BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF INFORMATION TECHNOLOGY
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF INFORMATION SYSTEMS
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

DIGITAL FORENSICS: THE ACCELERATION OF PASSWORD CRACKING

DIGITÁLNÍ FORENZNÍ ANALÝZA: ZRYCHLENÍ LÁMÁNÍ HESEL

PHD THESIS
DISERTAČNÍ PRÁCE

AUTHOR **Ing. RADEK HRANICKÝ**
AUTOR PRÁCE

SUPERVISOR **Doc. Ing. ONDŘEJ RYŠAVÝ, Ph.D.**
ŠKOLITEL

BRNO 2021

Abstract

Cryptographic protection of sensitive data is one of the biggest challenges in digital forensics. A password is both a traditional way of authentication and a pivotal input for creating encryption keys. Therefore, they frequently protect devices, systems, documents, and disks. Forensic experts know that a single password may notably complicate the entire investigation. With suspects unwilling to comply, the only way the investigators can break the protection is password cracking. While its basic principle is relatively simple, the complexity of a single cracking session may be enormous. Serious tasks require to verify billions of candidate passwords and may take days and months to solve. The purpose of the thesis is thereby to explore how to accelerate the cracking process.

I studied methods of distributing the workload across multiple nodes. This way, if done correctly, one can achieve higher cracking performance and shorten the time necessary to resolve a task. To answer what "correctly" means, I analyzed the aspects that influence the actual acceleration of cracking sessions. My research revealed that a distributed attack's efficiency relies upon the attack mode - i.e., how we guess the passwords, cryptographic algorithms involved, concrete technology, and distribution strategy. Therefore, the thesis compares available frameworks for distributed processing and possible schemes of assigning work. For different attack modes, it discusses potential distribution strategies and suggests the most convenient one. I demonstrate the proposed techniques on a proof-of-concept password cracking system, the Fitcrack - built upon the BOINC framework, and using the hashcat tool as a "cracking engine." A series of experiments aim to study the time, performance, and efficiency properties of distributed attacks with Fitcrack. Moreover, they compare the solution with an existing hashcat-based distributed tool - the Hashtopolis.

Another way to accelerate the cracking process is by reducing the number of candidate passwords. Since users prefer strings that are easy to remember, they unwittingly follow a series of common password-creation patterns. Automated processing of leaked user credentials can create a mathematical model of these patterns. Forensic investigators may use such a model to guess passwords more precisely and limit tested candidates' set to the most probable ones. Cracking with probabilistic context-free grammars represents a smart alternative to traditional brute-force and dictionary password guessing. The thesis contributes with a series of enhancements to grammar-based cracking, including the proposal of a novelty parallel and distributed solution. The idea is to distribute sentential forms of partially-generated passwords, which reduces the amount of data necessary to transfer through the network. Solving tasks is thus more efficient and takes less amount of time. A proof-of-concept implementation and a series of practical experiments demonstrate the usability of the proposed techniques.

Keywords

Forensics, password, cracking, acceleration, GPGPU, BOINC, hashcat, PCFG

Reference

HRANICKÝ, Radek. *Digital Forensics: The Acceleration of Password Cracking.* Brno, 2021. PhD thesis. Brno University of Technology, Faculty of Information Technology. Supervisor Doc. Ing. Ondřej Ryšavý, Ph.D.

Abstrakt

Kryptografické zabezpečení patří v oblasti forenzní analýzy digitálních dat mezi největší výzvy. Hesla představují jednak tradiční způsob autentizace, jednak z nich jsou tvořeny šifrovacích klíče. Zabezpečují tak často různá zařízení, systémy, dokumenty, disky, apod. Jediné heslo tak může tvořit zásadní překážku při zkoumání digitálního obsahu. A pokud vlastník tohoto obsahu heslo odmítne poskytnout, je pro forenzní experty jedinou možností heslo prolomit. Byť je lámání hesel principielně jednoduché, jeho výpočetní náročnost je mnohdy extrémní. Při složitějších úlohách je často nutné zkoušet miliardy různých kandidátních hesel, což může trvat dny či měsíce. A proto je cílem této disertační práce prozkoumat způsoby, jak proces lámání hesel urychlit.

