
8:      else
9:          rightDictIndex+ = 1
10:         fragmentIndex = 0
11: else if desiredPasswords ≥ 0.5 · |leftDict| then
12:     rightPwdCount = round(desiredPasswords/|leftDict|)
13:     wu.dict2 = rightDict[rightDictIndex : rightDictIndex + rightPwdCount]
14:     rightDictIndex+ = rightPwdCount
15: else
16:     wu.dict2 = [rightDict[rightDictIndex]]
17:     wu.limit = desiredPasswords
18:     fragmentIndex = desiredPasswords
```

starts further fragmenting using Hashcat's keyspace control. It does this by sending a single password from the „right" dictionary, and controlling the number of hashed passwords with the `--skip` and `--limit` Hashcat parameters. This is described on lines 15 to 18. However, this leads to only some passwords from the „left" dictionary being tried with the „right" dictionary password. It is therefore necessary to subsequently test the rest of the passwords from the „left" dictionary with the „right" password. Fitcrack currently works through the keyspace in a strictly sequential manner, so when it is generating the next workunit, it assigns the same single password from the „right" dictionary with some passwords from the „left" dictionary, but at most the rest of them. This is shown on lines 2 to 10.

Unfortunately, the way fragments are finished means that when there is a large disparity between worker nodes, the weaker worker nodes always start a fragment, and then when a more powerful worker node asks for work after a weaker node, it only receives whatever work is left after the weaker node, which can lead to great inefficiency. Right after the fragmentation is started, there is always at least half of the passwords from the „left" dictionary to process. However, since when fragmentation has already started, there is no consideration for how many passwords will be left in the fragment after the workunit is assigned, it can happen that a weak worker node will take slightly less than half the passwords, and then another weak node can take almost all the rest, but not all of them. If a more powerful node subsequently asks for more work, it can happen that it will be assigned only very few passwords.

This situation is illustrated in figure 5.3. In it, Tom-ASUS is approximately 10 times as powerful as Tom-NTB, yet it can be seen that it received a workunit with keyspace size of only 7564, as opposed to the approximately 5 million it could handle. This was caused by



Tom-ASUS (tomas)	0	0:00:00	18.12.2019 00:16	5	2952304	No	No	
Tom-ASUS (tomas)	32	0:00:00	18.12.2019 00:15	5	5229660	No	No	
TOM-NTB (tomas)	0	0:00:00	18.12.2019 00:15	5	405420	No	No	
Tom-ASUS (tomas)	100	0:00:50	18.12.2019 00:14	5	5229660	No	Yes	
Tom-ASUS (tomas)	100	0:00:20	18.12.2019 00:14	4	7564	No	Yes	
TOM-NTB (tomas)	100	0:01:00	18.12.2019 00:14	5	527340	No	Yes	
Tom-ASUS (tomas)	100	0:00:40	18.12.2019 00:13	4	4820760	No	Yes	
TOM-NTB (tomas)	100	0:01:00	18.12.2019 00:13	4	512220	No	Yes	
Tom-ASUS (tomas)	100	0:00:30	18.12.2019 00:12	4	2572860	No	Yes	
TOM-NTB (tomas)	100	0:00:50	18.12.2019 00:11	4	577680	No	Yes	
Tom-ASUS (tomas)	100	0:00:50	18.12.2019 00:11	4	5251440	No	Yes	
TOM-NTB (tomas)	100	0:00:50	18.12.2019 00:10	4	601860	No	Yes	

Figure 5.3: A powerful node can be given tiny workunits under the right circumstances. The rightmost number is the workunit's keyspace size.

it asking for a workunit when the current fragment was almost depleted. Nevertheless, the workunit still took 20 seconds to execute due to overhead. Note that this is far less likely to happen when the desired workunit processing time is greater than the 60 seconds used for this test, but it can happen even then.

Hashcat also supports two so-called hybrid attack modes, which are analogous to the combination attack mode, except instead of having the left or right part of the password come from a dictionary, it is instead generated from a mask. Fitcrack currently supports this attack mode, but it doesn't actually run Hashcat in this mode, as once again Hashcat treats keyspace of 1 as trying one password from the left part (whether it's a mask or a dictionary) with all passwords generated from the right part. This means that in the dictionary+mask version of hybrid attack, Fitcrack cannot use the same fragmentation technique it used with the combination attack, and therefore wouldn't be able to control the keyspace for a given workunit. Therefore, when the user submits a job with a hybrid attack mode, the Fitcrack server uses Hashcat to generate all the passwords from the mask and save them to a file. It then uses this file as a dictionary and treats the attack as a combination attack [8]. This unfortunately means that instead of simply transferring a dictionary and a mask, Fitcrack now has to transfer another dictionary over the network. Fitcrack could use the fragmentation technique on the mask+dictionary version of the hybrid attack, but it doesn't, possibly for consistency reasons.

Chapter 6

Proposed optimizations

Given the inefficiencies and other problems described in the previous chapter, I propose improvements in several areas, which shall be described in the rest of this chapter.

6.1 Node power estimation

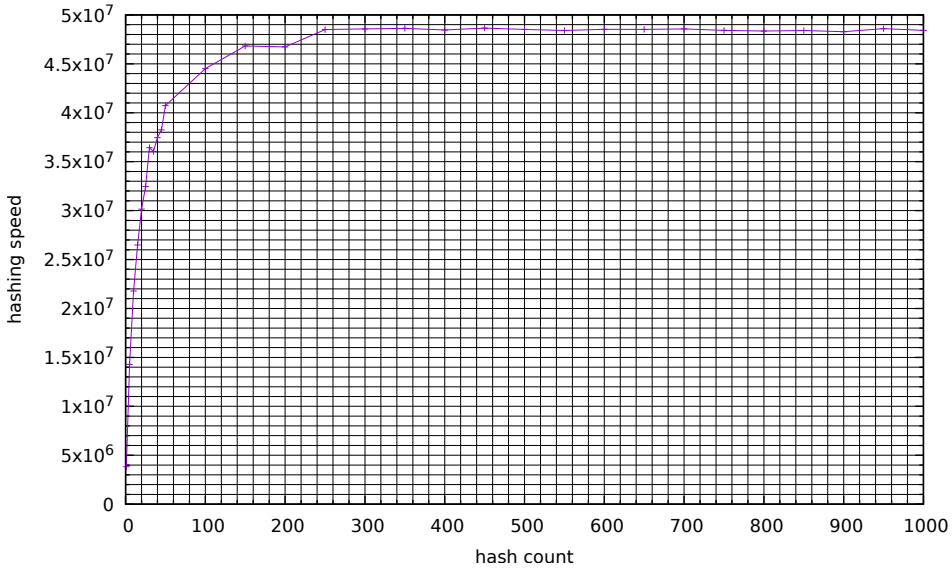


Figure 6.1: Influence of the number of salts being cracked on the hashrate with SHA-512

In addition to the `--benchmark` argument, Hashcat also has a `--speed-only` argument, which works differently. With this type of benchmark, Hashcat is started with the same arguments and inputs which are used for the actual workload. This then runs the actual attack for a small amount of time and then returns the actual measured performance. The returned speed is then a far better indicator of node performance during a real attack. It would also solve the problem mentioned in section 5.2, where the power was simply divided by the number of rules, even though having rules could actually increase cracking performance. Given that the benchmark would be carried out with the rules, their effect would already be taken into account, and dividing by their count would therefore be appropriate. Unfortunately, Hashcat apparently only attempts cracking a single hash when run with this

argument, regardless of the actual number of supplied hashes. Given that simultaneously cracking hashes with multiple salts can, in some attack modes, lead to a significant speedup (See figure 6.1), this argument is also not usable to obtain an accurate estimation of node performance during the actual attack. An example of a situation where this inaccuracy occurs is a dictionary attack on a fast salted hash such as md5 with salt.

The approach I propose is to use the `--runtime` option with some small value as an argument, which really starts cracking the supplied hashes, but exits after the number of seconds supplied as an argument to the `--runtime` option. This allows obtaining the actual achievable cracking speed in bounded time which does not depend on the node power (disregarding the time taken to start Hashcat, which should not be very dependent on the inputs).

The downside compared to the currently used approach is that in order to run the benchmark, data that will be used in the cracking attempt must be sent to the node as it can influence the hashrate (see figure 6.2 for the influence of length of the passwords in the input dictionary on 7-zip cracking speed).

This is not much of a problem with a mask attack, as the attacked mask is always sent to the worker node, and in any case is only a few bytes, which is quite insignificant. The only remaining question is then which mask should be sent, seeing as one Fitcrack job can specify several different masks. As can be seen in figure 6.5, masks with small keyspace generally result in a low estimate, which increases with keyspace size until it reaches a ceiling. Sending a mask that generates only small keyspace could therefore result in an estimate that is too low for larger masks. The reverse is also true, but that generally doesn't matter, as even at a lower speed, the small keyspace is exhausted in a matter of seconds, and the undesired effect of giving a worker node a task which runs far longer than desired will therefore not happen.

Sending all data can, however, incur significant overhead with a dictionary attack [6], as ideally the whole dictionary (the size of which can be on the order of hundreds of megabytes or even gigabytes) would be sent. One might argue that this data will then be used in the actual attack, but given the way Fitcrack splits the work among the nodes, each node would only use a part of the data and the rest would be transferred needlessly. Furthermore, cracking can start with a chunk of the dictionary, and subsequent chunks can then be received in the background, not contributing to overhead from the perspective of computation time. It was therefore desirable to find a way to get the cracking speed without sending the entirety of the actual data over the network.

Fortunately, it would seem that the only property of the dictionary that significantly affects cracking speed is its size, as can be seen from figure 6.4. The only exception that I have encountered in my tests is cracking 7-zip encryption, for which sorting the dictionary by candidate password length has resulted in a significant speedup (roughly threefold). This can be seen in figure 6.3. For all other password processing functions that I have investigated, the hashing performance in a dictionary attack seemed to remain fairly constant when various transformations were applied to the dictionary and seemed to change only slightly when the candidate passwords were piped into the Hashcat process instead of Hashcat reading them directly from the filesystem.

As for the effect of the password inputs on other attack modes, the hash rate in combination attacks seems to be affected only by the keyspace of the „left“ password source, whether it's a dictionary or a mask. This is illustrated in figure 6.7. You can see that although the plot is rather jagged, the actual data points are always clustered together.