Prostudoval jsem metody distribuce úloh mezi více výpočetních uzlů. Při vhodně zvoleném postupu lze dosáhnout vyššího výpočetního výkonu a snížit čas potřebný k řešení úlohy. Pro zodopovězení otázky, jaké postupy jsou „vhodné“ , jsem analyzoval aspekty, které ovlivňují zrychlení úloh. Můj výzkum ukázal, že efektivita distribuovaného útoku závisí na typu realizovaného útoku, tedy, jak hesla tvoříme, použitých kryptografických algoritmech, technologii a strategii distribuce. Práce proto srovnává existující řešení pro distribované zpracování a představuje možná schémata rozdělení výpočtu. Pro každý typ útoku práce diskutuje použitelné distribuční strategie a vysvětluje, které z nich je vhodné použít a proč. Navržené techniky jsou demonstrovány na prototypu ukázkového řešení - systému Fitcrack, který využívá technologie BOINC a nástroje hashcat jako „lámacího motoru.“ Přínos navržených řešení je demonstrován na řadě experimentů, které zkoumají zejména čas, výkon a efektivitu distribuovaných útoků. Součástí je také srovnání s distribuovaným systémem Hashtopolis, který také využívá nástroje hashcat.

Dalším způsobem, jak dobu výpočtu zkrátit, je snížit počet zkoušených hesel. Výzkumy ukazují, že uživatelé, pokud mohou, často volí taková hesla, která si lze snadno pamatovat a nevědomky tak následují množství společných vzorů. Ty je pak možné popsat matematicky. Matematický model může vycházet například z dat získaných automatickým zpracováním existujících sad hesel z nejrůznějších bezpečnostních úniků. Vytvořený model pak lze použít k přesnějšímu cílení útoků. Počet zkoušených kandidátních hesel tak můžeme zredukovat pouze na ta nejpravděpodobnější. Lámání hesel pomocí pravděpodobnostních bezkontextových gramatik tak představuje chytrou alternativu ke klasickému útoku hrubou silou, či slovníkovým útokům. Práce vysvětluje principy použití gramatik pro tyto účely a přináší řadu zlepšení existujících metod. Součástí je také návrh paralelního a distribuovaného řešení. Práce popisuje techniku, kdy distribuujeme větné formy v podobě částečně rozgenerovaných hesel, což snižuje množství přenášených dat. Díky toho můžeme úlohy řešit efektivněji a v kratším čase. Navržené řešení je demonstrováno prostřednictvím ukázkového nástroje a přiložené experimenty ukazují jeho použitelnost.

Klíčová slova

Forenzní analýza, heslo, lámání, zrychlení, GPGPU, BOINC, hashcat, PCFG

Citace

HRANICKÝ, Radek. *Digital Forensics: The Acceleration of Password Cracking.* Brno, 2021. Disertační práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Školitel Doc. Ing. Ondřej Ryšavý, Ph.D.

Digital Forensics: The Acceleration of Password Cracking

Declaration

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

. . . . . .
Radek Hranický
April 26, 2021

Acknowledgements

After seven years of my Ph.D. study and twelve years at the university in total, I consider my thesis finished. As this student chapter of my life is getting to an end, I want to give credit to all the admirable people who supported this intention. Thus, the names here cover not only my fellow researchers but also family and friends that encouraged me over the years. I feel they deserve some lines here.

I want to thank all members of the NES@FIT research group for being a great society of enthusiasts who helped and inspired me during my Ph.D. study. Let me accentuate some concrete names. Firstly, thanks to Ondřej Ryšavý for supervising my dissertation. Many times he gave me new ideas and always helped when necessary. Secondly, thanks to Petr Matoušek, the head of two research projects under which I conducted work that formed this thesis. Big thanks to Libor Polčák, who supervised my bachelor and master theses, offered me a research contract, and finally, convinced me to try the Ph.D. study. Next, thanks to Vladimír Veselý, a great neighbor and man who initially brought me to the password cracking area. Vladimír offered me to follow the work of his former student Jan Schmied. To finish up this office of three, I need to thank Matěj Grégr, a Linux guru with whom I was cooperating for some time. All these men were also teaching great Cisco academy courses that I attended and will never forget.