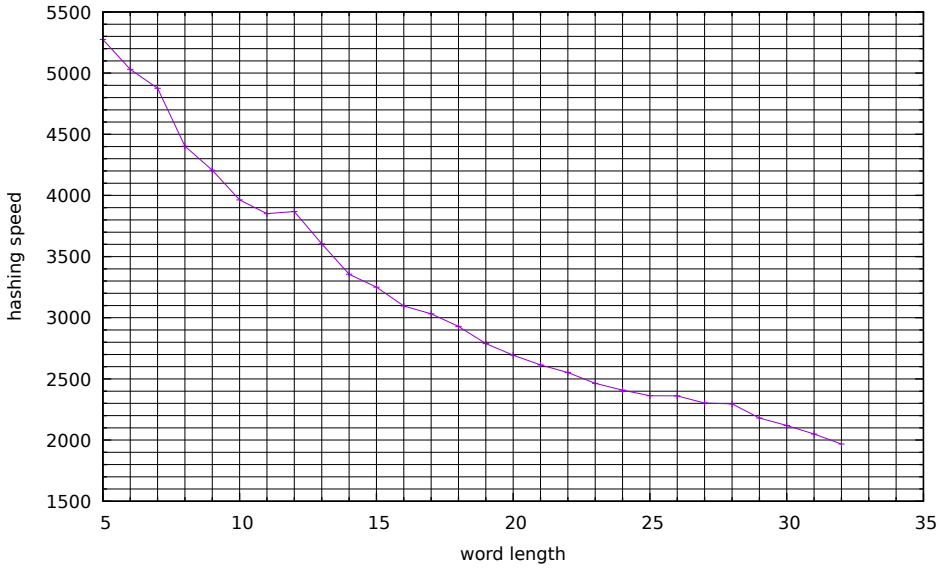


Figure 6.2: Word length influence on 7zip hashrate

As the baseline, I have used the case when Hashcat reads the input dictionary directly from the filesystem and uses all candidates. For this the benchmarking script has always prepared a version of the dictionary which was truncated to the desired length and then given as input to Hashcat. The variations on the input that I have benchmarked are as follows:

- Giving Hashcat the entire input dictionary and limiting the keyspace using the `--limit` option.
- Same as above, but skipping some of the input using the `--skip` option.
- Instead of writing the truncated file to disk and letting Hashcat read it, piping the truncated dictionary into the Hashcat process.
- Sorting the dictionary by candidate password length, shortest password first (sorting happened after truncation.
- Same as above, except the longest password first.
- Replacing every character in every candidate password with the letter 'a'.

Given that the effect of these variations on the cracking performance is negligible, it should be possible to obtain a good estimate for the cracking performance during the execution of the real workunits using a dictionary that merely has passwords of the same length as the real dictionary. This should allow Fitcrack to just send password lengths instead of the actual passwords as data for the benchmark workunit, which should lead to significant bandwidth savings. Also, using a dictionary sorted by password length, it would be possible to just send the list of password lengths present in the dictionary along with the count of passwords having said length. Compared to sending the real dictionary, this would be essentially free. Sorting the dictionary, however, does affect the cracking performance for 7-zip. Fortunately, the effect is positive, so I propose that all dictionaries be sorted by

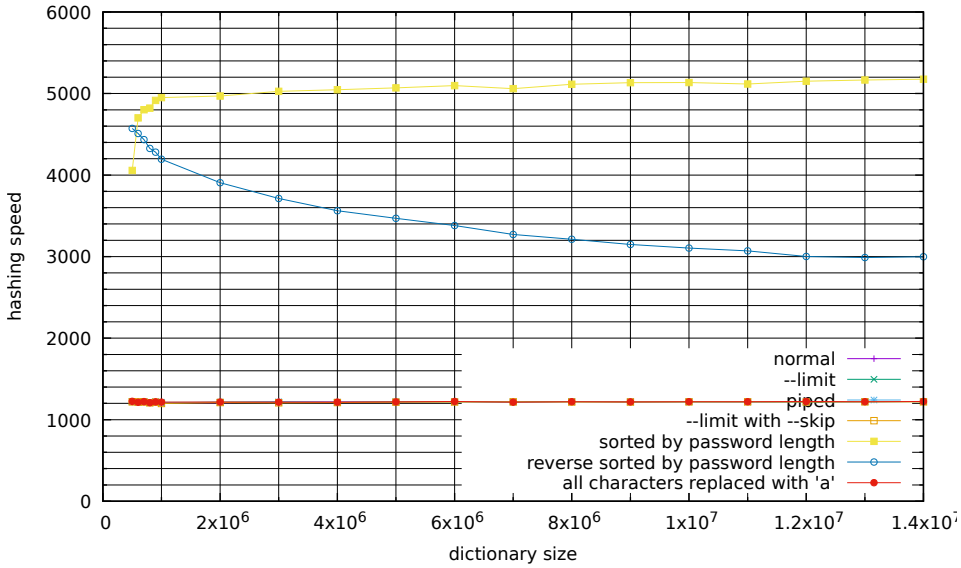


Figure 6.3: Influence of dictionary length on variants of a dictionary attack on 7-zip

password length when they are added to the Fitcrack server. This would enable efficient and accurate estimation of cracking performance, as well as lead to a significant increase in cracking performance in some types of tasks.

6.2 The handling of rules and salts

Since the fact that different attacks treat the power stored in the database differently can make reasoning about the host's power difficult, I propose that the power be stored for all attacks as the number of candidate passwords per second that hashcat can process *on its input*. I will then be able to remove the differences in handling in `generator`, since the workload expansion by rules and salts is done in Hashcat, and so will already be taken into account in this number.

Given that Hashcat reports the speed in hashing operations per second (that is after multiplication by rules and salts), the code dealing with rules and salts has to be either in `runner`, or in `assimilator`. Putting it inside `assimilator` would have the advantage of `runner` not having to deal with what are essentially server-side problems and just reporting the speed as it was given by Hashcat. It would also enable `assimilator` to save the results elsewhere to be used as a general estimate of the host's cracking power.

Unfortunately, it is not straightforward for `assimilator` to find out whether a hash is salted, and multiple unsalted hashes do not increase the number of required hashing operations. Hashcat outputs the number of salts in its progress reports, so `runner` can find the number of salts just by parsing that in addition to the cracking speed. I have therefore decided to divide the speed by the number of salts in `runner`, while the division by the number of rules can be done in `assimilator`.

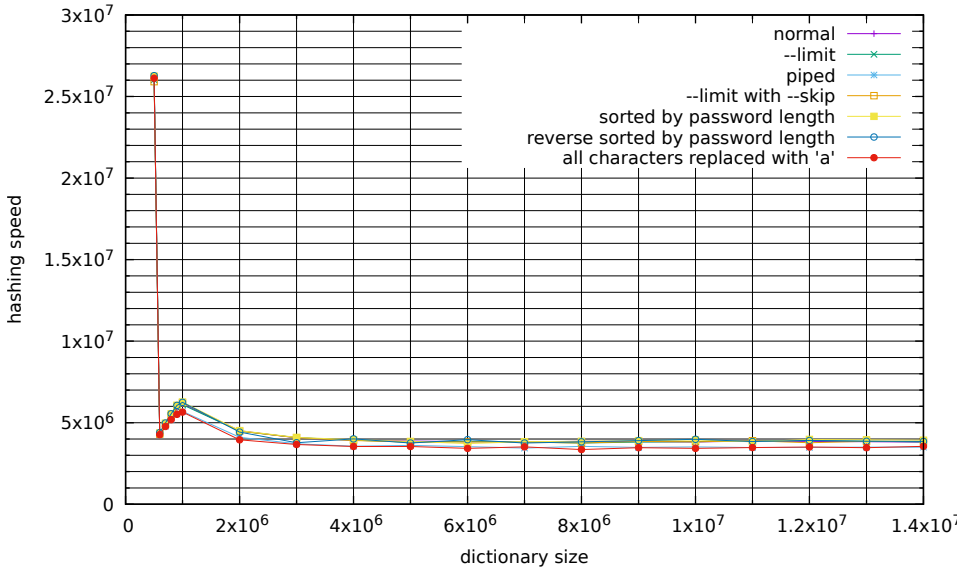


Figure 6.4: Influence of dictionary length on variants of a dictionary attack on SHA-512

6.3 Workunit pipelining

Efficient pipelining of the workunits requires some sort of mutual exclusion between the tasks. There are two possibilities of achieving mutual exclusion that I have identified:

- Alter Runner.
- Use BOINC facilities.

Altering Runner would mean turning it into something akin to singleton application, except instead of quitting on finding another instance of itself, it would wait for the previous instance to finish and then proceed with the computation.

Using BOINC facilities is also possible. The BOINC client has a configuration option that allows one to limit the number of concurrently running tasks. This limit can also be set per project. BOINC then downloads as many tasks as the server is willing to provide, but only runs the configured number of tasks simultaneously, while the rest waits for a free execution slot. Unfortunately, the BOINC client does not limit concurrent execution of tasks by default, and so the server cannot be set to send more workunits by default either. This is also documented in [10]. It is only possible to do so when it can be reasonably expected that all clients will configure the Fitcrack project to only run one task at a time. A possible workaround can be realized due to the fact that the client's project configuration is stored in a file in the same directory in which BOINC stores the project files received from the server. It should therefore be possible to just send the project configuration from the server along with the other files. However, the BOINC client does not attempt to load the configuration file after receiving files from the server. It would therefore still be necessary to make the client reload the configuration after it is downloaded from the server. This can be done either by sending the BOINC client a command to reload the configuration, or by restarting the client. Both can be accomplished by user intervention, but if user action is required, sending the file offers little real advantage over simply having the node's user create the file themselves. The reload command can also be sent from a program, so it

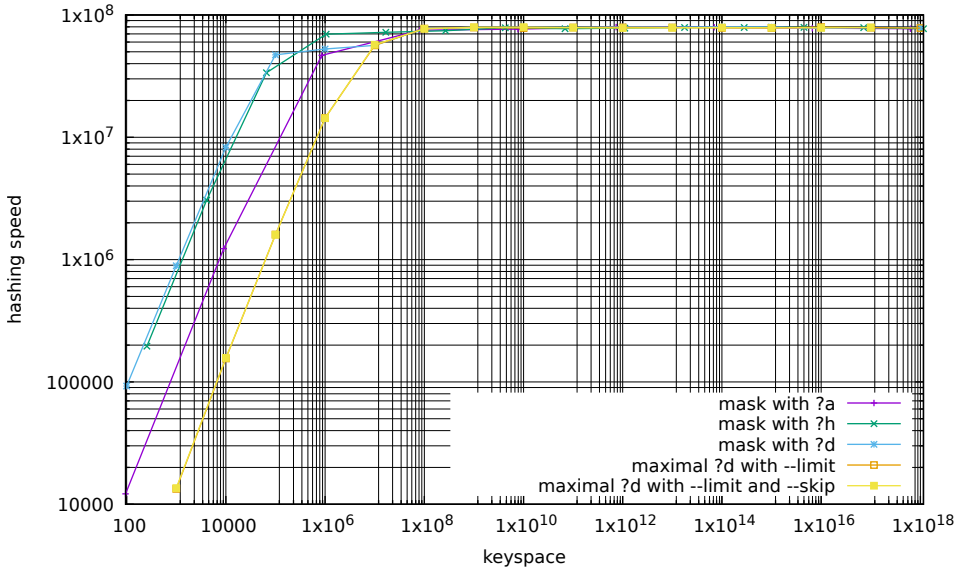


Figure 6.5: Influence of keyspace size on mask attack on SHA-512

should be possible to modify Runner to send the reload command, but since that would mean modifying Runner, it would be far simpler to just modify Runner to implement mutual exclusion by itself, so that is what I decided to do.

Furthermore, when using the BOINC exclusion, there are delays between the end of one Hashcat run and the start of the next one. On my machine, the delays are typically between 17 and 19 seconds. Implementing mutual exclusion in the Runner should bring these delays down to a few milliseconds, as the next instance will be ready to launch Hashcat and just waiting for the previous instance's Hashcat process to end.

6.4 Hashcat native hybrid attacks

Firstly, I propose that the workunit generator should store offset into the second dictionary instead of (or in addition to) the current password index. This would allow the server to simply start reading from the right place instead of reading the whole file and counting lines, which could be quite slow for large dictionaries. (Note that the same optimization could be done for the plain dictionary attack as well)

Secondly, I propose that the mask+dictionary attack mode be executed by Hashcat as such instead of being turned into a combination attack mode. It can then use the same fragmenting technique as the combination attack.

Thirdly, I propose that when the combination attack sends out workunits which are supposed to crack just one password from the „right“ dictionary along with just a part of the „left“ dictionary, the workunit generator always check the count of passwords remaining in the „left“ dictionary and split the work in such a way as to not cause excessively small workunits to remain. This should be easily achievable by looking at the desired keyspace and comparing it to the count of remaining passwords in the „left“ dictionary.

If the desired keyspace is less than half of the available keyspace, just continue with that, as the next workunit has at least the same keyspace left to either take as a whole or divide further.

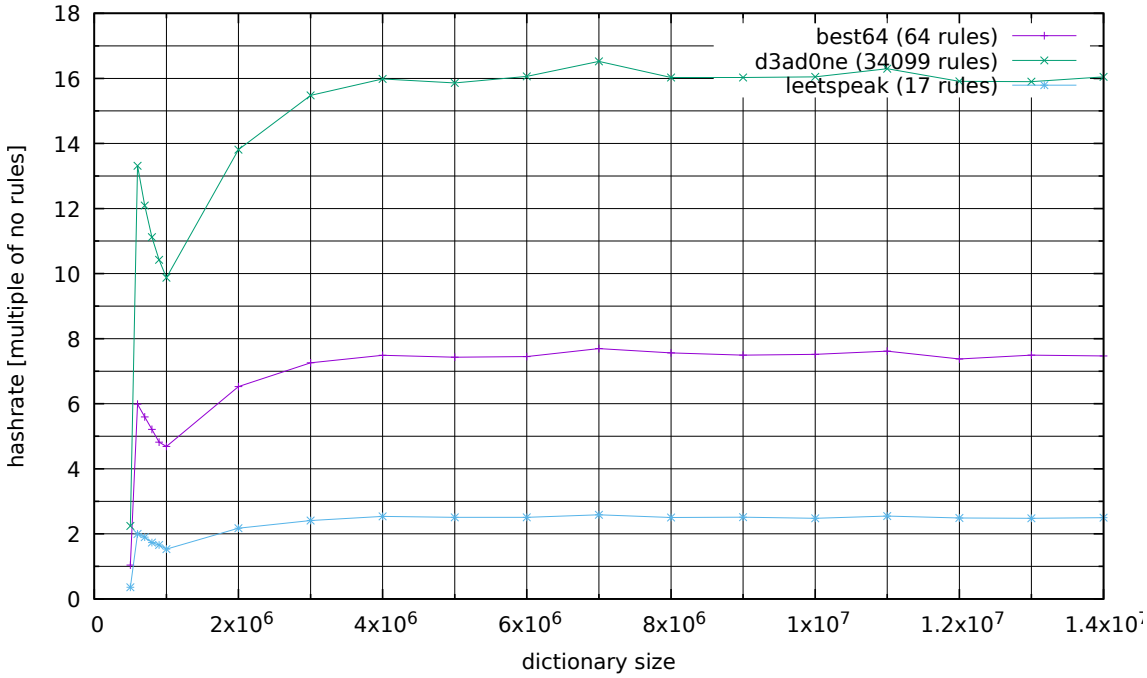


Figure 6.6: The factor by which hashrate is increased for different rulecounts and keyspaces in dictionary attacks.

.

If the desired keyspace is more than half of the available keyspace, check to see if it is more or less than three quarters of it. If it is less, then only take half of the available keyspace. This will mean the workunit will be smaller than desired, but this error is at most $33.\overline{3}\%$ seeing as $\frac{1}{2}/\frac{3}{4} - 1 = \frac{2}{3} - 1 = 0.\overline{6} - 1 = -0.\overline{3}$.

If it is more than three quarters of the available keyspace, take all the remaining passwords. This will lead to a workunit bigger than desired, but the error is again at most $33.\overline{3}\%$ as $1/\frac{3}{4} - 1 = \frac{4}{3} - 1 = 1.\overline{3} - 1 = 0.\overline{3}$.

This leaves only the case when the desired keyspace is larger than the available keyspace. In that case, it should not be significantly larger, since the size should be at least as big as the previous workunit. It could only be significantly larger in case the current node is significantly more powerful than the one for which the previous workunit was created. This is the same problem that is present in the current Fitcrack, but it could be solved by moving to the next password in the „right“ dictionary and storing the current one as unfinished. The workunit generator would then check the unfinished chunks before trying to continue from the current index. It would then assign them to nodes for which they wouldn't be too small by some margin (for example if they were at least half of the desired keyspace), or, since the weak node which produced them could disconnect or become more powerful by, for example, its resources being freed, the system would keep track of which node is the weakest and this node would then just take the smallest unfinished chunks (or possibly the oldest ones), which were in any case probably produced by that node in the first place. This restriction would also be lifted once the „right“ dictionary is exhausted except for these unfinished chunks as it would make little sense for the powerful nodes to do nothing while the least powerful node is attempting to finish the job.

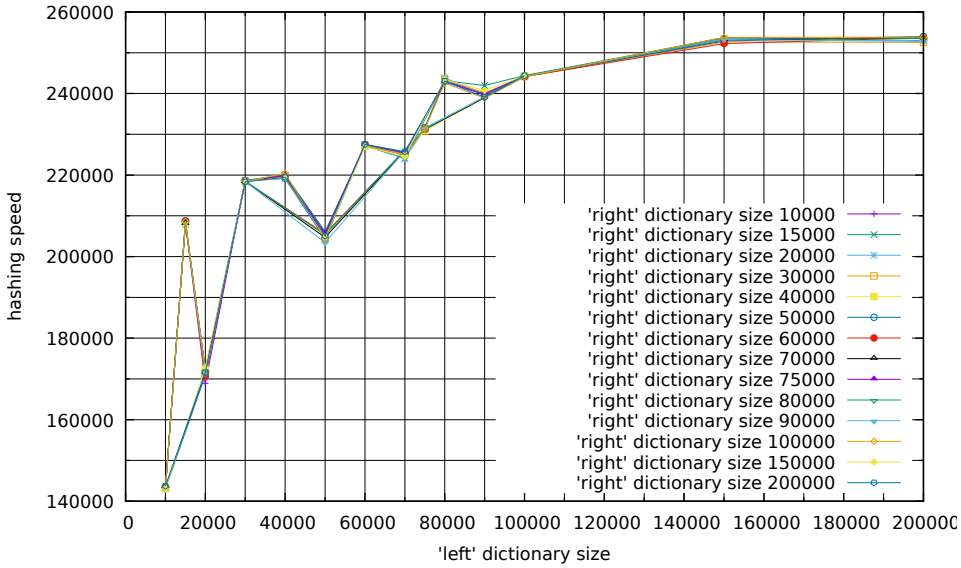


Figure 6.7: Influence of keyspace on the combination attack on PBKDF-HMAC-SHA512

Algorithm 2: Algorithm for continued fragmentation that prevents the creation of extremely small workunits

Input: $remainingPasswords, desiredPasswords, fragmentIndex, dictLen$
Output: $passwordsToSend$

$remainingPasswords = dictLen - fragmentIndex$
if $desiredPasswords < 0.5 \cdot remainingPasswords$ **then**
  $passwordsToSend = desiredPasswords$
else if $worker.power < 0.75 \cdot remainingPasswords$ **then**
  $passwordsToSend = desiredPasswords \cdot 0.5$
else
  $passwordsToSend = dictLen$

Lastly, I propose that the dictionary+mask attack mode also be executed by Hashcat as such. The keyspace cannot be regulated the same way it is regulated in the other hybrid attack modes, but there is a way to limit the keyspace of a mask using the ability to specify custom character sets to be used in a mask. The overall algorithm can be the same as for the dictionary, except for a different way of finding a mask that gives the desired amount of passwords, which is described in the following algorithms: Algorithm 3 describes a subroutine for getting a character set of a size close the desired size while making sure that the remaining characters do not later force the creation of workunits that are far smaller than desired. Algorithm 4 then takes a mask, the current index, and the desired keyspace and outputs a mask that should have keyspace that is at least half of and at most twice the desired keyspace. This way has its downsides, but none are insurmountable.

Since hashcat allows one to specify custom charsets to be used in masks and the masks can also contain literal characters, it is possible to send a mask representing only a fraction of the original mask's keyspace by replacing some of the mask characters with literal characters and one of them with a custom charset. This unfortunately means that only three custom

Algorithm 3: The algorithm of the GetCharsetSlice subroutine

Input: $desiredSize, charset$
Output: $charsetSlice$

```
1: if desiredSize > |charset| * 0.75 then
2: |   desiredSize = |charset|
3: else if desiredsize > |charset|/2 then
4: |   desiredSize = |charset|/2
5: if desiredSize ≤ 1 then
6: |   return charset[0]
7: else
8: |   return charset[0:desiredSize]
```

charsets will be left for the user to use, but Fitcrack currently doesn't allow the user to specify any custom charsets, so this does not impair current functionality in any way.