Let me now proceed to us, youngsters. Thanks to Martin Holkovič, a great friend who helped me create the original version of the Fitcrack system. In addition to Martin, I would like to thank Barbora Franková and Stanislav Bárta, co-researchers and friends from the old times of the Sec6Net project. Thanks to Jan Pluskal, originally a classmate, later the main coffee-master of NES@FIT. Thanks to Ondrej Lichtner, former classmate, Red Hatian, and sports enthusiast who runs like a pro. Big thanks to Kamil Jeřábek, a great buddy from the northern-Moravian region with whom I was teaching forensic courses and inspired me with the ultimate thesis-writing music. I also need to mention Marcel Marek, Filip Karpíšek, and Martin Kmeť, with who I spent some time in the office during my studies. Finally, thanks to newbie Ph.D. candidates Viliam Letavay, Michal Koutenský, and Martin Bednář. You all formed a fantastic community of researchers that I was happy to work with. Also, thanks to my friends outside our group: David Grochol, Ondřej Kanich, and others.

Next, I would like to thank all Fitcrack team members who joined me in creating a state-of-the-art password cracking system. Thanks to Lukáš Zobal, a Ph.D. student from NES@FIT, and my "right hand" in this team, who dedicated an outstanding amount of time to help me move Fitcrack to the current state. Thanks to Adam Horák, a VueJS guru, for redesigning the front-end of the WebAdmin. Thanks to Dávid Bolvanský for helping me implementing new attack modes. Thanks to Jiří Veverka, Jan Polišenský, Kristýna Jandová, and Michal Eisner for cooperation on bug fixing, creating various improvements, and being great friends. Finally, big thanks to former developers Matúš Múčka and Vojtěch Večeřa for helping me lay the foundation stones of the hashcat-based version of Fitcrack. Also, thanks to other contributors whose list is on the Fitcrack website. Besides Fitcrack, I give special thanks to Dávid Mikuš and Filip Lištiak, my former students who became co-researchers of the grammar-based cracking area and co-authors of some of my publications.

To conclude my alma mater, I need to thank the Department of Information Systems, supervised by Dušan Kolář, for being an umbrella for our research group. Thanks to Sylva Sadovská and Svatava Nunvářová from the Scientific Department for the guidance and opportunity to be the Ph.D. study ambassador. Finally, thanks to dean Pavel Zemčík and his collegium for the excellent cooperation.

Next, I would like to thank my mates from the original composition of the Student's chamber of the Academic Senate of BUT for being great friends and supporters: Pavel Maxera, Anna Kruljacová, Daniel Janík, Tereza Konečná, Kristína Šintajová, and Eliška Jarmerová. And I wish good luck to Viktor Konupčík, the president of SU FIT, who is taking my seat as the new representative of FIT BUT students, as my student days are getting to an end. Also, big thanks to David Sedlák for being the head of students in the faculty senate.

Let me now proceed to the names outside academia. First of all, I want to thank my family for all the love and support over the years of my study. Concretely, to my mother, Ivana, who always stayed with me in both good and bad times. During the lockdown period, she supported me with food and coffee when I was writing the final chapters. Thanks to my father, Radovan, who took me into sports and supported me over the years. As a medical doctor who flies the helicopter with the rescue service, I always saw him as a hero. Thanks to my younger sister, Monika, for support and all the adventures we experienced together. Big thanks to my younger brother, Michal, who is currently finishing his Ph.D. as well, and to my little brother Václav for wishing me good luck.