This way also doesn't allow as fine-grained control of the keyspace as the dictionary method. Nevertheless, I will show that the upper bound for the relative error is the same with this method as it is when the mask is transformed into a dictionary first. Consider these possible cases:

- The worker node is not powerful enough for even one candidate password from the mask. In this case, it can be handled exactly the same as when the right part of the password comes from a dictionary.
- The worker node can handle more passwords and the custom charset is a strict subset of the charset that is supposed to be in that position. In this case, we just add another character from the complete charset. This leads to keyspace growth by a factor of $\frac{\{previouscharsetsize\}+1}{\{previouscharsetsize\}}$, which is at most 2. This is equivalent to the case when we had one password from the „right" dictionary and added a second one.
- The worker node can handle more passwords and there is no character from the original charset which we could add. At this point, we can use the full charset and turn one of the „hardcoded" characters into a custom charset, which starts with one character. With this operation, the keyspace remains the same.

To understand the operation and purpose of Algorithm 4, it is desirable to first define several concepts: First, let us define a **mask slice** of a mask M as a mask that generates a set of candidate passwords which is a subset of the set of candidate passwords generated by M. Next, define the **mask sequence** of a mask M to be the sequence of candidate passwords generated by M.

The purpose of Algorithm 4 is to create a mask slice whose mask sequence has the following properties:

1. It is a *continuous* subsequence of the mask sequence of the input mask.
2. Its first element is exactly the Ith element of the mask sequence of the input mask, where I is the `startIndex` parameter of the algorithm.
3. Its length is close to the `desiredKeyspace` parameter.

Algorithm 4: Algorithm to build a mask with keyspace close to the desired one. It assumes no fragmentation of the dictionary.

Input: $mask, startIndex, desiredKeyspace$
Output: $maskSlice$

$adjustedStartIndex = startIndex$
$resultKeyspace = 1$
$maskSlice = []$
forall $symbol \in mask$ **do**
    if $\neg IsCharset(symbol)$ **then**
        $maskSlice+ = symbol$
        continue
    $charset = symbol$
    $charIndex = adjustedStartIndex \mod |charset|$
    $adjustedStartIndex/ = |charset|$
    if $charIndex > 0$ **then**
        $charset = charset[charIndex :]$
    if $desiredKeyspace \leq resultKeyspace$ **then**
        $maskSlice+ = charset[0]$
        continue
    $remainingKeyspace = desiredKeyspace/resultKeyspace$
    if $charIndex = 0 \ \& \ remainingKeyspace \geq |charset|$ **then**
        $maskSlice+ = symbol$
        $resultKeyspace* = |charset|$
    else
        $maskSlice+ = GetCharsetSlice(remainingKeyspace, charset)$
        $desiredKeyspace = resultKeyspace$
return $maskSlice$

The first and second property are crucial for practical usability of the algorithm, because with them, as long as the next mask slice always begins with the candidate password that follows the last candidate password of the previous mask slice in the original mask's mask sequence, we can keep track of which candidate passwords have already been processed by simply keeping the index of the first unprocessed candidate password in the original mask's mask sequence.

The algorithm works as follows: It iterates over the symbols of the mask, each of which can be a simple character, in which case it is simply appended to the mask slice without any further processing (lines 5 through 7). If the symbol represents a character set, it is analyzed to determine what part of it should be sent to the client. On lines 9 through 12, the algorithm finds how many characters from the character set have already been processed and discards them. It then checks whether the desired keyspace has already been reached. If it has, it just appends the first character of the current character set to the mask slice and goes to process the next symbol in order to not increase the mask slice keyspace any more (lines 13 through 15). On line 16, the algorithm finds the count of characters that the sent character set should have in order to best fit the desired keyspace. If that count is greater than or equal to the size of the character set, the oriignal symbol is appended to the mask slice without alteration (lines 17 through 19). Note however that this only happens if no characters were processed earlier and therefore discarded on line 12. In that case or if the character set is too large and appending it would lead to a mask slice with keyspace much larger than desired, the character set and the desired number of characters are passed to Algorithm 3, and its result is appended to the mask slice (line 21). On line 22, the algorithm then makes sure that only single characters will be appended to the mask slice from now on, since at this point, one of two things is true:

- The mask slice has roughly the keyspace that was desired, in which case increasing it further would be undesirable. In this case, forbidding the addition of further character sets is not strictly necessary as the algorithm wouldn't have done so in any case, but it is not harmful either.

- We have sent only part of the original character set, either because doing otherwise would have created a mask slice larger than desired, or because part of the character set has already been processed. In this case adding other character sets would leave holes in the mask sequence of the resulting mask slice, which must be avoided. This can lead to the creation of mask slices whose keyspace is significantly lower than desired, but this is an acceptable trade-off. Furthermore, thanks to the use of Algorithm 3, this only happens when a previous mask slice was of a smaller or equal size.

6.5 Elimination of ramp-up and limiting ramp-down

At present, Fitcrack implements a ramp-up and a ramp-down mechanism, so the progression of workunit sizes generally looks like in Figure 6.8. The gradual increase of workunit size at the start was implemented to deal with the inaccuracy of the benchmarks. The gradual decrease at the end, on the other hand, serves to ensure that as many computers are active for as long as possible [6]. This is because if there is, for example, one hour left, but the desired workunit running time is one hour, it probably means that all candidate passwords have already been assigned, and a new host would not be able to get any work. Likewise, if the amount of remaining work is not taken into account, it could happen that one host

is assigned an hour of work, while other hosts that ask a few minutes afterward will get no work and be idle while the first host keeps computing for another hour.

Figure 6.8: The intended workunit size progression in the original version without any alterations. Taken from [6]

.

Given that if the benchmarks are made more accurate, the reasoning for the ramp-up period will no longer be valid, I propose to eliminate it entirely. As for the ramp-down, it is useful, but it does not need to bring the workunit duration all the way down to one minute. However, given that Fitcrack offers the user the ability to choose the time a workunit should take in the phase labeled „main“ in Figure 6.8, there is no single constant that could be applied as a minimum for all jobs. I therefore propose to instead limit the ramp-down to a fraction of the job's configured time.

Chapter 7

Implementation of improvements

In this chapter, I will discuss the implementation of the improvements proposed in the previous chapter.

7.1 Adjusting power for salts and rules

I have unified the meaning of the number stored in the database to be the number of input candidate passwords that can be processed by the given host for the given job. This already takes into account the number of rules as well as the number of salts. This was done by altering `runner` to divide the result sent to the server by the number of salts reported by Hashcat, and then altering `assimilator` to further divide that number by the number of rules used by the job, if any. This is now uniform for all jobs.

In `generator`, all of the attack classes now get the desired number of passwords by simply multiplying the power stored in the database with the desired workunit execution time. Since now this already gives the number of desired input candidate passwords, no further processing is necessary.

7.2 Ramp-up and ramp-down

The implementation of my proposals from section 6.5 was mostly implemented by altering the `calculateSecondsIcdf2c` method of the `CAbstractGenerator` class. Other than that, it was just necessary to change the settings table in the database to contain the new configurable settings. Another student working on Fitcrack changed the GUI to enable setting the parameters in a user-friendly manner. The details follow.

Firstly, I have made the distribution coefficient configurable. It can now be set to a number between 0 and 1. Higher number means that the ramp-down will start later.

Secondly, I have introduced another configurable parameter into the algorithm, which I called the ramp-down coefficient. This is the fraction of the desired time per workunit configured for the job that serves as the new minimum workunit time for the ramp-down. By default, I have set this value to be 0.25, but it can be configured by the user to any value between 0 and 1, where 1 means that there is no ramp-down, while 0 means that the ramp-down is only limited by the hard-coded minimum workunit duration.

According to the Fitcrack maintainers, ramp-up was mostly implemented as a mitigation of the inaccurate benchmarks, as it would cap generated workunit size at the number of seconds since the job was started. This would make first workunits smaller and therefore

allow the size of workunits whose duration should be longer to be based on real performance as opposed to a benchmark. Since I have improved the accuracy of the benchmarks and their error is now mostly in single percentage points as opposed to the order-of-magnitude errors which were possible before, this is no longer necessary. I have therefore disabled this mechanism by default, but since the maintainers wished to give the users a choice to keep the behavior, there is an option to enable it.

7.3 Rewriting of the benchmark generation code

In Fitcrack, generating a workunit involves generating all the input files for `runner` and telling BOINC about them so it can send them to the client. For each attack, it is necessary to send the configuration file and the file with the hashes, but the rest of the files are dependent on the attack mode. For example, plain brute force attack requires no further files while a combination attack also requires two dictionaries. Different attacks also have different ways of splitting the keyspace into chunks of a given size. For example, with a dictionary attack, the server can simply send each client just the passwords it is supposed to process, while with a mask, it sends the mask along with instructions for `runner` to pass the `--skip` and `--limit` command-line parameters to Hashcat. For more details please see the Fitcrack technical report [9].

Due to these differences, each attack is implemented in a separate class, all of which are derived from the `CAttackMode` base class. In the old version of Fitcrack, benchmark workunits simply ran hashcat in its benchmark mode, which requires no inputs aside from the algorithm being cracked, so benchmarking was simply implemented as another class derived from `CAttackMode` and when a host needed to be benchmarked, `generator` simply created a workunit using that class instead of the usual one.

The new way of benchmarking I proposed involves running the attack as it would be run normally except it would be aborted once the cracking speed could be extracted. Keeping the same concept would have meant essentially duplicating the logic of each attack mode with a few alterations which would enable usage for creating benchmark workunits. Given that code duplication brings many problems, I have instead decided to reimplement the `CAttackBenchmark` class as a template class, which would be designed to take one of the other attack classes as a template parameter from which the instantiation would be derived. It would then override the behaviors which are undesirable in a benchmark workunit.

Specifically, it would have to change the type of workunit sent in the configuration file to benchmark and disable the mechanisms by which the workunit size is limited, as the correct size would not be known yet in any case. For dictionary attacks, it would also need to avoid sending the entire dictionaries unnecessarily and instead send a suitable representation of them that would enable `runner` to perform the benchmark without affecting performance, but at the same time be compact enough to not cause large delays by sending large amounts of data over the network just in order to obtain a power estimate. Finally, it would have to avoid making any changes to the database, unlike normal workunits which have to record which part of the keyspace they take in order for the subsequent workunits to be able to continue processing the rest of it.

Most of this I managed by introducing several methods into `CAttackMode` and then overriding them in `CAttackBenchmark`, but to avoid the database updates, I have instead created subclasses of database table wrapper classes `CDictionary`, `CMask` and `CJob`, and I overrode the update methods to do nothing.

7.4 Accurate runtime from runner

In the old version, all workunit running times were a multiple of 10 seconds. This seemed strange to me, and after some investigation I found that this is caused by the way `runner` communicates with the Hashcat process. Firstly, `runner` passes a command-line parameter to Hashcat instructing it to print status information every 10 seconds. At first, this seemed like the reason, but Hashcat does not actually wait for the 10 seconds to pass from the previous progress message when it finishes computation. Once that happens, Hashcat immediately prints one last status line and exits.

Further investigation revealed that `runner` created (or, in case of Windows, simulated) non-blocking pipes, but then it simulated blocking reads on top of that mechanism by putting itself to sleep for 10 seconds when no input was available in the pipe. I then checked the code for a reason for this behavior, but I found none and so I simply changed the reads to be blocking. After that change, `runner` reacted to Hashcat's exit immediately and the runtime reported by `runner` started agreeing with the time between the start and the exit of the Hashcat process.

7.5 Handling attacks with an external password generator

The original proposal for the better benchmarks and the experiments for it were done using Hashcat version 4. With it, running Hashcat for 3 seconds has always been enough to be able to extract cracking speed. While I was working on this thesis, the version of Hashcat used by Fitcrack changed to version 5. This new version sometimes failed to get the cracking speed after three seconds when the candidate passwords were supplied by an external generator. When this occurs, the speed reported afterward is typically reported as much higher than the actual speed. For this reason, when the attack uses an external generator, `runner` runs the benchmark for 30 seconds, getting the speed every 3 seconds. Then it removes all the instances where the reported speed was 0, and if there are enough nonzero values, it also ignores the highest and lowest of them. The value sent to the server is then the average of the remaining values.

Since the external generators used by Fitcrack also sometimes take some time to start generating output, I have made sure that the external generator is always started before Hashcat, and before the Hashcat process is started, `runner` waits until there is data available on the generator's output pipe.

7.6 Changes to the „bench all“ functionality

Fitcrack contains a functionality to benchmark all cracking algorithms on a given host in order to be able to estimate the total running time when the user is creating a new job. This uses the `--benchmark` argument of Hashcat and therefore is not always very accurate, but the maintainers want to preserve the functionality.

Previously, it received a list of algorithm identifiers from the server, and then created and ran a benchmark task for each of them. Given that after my modifications the benchmark task required actual inputs, another way had to be used.

Originally, I reimplemented the bench all task using Hashcat's `--benchmark-all` mode, in which Hashcat runs a benchmark for each algorithm and outputs the results. This would be the simplest solution, but unfortunately on a weaker computer, attempting to benchmark

one of the more resource-intensive algorithms leads to an „out of resources“ error. When this happens, Hashcat simply crashes, and so no further benchmarks are done, even though they may succeed. This meant that I had to implement the bench-all mode on my own.

Firstly, instead of receiving the algorithm to benchmark in a file, `runner` instead runs Hashcat with the `--example-hashes` command-line switch, which causes Hashcat to print every algorithm along with an example. From this, `runner` parses the algorithm identifiers and saves them to be benchmarked. The only problem was that for some reason, there is a `(null)` algorithm that caused crashes, but after filtering it out, no more problems occurred.

After that, `runner` simply runs Hashcat once for each of the algorithms and parses the result. It locks the global mutex in order to get accurate results, but unlocks it after every Hashcat execution in order to allow other instances of `runner` to run actual computation if there are any.

7.7 Changes in `assimilator`

Assimilator required very few changes. The only thing I had to do was remove the special handling of benchmarks for some attack modes along with implementing the division of the benchmark result by rules if they exist. These changes were already detailed in 5.2.

The only other change I have made in `assimilator` is in the host power estimate adjustment logic. `Assimilator` calculates the new power based on the workunit's reported size divided by the time taken by the computation. Previously, `assimilator` used the time reported by BOINC for this calculation, which made sense since previously the default was that the BOINC client requested a new workunit only after the previous was done and the results were sent to the server. Now that Hashcat execution is guarded by a mutex, the default was changed so that the BOINC client tries to have at least two workunits running at a time. Given that the BOINC reported time also includes the time `runner` spends waiting to acquire the mutex, this would result in greatly overestimated running time and therefore greatly underestimated power.

I have therefore changed `assimilator` so that the time on which the cracking power estimate is based is the time reported by runner itself. Given that this time is the running time of the Hashcat process, which only runs when the mutex is acquired, the sum of workunit times from a single computer should be the same as the elapsed real time minus the overhead of unlocking and locking of the mutex as well as the process startup time.

Chapter 8

Experiments

After implementing the proposed changes, I have performed several experiments to test how these changes affected the behavior of Fitcrack. They were performed in the C304 laboratory at FIT BUT using the computers present there at the time (named `h01 - h18` in the gathered data) along with two computers of my own: One called `Tom-ASUS`, which served as the Fitcrack server and, in some experiments, an example of a more powerful computer among many identical computers, as well as one called `tom-NC-E5-572G-728N`, which served as an example of a weak computer in those experiments.

There were two sets of experiments, with one set performed using the latest version of Fitcrack (the tip of the dev branch, and the other set performed using the commit before my first contribution to the repository along with some fixes that were necessary to get Fitcrack working with the computers in laboratory C304. For a detailed rundown of the changes introduced to the older version, see the `xzenca00_experiments` branch in the Git repository inside the `fitcrack` folder on the attached data storage device.

Given that the Fitcrack system is still in active development by multiple people at the time of the writing of this thesis, there were many changes made by other people, and I therefore cannot claim sole credit for the difference in the achieved results, but to the best of my knowledge, all the changes I will be focusing on in this chapter were made by me.

Throughout the chapter, I will describe the parameters of the attacks used in the experiments, and I will also specify the ids of the relevant jobs in the backups present on the attached storage medium. This is so that anyone who would want more data about the experiment can look it up in that backup to access all the information available about it.

8.1 Benchmark accuracy

First, I will focus on the accuracy of the benchmarks in the various modes. I expect that in most cases, the duration of the first workunit will be much closer to the desired workunit duration due to the changes in the way benchmarks are performed. Should there be no significant improvement, it should be because the benchmark results were already close to reality in the old version.

Table 8.1 lists the results of the experiments performed with the old and the new versions of the Fitcrack system in order to test the accuracy of the benchmark results. The attack configurations can be looked up by the job id in the dumped data on the attached memory storage medium, but for convenience, the values of relevant parameters used in the experiments are listed in Table 8.2. Table 8.1 contains for each version the time that was

	Old version			New version			
Job id	**Desired time [s]**	**Real time [s]**	**Diff**	**Desired time [s]**	**Real time [s]**	**Diff**	**Change**
35	60	16.240	−73%	300	332.80	+11%	62%
34	300	1445	+482%	300	360.45	+20%	462%
31	60	38.491	−36%	300	309.49	+3%	33%
30	300	355.57	+19%	300	291.71	−3%	16%
29	300	97.796	−67%	300	316.48	+5%	62%
28	300	124.93	−58%	300	315.64	+5%	53%
27	60	96.317	+61%	300	314.33	+5%	56%
39	300	3570	+1090%	300	323.98	+8%	1082%

Table 8.1: Comparison of desired and real workunit times

`generator`'s target time (as „Desired time“), the time the first workunit actually took to execute (as „Real time“) and the relative difference of the two times (as „Diff“). The first column then contains the id of the job used to perform the experiment, and the last column shows the difference between the relative error of the old and new version, calculated as $|diff_{old}| - |diff_{new}|$. The desired seconds per workunit were set to 300s in all cases, but under some configurations, the old implementation attempted to create 60s long workunits due to a bug in the adaptive scheduling algorithm that I have since fixed. This bug was caused by the calculation of the amount of the remaining work using the hashcat keyspace as the number of passwords, which is not true for all attack modes. All the jobs were assigned to a single lab computer, except for job 39, which was assigned to my own laptop.

With Job 35 it should be mentioned that the 16 seconds are actually quite misleading since the first generated workunit was sent with 1,760 candidate passwords and took 16 seconds to compute. Since Fitcrack uses the time taken to compute the workunit to improve its estimate of the host's cracking power, the workunit generated after the first workunit was done contained 4,934 candidate passwords and also took 16 seconds to compute, indicating that with workunits this small, the time taken was mostly the result of overhead. Through subsequent improvements of the estimated power, the workunit size has reached 53,744 candidate passwords by the time I terminated the job. This workunit took 48 seconds to compute. For comparison, on the new version, the first workunit took 332 seconds and contained 500,400 candidate passwords.

Job 39 demonstrates the influence multiple salted hashes can have on the estimated cracking power, which was not considered at all in the old version. This resulted in the old version sending a workload almost 12 times larger than desired with 20 different salts. I expect that the difference would grow roughly linearly with the number of salts given that hashcat essentially has to hash every candidate password with every salt.

Job 35 has multiple salts, so one might expect it to actually take longer, but due to the way benchmarks are handled in dictionary attacks (as outlined in Section 5.2), the negative effect of rules on the workunit duration far outweighs the positive effect of salts. It is quite similar with job 31, which is not affected by the cap on benchmarked power, but the workunit was smaller than it should have been according to the benchmark due to several factors. First, going strictly by the benchmarked power and the desired workunit execution time, the number of candidate passwords should have been about 367 billion for a 60 second workunit, but the workunit size was limited to the keyspace size of the PCFG,

which was only a little over one billion. The division by the number of rules was only applied after that, and so the workunit got approximately the correct number of passwords through pretty much coincidence.

Jobs 28 and 29 are not affected by rules, but they are approximately a third of the ideal size, which is consistent with the benchmarked power being reduced to a third in all brute force attacks.

Job id	Attack mode	Algorithm	Salt count	Keyspace
35	Dict+rules	md5($pass.$salt)	30	489,129,150,016
34	Dictionary	7-zip	1	14,344,384
31	PCFG+rules	md5($pass.$salt)	30	45,059,481,158,939
30	PCFG	7-zip	1	1,321,431,161
29	Mask	sha512($pass.$salt)	1	6,634,204,312,890,625
28	Markov	sha512($pass.$salt)	1	6,634,204,312,890,625
27	Combinator	sha512($pass.$salt)	1	205,761,352,339,456
39	Markov	md5($pass.$salt)	20	9,630,967,556

Table 8.2: Configurations for the attacks in Table 8.1

The experiments were carried out using fast hashes, except for the dictionary and PCFG experiments, where supplying a keyspace large enough to not be computed in its entirety in a few seconds using a fast hash was infeasible. The Dictionary and PCFG attacks (as well as the attack with job number 39, which was specifically meant to show the effects of multiple salts) with rules were carried out on multiple salted hashes. In the old version, this partially counteracted the effect of rules on the workunit size, which is described in section 5.2.