I want to express thanks to Zdeněk Brynda for introducing me to IT before my university study. Big thanks to Michal Myška, a former classmate, friend, and sparring partner in the gym. Thanks to Lucie Pavlásková for supplying me with Yerba Maté tea that stimulated my brain during the writing. Thanks to Martin Řezáč, a professional photographer, classmate, friend, and roommate for the university's first years. Also thanks to other droogies from these times: Martin Kunčík, Martin Lokvenc, Martin Klusoň, Lubomír Luža, and others. As conducting research requires other activities to refresh the mind, I would like to thank my buddies from our airsoft and military reenacting team: Martin Pohludka, Robert Hlavica, Jiří Lapka, and others. Since my other big hobby is dancing, I would like to give big thanks to my partners: Nela Kašparová, Brigita Arnoštová, Adéla Šoborová, and Jana Čapková for ballroom dancing. Silvia Mišáková for being my follower in Lindy Hop swing dance, academic senate colleague, and friend. And finally, Kristína Šintajová, a queen of BUT that introduced me to Salsa. Special thanks to Michaela Hantáková, an IT project manager who was always into great trips and hiking. And I would also like to thank Alicia Lozano, a post-doc researcher from Madrid, for being the best roommate ever.

The research contained in the thesis was supported by the following projects:

- *Modern Tools for Detection and Mitigation of Cyber Criminality on the New Generation Internet*, no. VG20102015022 granted by Ministry of the Interior of the Czech Republic,
- *Research and application of advanced methods in ICT*, no. FIT-S-14-2299 granted by Brno University of Technology,
- *Integrated platform for analysis of digital data from security incidents* project, no. VI20172020062 granted by Ministry of the Interior of the Czech Republic,
- *ICT tools, methods and technologies for smart cities*, no. FIT-S-17-3964 granted by Brno University of Technology,
- *National Programme of Sustainability (NPU II) project IT4Innovations excellence in science*, no. LQ1602 granted by Ministry of Education, Youth and Sports of the Czech Republic.

Contents

1 Introduction **5**
1.1 Background . . . 7
1.1.1 Early Password Protection . . . 7
1.1.2 DES and Password Hashing . . . 8
1.1.3 The First Cracking Tools . . . 8
1.1.4 The Revolution in Symmetric Cryptography . . . 9
1.1.5 Modern Password Cracking . . . 10
1.1.6 Advances in Cryptographic Protection . . . 12
1.1.7 Possibilities for Further Acceleration . . . 12
1.2 Research Goals . . . 13
1.3 Contribution . . . 13
1.4 Structure of the Thesis . . . 14

2 Password cracking essentials **15**
2.1 The Password Cracking Process . . . 15
2.2 Password Generation . . . 16
2.2.1 Exhaustive Search . . . 16
2.2.2 Dictionary-based Attacks . . . 16
2.2.3 Probabilistic Methods . . . 17
2.3 Password Verification . . . 17
2.3.1 Hash-based Password Verification . . . 17
2.3.2 Decryption-based Password Verification . . . 18
2.3.3 Checksum-based Password Verification . . . 19
2.4 Existing tools . . . 20
2.4.1 John the Ripper . . . 20
2.4.2 Cain & Abel . . . 21
2.4.3 L0phtcrack . . . 22
2.4.4 Hashcat . . . 22
2.4.5 Elcomsoft Password Recovery . . . 24
2.4.6 AccessData Password Recovery Toolkit . . . 24
2.4.7 Passware Kit . . . 25
2.4.8 Ophcrack . . . 25
2.4.9 RainbowCrack . . . 26

3 Distributed Password Cracking **27**
3.1 Motivation and Parallel Cracking Sessions . . . 28
3.1.1 Parallel Cracking . . . 28
3.1.2 Utilization of GPGPU . . . 30

3.1.3 The Limits of a Single Machine . . . 32
3.2 Related Work . . . 34
3.2.1 Early Work . . . 34
3.2.2 Cracking in HPC Clusters . . . 34
3.2.3 Non-HPC Solutions . . . 35
3.2.4 Commercial Distributed Password Crackers . . . 37
3.2.5 Hashcat-based Solutions . . . 38
3.3 Requirements for a Distributed Cracking Solution . . . 40
3.4 Frameworks for Distributed Computing . . . 41
3.4.1 MPI . . . 41
3.4.2 Apache Hadoop . . . 42
3.4.3 VirtualCL . . . 43
3.4.4 CLara . . . 44
3.4.5 BOINC . . . 45
3.4.6 Summary . . . 46
3.5 The Choice for the Cracking Engine . . . 47
3.6 Workload Distribution in Cracking Tasks . . . 50
3.6.1 Essentials . . . 50
3.6.2 Distribution Schemes . . . 51
3.6.3 Workunits in Fitcrack . . . 53
3.6.4 The Keyspace in Hashcat . . . 53
3.6.5 Adaptive Scheduling . . . 54
3.7 The Architecture of Fitcrack . . . 59
3.7.1 Generator . . . 60
3.7.2 Validator . . . 62
3.7.3 Assimilator . . . 63
3.7.4 Trickler . . . 64
3.7.5 BOINC Server Subsystems . . . 65
3.7.6 WebAdmin . . . 65
3.7.7 PCFG Monitor and PCFG Manager . . . 69
3.7.8 MySQL Database . . . 69
3.7.9 BOINC Client . . . 69
3.7.10 BOINC Manager . . . 70
3.7.11 Runner . . . 70
3.7.12 Hashcat . . . 70
3.7.13 Princeprocessor . . . 70
3.8 Attack Modes and Proposed Distribution Strategies . . . 71
3.8.1 Dictionary Attack . . . 72
3.8.2 Combination Attack . . . 75
3.8.3 Brute-force Attack . . . 78
3.8.4 Hybrid Attacks . . . 85
3.8.5 PCFG Attack . . . 89
3.8.6 PRINCE Attack . . . 92
3.9 Experimental Results . . . 96
3.9.1 The Time and Efficiency . . . 96
3.9.2 Adaptive Scheduling . . . 103
3.9.3 Distributed Dictionary Attack . . . 108
3.9.4 Distributed Brute-force Attack . . . 110

3.9.5 Distributed Combination and PRINCE Attacks . . . 112
3.9.6 Summary . . . 116

4 Probabilistic Password Models **118**
4.1 Motivation for Smart Password Guessing . . . 118
4.1.1 The Downside of Traditional Methods . . . 118
4.1.2 The Potential of Probabilistic Models . . . 119
4.2 Related Work . . . 119
4.2.1 Early Work . . . 119
4.2.2 Markovian Models . . . 120
4.2.3 Probabilistic Grammars . . . 121
4.2.4 The PCFG Cracker . . . 122
4.2.5 Motivation for Improvement . . . 123
4.3 The Scope of Improvements . . . 124
4.4 Probabilistic Context-free Grammars (PCFG) . . . 124
4.4.1 Creating Grammars from Dictionaries . . . 125
4.4.2 Letter Capitalization . . . 127
4.4.3 Sequential Password Guessing . . . 128
4.4.4 Probability Groups . . . 129
4.4.5 The Next Function . . . 129
4.4.6 The Deadbeat Dad Algorithm . . . 133
4.5 Key Observations . . . 136
4.6 Parallel PCFG Cracking . . . 137
4.7 Grammar Filtering . . . 139
4.7.1 Long Base Structures . . . 139
4.7.2 Calculating the Number of Password Guesses . . . 140
4.7.3 Rule Filtering . . . 141
4.8 Distributed PCFG Cracking . . . 143
4.8.1 Communication Protocol . . . 144
4.8.2 Server . . . 145
4.8.3 Client . . . 147
4.9 Experimental Results . . . 149
4.9.1 Parallel PCFG Cracking . . . 149
4.9.2 Grammar Filtering . . . 151
4.9.3 Distributed PCFG Cracking . . . 153
4.9.4 Summary . . . 159

5 Conclusion **160**
5.1 Achievements in Distributed Password Cracking . . . 160
5.2 Achievements in Probabilistic Methods . . . 161
5.3 Overall Summary . . . 162
5.4 Future Work . . . 162

Bibliography **163**

A An overview of password-protected formats **182**
A.1 Documents . . . 182
A.1.1 Portable Document Format . . . 182
A.1.2 Microsoft Office - up to 2003 . . . 184