From the table we can see that for most attack configurations, the new version has error 5% or less. The exceptions are job ids 35 and 34 with relative errors of the execution time of 11% and 20% difference from the desired time, respectively. This is caused by the benchmarked speed being higher than the average cracking speed during the actual attack due to the benchmark workunit using the entire dictionary. With the old implementation, on the other hand, the lowest error is 19%. In no case is the old implementation better than the new one. Comparing the relative errors, the lowest difference in the relative error is 16%. Note, however, that the relative error of the old version is generally about 10 times higher than the relative error of the new version.

Overall, it can therefore be said that the accuracy of the new version of the benchmarks is far superior to the previous implementation. This enables me to get rid of the ramp-up period, so now (by default) the workunit size starts at or very near the size that it eventually settles at. This eliminates small workunits at the start, thus reducing the number of workunits and the associated overhead.

8.2 Dictionary fragmentation in combination attacks

Another experiment I performed in order to examine the fragmentation problems of combination attacks. It should demonstrate that in the old version of Fitcrack, `generator` could sometimes produce workunits which are far below the size the host (or any host working on the job) should receive, and that the new version does not exhibit this behavior. Specif-

ically, the new version of Fitcrack should never (except for the last workunit) generate a workunit whose size is less than the size of the smallest workunit created for the least powerful host, and this smallest workunit should never be smaller than two thirds of the size calculated by multiplying the weakest host's power by the desired time per workunit.

As for the attack configuration, the left dictionary contained 14344384 passwords, and the attack was carried out on a single PBKDF2-HMAC-SHA512 hash over a period of 20 minutes. The time per workunit was set to 60 seconds. After that, the job was stopped, which means that `generator` ceased creating new workunits, but the already generated workunits were allowed to run to completion. There was a total of 7 hosts assigned to the job: 5 computers from the lab and both of my own laptops. For more information, see job 37 in the new version, or job 38 in the old version.

The parameters of this experiment were chosen specifically to trigger the worst behavior of the old version of Fitcrack. As this occurs only when the left dictionary is fragmented, I needed a large left dictionary and workunits short enough that most of the hosts would cause `generator` to break the left dictionary into multiple workunits.

The school computers, which were in the majority in this experiment, were each able to try roughly 110,000 candidate passwords per second with these settings, and in the 60 seconds a workunit should take, they could therefore process about 6.6 million candidate passwords. This is slightly less than half of the left dictionary's keyspace, which means that they should cause the desired fragmentation, and also that the left dictionary should be split into two chunks, which together would have about 13.2 million candidate passwords, leaving only about 1.1 million passwords for the next workunit.

My own computers could process roughly 190,000 and 32,000 candidate passwords per second, respectively (therefore processing about 11.4 million and 1.9 million candidate passwords in 60 seconds). This meant that the more powerful computer would generally receive all the passwords remaining to be processed in the left dictionary when a workunit was generated for it while the weaker one may further split small leftovers.

In reality, due to various sources of overhead (for example, a reading of the Runner log files from several workunits suggests that Hashcat spent around 13 seconds starting up for this workload), the workunit keyspace at which the school computers reached execution time of 60 seconds with workunits whose keyspace size was between 4.9 and 5 million. This would enable splitting the left dictionary into three nearly same-sized parts with the last one being only slightly smaller, so further tuning of the parameters could probably lead to even worse results. Nevertheless, the difference is clear even with the parameters given above. Figure 8.1 shows the progress of workunits' keyspace size with the old version. The blue line is the single powerful computer, while the green line is the single weak computer. From the graph, we can see that the size of the weak computer's workunits remains fairly constant, while the other computer's workunit sizes differ wildly. The smallest workunit's keyspace size is 160,744, but it still took 17 seconds to execute. Out of the 108 completed non-benchmark workunits, 4 were smaller than 330,000 and the one with keyspace size of 329,944 took 18 seconds to execute. Two more workunits had keyspace sizes of less than 1 million, and four further workunits had keyspace sizes less than 1.5 million.

The average keyspace of hosts other than the weakest one (whose keyspace size is stable and doesn't change much) is 5,301,935. However, the first two workunits for each host are generated from the benchmark result, which was high enough that the workunits just received the entire left dictionary. This was, however, a one-time event. If we disregard these workunits and only count the workunits generated after the power estimate was updated from non-benchmark workunits, the average keyspace size drops down to only 4,192,105.

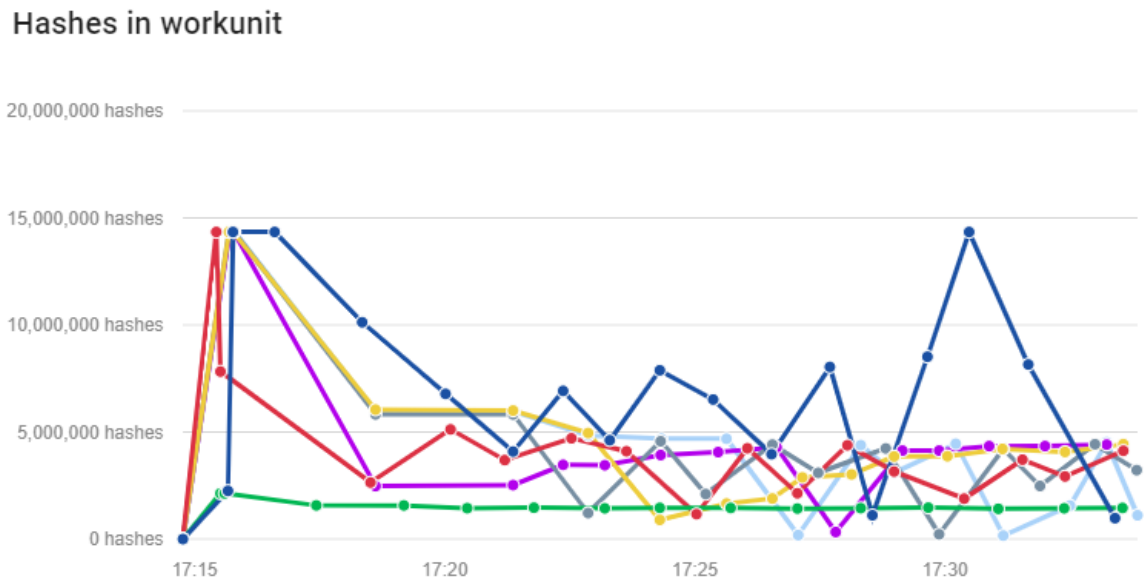


Figure 8.1: The sizes of workunits in a combination attack with the old version

If we only consider the workunits processed by the school computers, the average keyspace size is 3,532,858. This is only around 71% of the ideal keyspace size.

As for the overall efficiency of the attack, the total time spent computing the workunits was 1:44:39, and in this time the computers managed to process almost 516.4 million passwords. This means that on average, this attack was able to process 82,242 candidate passwords per second of GPU time. Figure 8.2 shows the progress of workunits' keyspace size with the new version. You can see that the variance of the workunit sizes due to the problems described in section 5.4 is significantly reduced. The smallest workunit assigned to a host other than the weakest one had keyspace size of 3131792. The average keyspace of hosts other than the weakest one is 6,367,029. If we further disregard the workunits whose size was calculated from benchmark results, the average keyspace becomes 5,380,761. For only the school computers, the average workunit size is 4,685,605, which is almost 94% of the ideal. 124 non-benchmark workunits were generated and 698,024,172 candidate passwords were tried. The total time spent computing workunits was 2:17:49. This means that on average, this attack was able to process 84,414 candidate passwords per second of GPU time. This is 2.5% more than with the previous version.

Note that this comparison should not be affected by workunit pipelining, as this compares the computation time, and not the overall running time. Nor should it be affected by better benchmarking, since the initial workunit size is the same in both versions.

Overall, the

8.3 Performance of the mask fragmenting algorithm

To evaluate the performance of the mask fragmenting algorithm, I have created another experiment. It was designed to verify that the mask fragmenting algorithm indeed has the characteristics I have predicted in section 6.4. Specifically, that it does not create workunits of wildly different sizes for the same workstation, and that the created fragment sizes scale in multiples of the first N character sets.

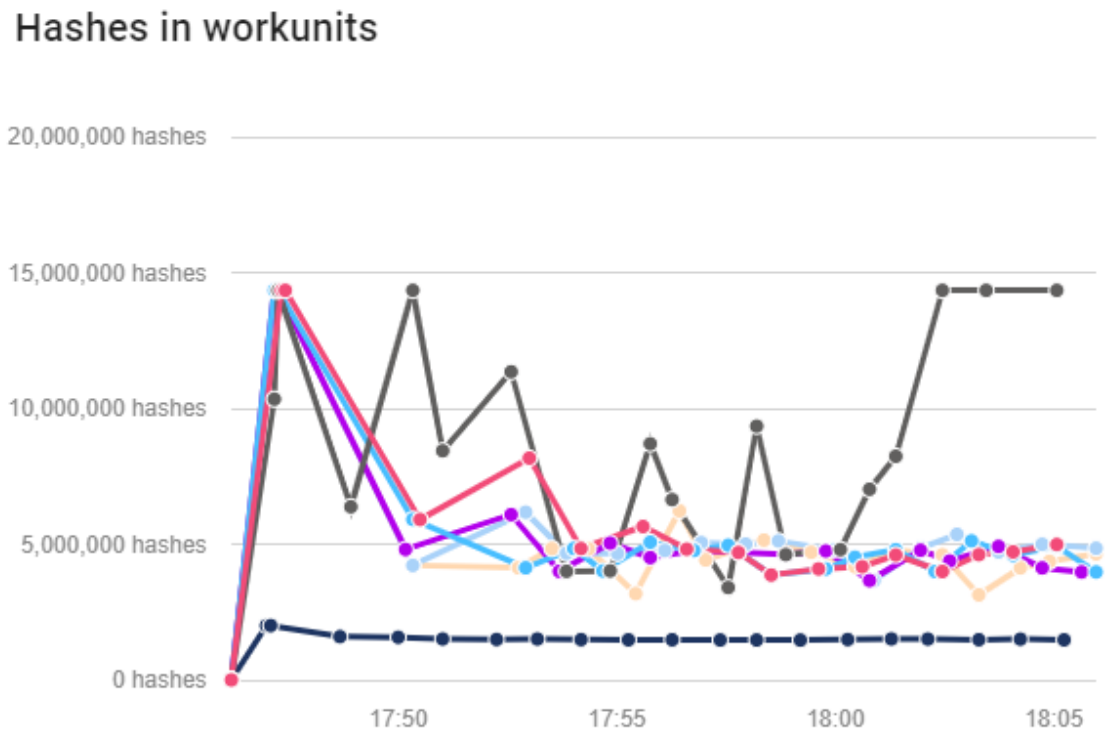


Figure 8.2: The sizes of workunits in a combination attack with the new version

The attack used for the experiment is a hybrid (wordlist+mask) attack on unsalted SHA-1. The desired workunit duration was 100 seconds, and the job used the rockyou.txt dictionary with the mask `?a?a?u?d`. A total of 7 computers were assigned to the task, 5 school computers, and both of my laptops, in order to make the algorithm deal with a variety of workunit sizes. For more details about the attack, see job 23.

Figure 8.3 shows the progress of workunit sizes through the job. The chart shows that there were a few workunit sizes which most computers received before the ramp-down started. The first is in a line near the top of the chart. This is a size where the first two character sets are sent in their entirety along with two characters from the third one. Workunits of this size are only generated for the most powerful computer. The second line is at half the size and is the first two character sets with one character from the third one. After that, there are two more. The lowest one results when the second character set is also fragmented for the weakest computer, and the one above is the more powerful computers finishing that character set.

The gap in the lower line is due to the weakest computer losing connection. The two workunits of the smaller size done by a more powerful computer are there because the workunits timed out and were assigned to another computer to compute.

Overall, the new fragmentation algorithm has eliminated the creation of extremely small workunits, and has increased the average workunit size significantly. Furthermore, the workunit sizes generated for this attack are completely in line with my expectations. This confirms that the new fragmentation algorithm performs as expected.

Figure 8.3: The workunit size progression in a hybrid attack in the new version.

8.4 Ramp-down

Another experiment was done to confirm that the ramp-down algorithm behaves as expected. In both versions, the workunit size should first be more or less constant until ramp-down begins, and it should stabilize again at different values. In the old version, it should stabilize at 60 seconds, while in the new version, it should stabilize at one quarter of the pre-ramp-down value.

The attack was a brute-force attack with keyspace of $408 \cdot 10^{11}$ and desired workunit time of 300 seconds, carried out on unsalted SHA-1 using four computers in the C304 lab. Brute-force attack was chosen due to its ability to fine-tune the desired keyspace as well as the fact that the splitting of a brute-force attack into workunits is done using the `--skip` and `--limit` command-line parameters of Hashcat. This means that the workunit size should be dependent solely on the host's power (which should remain constant for the duration of the experiment) and the desired workunit duration.

Ramp-up was removed from the server code since at the time I was also performing experiments to assess the accuracy of the benchmarks and I wanted to be able to compare the results with the same desired workunit time. This ultimately failed in some experiments due to other bugs which would have been more difficult to fix in the old version of Fitcrack, but unfortunately, I did not have time to perform the experiments again.

Figure 8.4 shows the shape of a mask attack in the old version. At first, there are smaller workunits due to the way mask attacks were handled in the old version. Then the workunit size levels off and starts ramping down towards 60 seconds, where it levels off again since 60 seconds is the minimum desired workunit duration in the old version.

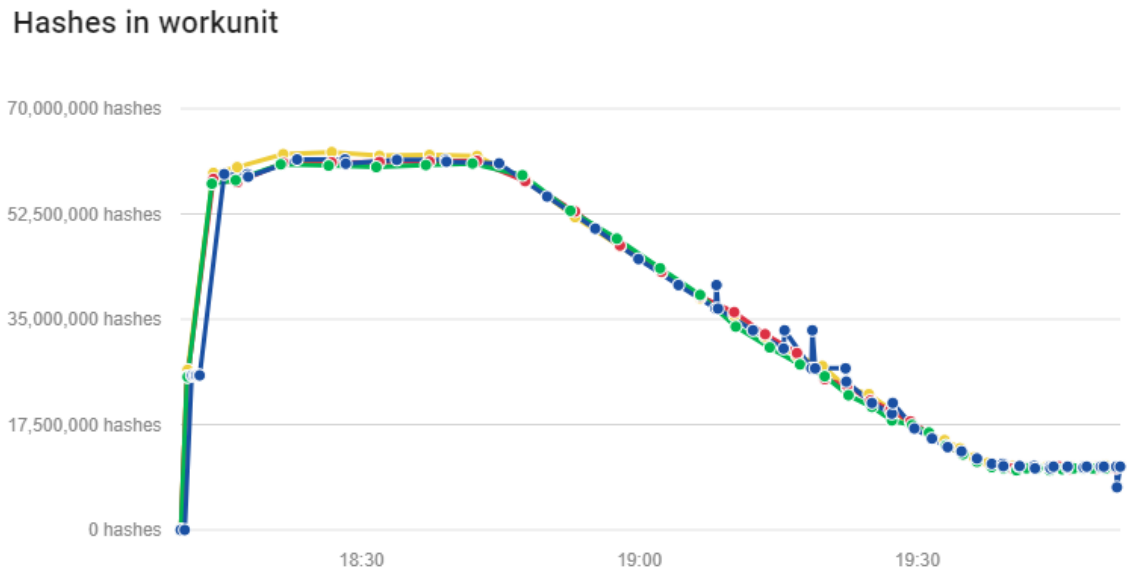


Figure 8.4: Shape of mask attack in the old version

Figure 8.5 shows the shape of a mask attack in the new version. It is mostly the same, but since the ramp-down coefficient is 0.25, the time per workunit levels off at 75 seconds instead of 60. Also note that it starts very near the size that it uses for all workunits until the ramp-down starts.

The above confirms that the new ramp down mechanism works as expected, which should reduce the overhead at the end of jobs with long desired workunit times significantly.

8.5 Efficiency of time utilization

In general, the efficiency of the computation itself should not be affected very much, as that is still left to Hashcat. The efficiency of utilizing the available time, however, should increase noticeably due to the fact that in the new version, the computation of the next workunit should start as soon as the computation of the previous one ends. The only times during which the assigned hosts should be idle while the job is running are therefore at the start of the job before the benchmark workunits are sent to them, between completing the benchmark and receiving the first workunits, and at the end when their assigned work is completed but other hosts are still computing. In the old version, in contrast, the host will also be idle between finishing one workunit and receiving and starting another. This is in addition to the delays affecting the new version.

To evaluate how this difference will affect the efficiency of time utilization, I have used the data from the experiment with ramp-down, as it is a brute-force attack and it ran to completion. I wanted a brute-force attack since it doesn't have to deal with dictionaries, external candidate password generators and other possible sources of overhead. The handling of mask attacks was also left almost unaltered, so the difference should only be caused by enabling workunit pipelining.

In the new version, the ramp-down experiment took 1:27:42 to run, and the sum of the workunit durations is 5:43:32. Given that the job was performed by 4 computers, ideally

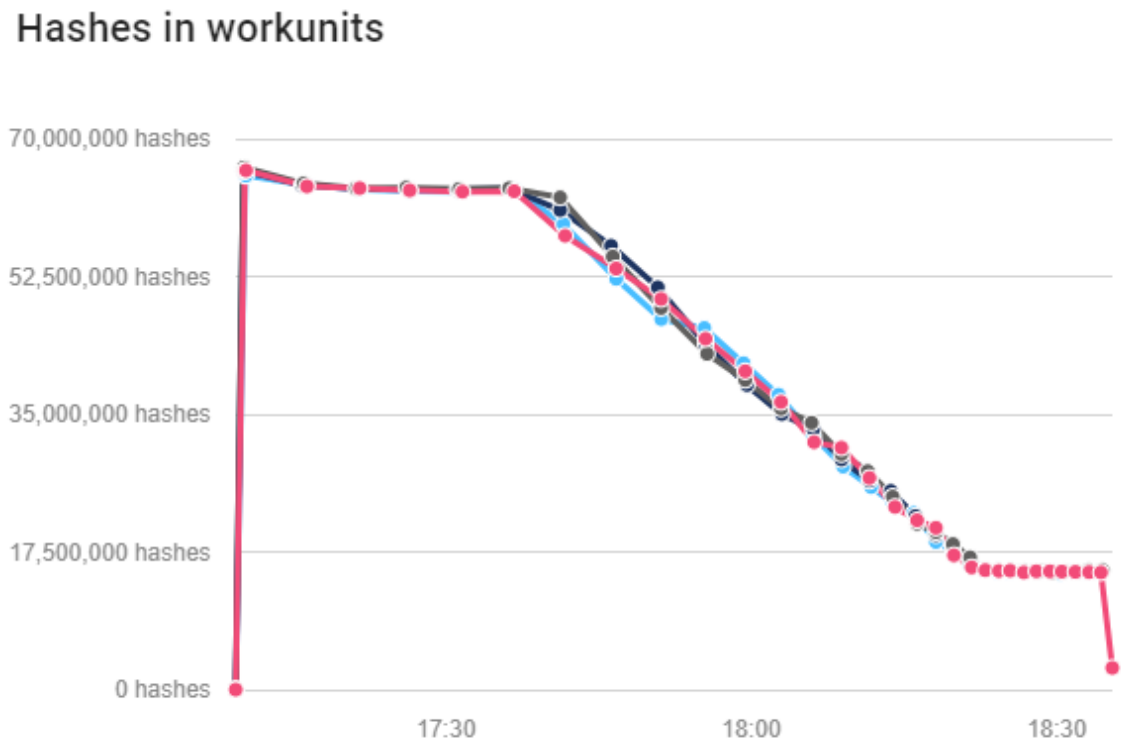


Figure 8.5: Shape of mask attack in the new version

the sum of the workunit times should be 4 times the wall clock time, which is 5:50:48. This means that the efficiency of time utilization was 98%.

Version	**Physical time**	**Ideal computing time**	**Real computing time**	**Efficiency**
New	1:27:42	5:50:48	5:43:32	97.93%
Old	01:43:48	6:55:12	5:49:46	84.24%

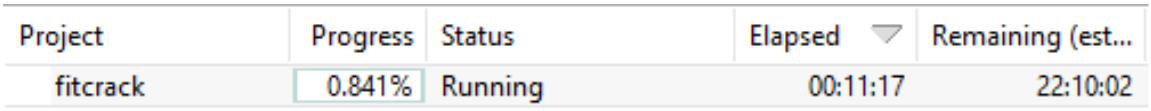
Table 8.3: Table showing how efficiently various attacks used the available time

Table 8.3 summarizes the efficiency of time usage in the ramp-down experiment in the old and the new versions. Unfortunately, in the Old ramp-down experiment, Runner sometimes failed to start on one of the computers resulting in the necessity of resending the workunit. Nevertheless, this only happened a few times and the failure was detected almost immediately, and so the result should only be affected by a few minutes.

It can be seen that the efficiency in the old version is just 84%, which is much worse than in the new version. Another interesting point of comparison is the time required to exhaust the jobs. The ramp-down experiment took 16 minutes longer to finish with the old version, which means that it took 18% longer than with the new version.

Overall, the new version of Fitcrack has significantly reduced overhead, which, in this experiment, was 109 seconds per computer. Overall, the computers only spent 2% of the time idle in the new version, as opposed to almost 16% in the old version, so the introduction of workunit pipelining by default was very successful.

8.6 Attacks with many rules and slow hashes



Project	Progress	Status	Elapsed	Remaining (est...
fitcrack	0.841%	Running	00:11:17	22:10:02

Figure 8.6: Illustration of the estimated workunit running time for a slow algorithm with many rules

Unfortunately due to the way Hashcat processes rules, attacks on slow algorithms with many rules can result in workunits which take far too long to complete, and whose computation time can only be reduced to a certain value by varying the candidate password count. Lowering the candidate password count further only slows down the computation and leads to little to no reduction in the workunit computation time. This can be solved by splitting the rules (and possibly the keyspace as is done currently) among multiple workunits.[1] Unfortunately, I only discovered the effect of this in Fitcrack quite late in the process of writing this thesis and did not have time to implement a solution.

Figure 8.6 illustrates the problem. It was taken from the first non-benchmark workunit generated in an attack on 7-zip with 34099 rules. It can be seen that processing the entire workunit, which was meant to be done in 300 seconds, was estimated to take over 22 hours. I aborted the experiment as letting the workunit run to completion would be far too time-consuming, not to mention pointless as Fitcrack sets a timeout on each workunit (currently a multiple of the desired workunit time, six times by default in the new version), and after it passes, the workunit is marked as failed.

The experiment was performed using the new version of Fitcrack, but there is no solution for this problem in the old version either.

8.7 Summary

The experiments demonstrated that the implemented improvements have succeeded in their stated goals. In the new version of Fitcrack, the accuracy of benchmarks is significantly improved, in some cases by orders of magnitude. The dictionary fragmenting algorithm used in the combinator attack no longer creates fragments that are far smaller than the desired size of workunits for even the slowest computer, and the average workunit sizes are now much closer to the ideal. The new mask fragmenting algorithm is tested and performs as expected. The ramp down is now limited to a user-configurable fraction of the desired workunit size, and due to the de facto elimination of delays between the computation of subsequent workunits the connected computers now spend almost 100% of the time actually computing workunits.

There is also a demonstration of a problem that has come to my attention late in the process of writing this thesis, and so was not fixed. Nevertheless, it can serve as a demonstration motivating further work.

[1] https://s3inlc.wordpress.com/2018/07/18/rule-splitting-in-hashtopolis/

Chapter 9

Conclusion

The goal of this thesis was to analyze the algorithm that Fitcrack currently uses for splitting a job's keyspace into smaller pieces that are then distributed to worker nodes, as well as the way these pieces are then distributed to said worker nodes, and to propose ways of improving found shortcomings.

9.1 Achieved results

This thesis presents several inefficiencies and defects found in Fitcrack's work planning algorithm and job distribution and proposes improvements that should remedy or lessen the impact of these problems. It also presents an analysis of the influence of variations in the workload on the cracking speed and proposes a way of obtaining a more precise initial estimate of a node's computational capabilities. It then discusses the implementation of these proposals and describes experiments done to evaluate the impact of the changes as well as their results.

As a result of the aforementioned changes, the workunits initially assigned to the worker nodes are now closer to the desired size and, in some workloads, the work transfer overhead should be reduced. The overall efficiency of the computation is also improved.

9.2 Future work

In future work, it would be beneficial to implement splitting of rules among workunits in order to solve the problem described in section 8.6. It may also be possible to further improve the accuracy of the benchmarks by running Hashcat several times with different keyspace sizes to take into account the fact that smaller workunits can also cause lower cracking speed and the relationship between the time required to compute a workunit and its size is not linear.

Another area which could be improved is the handling of failed workunits. Currently, when a workunit fails, it is given as is to the next host for which work should be generated. Should multiple workunits fail in quick succession and accumulate, they are given out for another attempt from smallest to largest. This process does not take into account host power, and can therefore result in great departures from the desired workunit execution time.

Due to time constraints, I was also not able to implement several of my proposals, specifically skipping passwords on the server by saving the file offset in addition to the password index and leaving the processing of small combinator fragments to weaker hosts.

Bibliography

[1] hashcat [hashcat wiki]. [Online; navštíveno 30.12.2019].
Retrieved from: https://hashcat.net/wiki/doku.php?id=hashcat

[2] mask_attack [hashcat wiki]. [Online; navštíveno 30.12.2019].
Retrieved from: https://hashcat.net/wiki/doku.php?id=mask_attack#disadvantage_compared_to_brute-force

[3] Password Storage OWASP Cheat Sheet. [Online; navštíveno 30.12.2019].
Retrieved from: https://cheatsheetseries.owasp.org/cheatsheets/Password_Storage_Cheat_Sheet.html

[4] Anderson, D. P.: BOINC: a system for public-resource computing and storage. In *Fifth IEEE/ACM International Workshop on Grid Computing*. Nov 2004. ISSN 1550-5510. pp. 4–10. doi:10.1109/GRID.2004.14.

[5] Hranický, R.; Holkovič, M.; Matoušek, P.; et al.: On Efficiency of Distributed Password Recovery. *The Journal of Digital Forensics, Security and Law*. vol. 11, no. 2. 2016: pp. 79–95. ISSN 1558-7215. doi:10.15394/jdfsl.2016.1380.
Retrieved from: https://www.fit.vut.cz/research/publication/11276

[6] Hranický, R.; Zobal, L.; Ryšavý, O.; et al.: Distributed password cracking with BOINC and hashcat. *Digital Investigation*. vol. 2019, no. 30. 2019: pp. 161–172. ISSN 1742-2876. doi:10.1016/j.diin.2019.08.001.

[7] Hranický, R.; Zobal, L.; Večeřa, V.; et al.: Distributed Password Cracking in a Hybrid Environment. In *Proceedings of SPI 2017*. University of Defence in Brno. 2017. ISBN 978-80-7231-414-0. pp. 75–90.
Retrieved from: https://www.fit.vut.cz/research/publication/11358

[8] Hranický, R.; Zobal, L.; Večeřa, V.; et al.: The architecture of Fitcrack distributed password cracking system. Technical report. 2018.
Retrieved from: https://www.fit.vut.cz/research/publication/11887

[9] Hranický, R.; Zobal, L.; Večeřa, V.; et al.: The architecture of Fitcrack distributed password cracking system, version 2. Technical report. 2020.
Retrieved from: https://www.fit.vut.cz/research/publication/12300

[10] Hranický, R.: Fitcrack - installation manual. [Online; navštíveno 30.12.2019].
Retrieved from:
https://fitcrack.fit.vutbr.cz/files/doc/Fitcrack_installation_manual.pdf

[11] Hranický., R.; Matoušek., P.; Ryšavý., O.; et al.: Experimental Evaluation of Password Recovery in Encrypted Documents. In *Proceedings of the 2nd International Conference on Information Systems Security and Privacy - Volume 1: ICISSP,*. INSTICC. SciTePress. 2016. ISBN 978-989-758-167-0. pp. 299–306. doi:10.5220/0005685802990306.

[12] ISO: *ISO/IEC 14882:1998: Programming languages — C++*. Geneva, Switzerland: International Organization for Standardization. September 1998. 732 pp.. available in electronic form for online purchase at `http://webstore.ansi.org/` and `http://www.cssinfo.com/`.
Retrieved from: http://webstore.ansi.org/ansidocstore/product.asp?sku=ISO%2FIEC+14882%2D1998;http://webstore.ansi.org/ansidocstore/product.asp?sku=ISO%2FIEC+14882%3A1998;http://www.iso.ch/cate/d25845.html;https://webstore.ansi.org/

[13] ISO: *ISO/IEC 14882:2011 Information technology — Programming languages — C++*. Geneva, Switzerland: International Organization for Standardization. third edition. September 2011.
Retrieved from: http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=50372

[14] Weir, C. M.: *Using Probabilistic Techniques to Aid in Password Cracking Attacks*. Dissertation. Florida State University, Department of Computer Science. 2010.
Retrieved from: https://diginole.lib.fsu.edu/islandora/object/fsu%3A175769

[15] Weir, M.; Aggarwal, S.; de Medeiros, B.; et al.: Password Cracking Using Probabilistic Context-Free Grammars. In *2009 30th IEEE Symposium on Security and Privacy*. IEEE. IEEE. 2009. ISBN 978-0-7695-3633-0. pp. 319–405. doi:10.1109/SP.2009.8.

Appendix A

The contents of the attached storage medium

The attached storage medium contains the following folders and files:

- bups - This folder contains the backed up data from the experiments I have done on Fitcrack, as well as scripts used to create and restore these backups. The scripts are `doDump.sh` and `doRestore.sh`. Both of them take the name of the backup as the single arument. The restoring script also prompts the user for the IP address that should be used to communicate with this Fitcrack instance. The backed up data consists of several backups, with each backup consisting of one `.tar.gz` file and one `.sql` file.
- experimenty - This folder contains the scripts and data used to perform the experiments to determine how hashcat performance is affected by the inputs. It also contains the results of these experiments as well as some screenshots from other experiments.
- fitcrack - This folder contains the Fitcrack git repository
- hashcat - This folder contains Hashcat 4.2, which was the version used by Fitcrack when I started working on this thesis. It also contains the files Hashcat needs to run. It is needed by the experiment scripts from the `experimenty/` folder.
- analysis_queries.sql - This file contains several SQL queries that I have used to analyze the results of several experiments done to determine the effects my changes had on Fitcrack.
- runFitcrack.sh - This is a script used to run the Fitcrack system when it is installed.
- Thesis.pdf - This document as a PDF.
- Thesis.zip - The source code for this document.