A.1.3 Microsoft Office - Office Open XML . . . 185
A.1.4 OpenDocument . . . 187
A.2 Archives . . . 188
A.2.1 ZIP . . . 188
A.2.2 7z . . . 190
A.2.3 RAR . . . 192
A.3 Disk volumes . . . 193
A.3.1 TrueCrypt . . . 193
A.3.2 VeraCrypt . . . 194
A.3.3 CipherShed . . . 195
A.3.4 BitLocker . . . 195
A.3.5 PGP . . . 197
A.3.6 Mac Disk Utility . . . 197
A.3.7 FileVault . . . 197
A.4 Portable devices . . . 197
A.4.1 Android . . . 197
A.4.2 Apple iOS . . . 199

B The contents of the attached storage medium 202

Chapter 1

Introduction

Forensics is no more just lurking around the crime scene with a magnifier glass, taking fingerprints using a brush and powder. Over the years, the area evolved and introduced revolutionary methods like striation mark analysis or DNA profiling [197]. Alongside the technological revolution, digital forensics became a new branch of forensic science. The early attempts focused mainly on analyzing the contents of computer hard drives. Soon, the aim of interest extended from computers only to cellphones, digital music players, tablets, thumb drives, memory cards, and network traffic. Nowadays, digital forensics contains many sub-branches, including mobile device forensics, network forensics, or malware forensics [147, 39]. Moreover, it is not just "the police stuff" anymore. The importance of collecting and analyzing digital evidence is used in the industry as well, often connected to incident response and improving defense against cyber-attacks. Companies need to secure their computer systems and networks from both internal and external threats [11].

With the spread of digital devices, forensic experts face new challenges while seeking evidence. It is not just the growing amount of data and the popularity of cloud computing that complicates investigations [101]. There is another bogeyman in the area of digital forensics, and this bogeyman is cryptographic protection [11, 64, 65, 40]. A single password is often the only obstacle that prevents access to the most crucial pieces of evidence. Nevertheless, removing this obstacle is not always easy.

Investigators may try to obtain the password from its creator. Some countries even employ the "key disclosure law," which, under certain conditions, requires individuals to surrender cryptographic keys to law enforcement. In the United Kingdom, Part III of the Regulation of Investigatory Powers Act (RIPA) [201] requires the person to either decrypt the content or provide the encryption keys. The disclosure is mandatory even without a court order, which makes this law slightly controversial [41]. Many countries, however, do not use such legislation or have the exact opposite. The Fifth Amendment to the United States Constitution protects witnesses from being forced to incriminate themselves [200]. In the Czech Republic, the Criminal Procedure Act No. 141/1961 Coll. guarantees self-incrimination protection by defining the right to refuse testimony [46]. Either way, the investigators never have a guarantee the suspects eventually disclose the password.

Therefore, in many cases, the only suitable option is to obtain the password by force. Such a way of password recovery is often referred to as cracking. The principle is simple: guessing and verification. Imagine a child trying to open a common 4-digit mechanical lock, like the one shown in Figure 1.1. In our case, guessing is the physical movement of the digit wheels, while verification is the mechanical pulling of the shackle. Assume the child needs approximately two seconds to change the digits and a second to verify the combination.

Figure 1.1: A 4-digit mechanical lock

Then, checking all 10,000 options takes about 8 hours and a half. Yet, the child may be lucky and find the correct password in the first ten attempts. Guessing the correct password is like "looking for a needle in a haystack," and we often have no clue how close we are.

In digital cryptography, instead of mechanical locks, we have encryption functions like AES [50] and hash functions like SHA-3 [88]. Verification of a single password using "pencil and paper" may take a long time. Using a computer, the process is much faster. However, instead of 10 thousand, we may need to check billions of possibilities. With the complexity of today's algorithms, the process may take years if the password is strong enough. Fortunately, there are ways how to make the checking faster, and the aim of this thesis to accelerate the password cracking.

An approach that is not possible with a mechanical lock is to verify multiple passwords at the same time. Parallel processing brings substantial benefits, especially with the use of General-purpose computing on graphics processing units (GPGPU) [17, 221, 83]. And where the single-machine approach ends comes the distributed computing. Therefore, I describe and compare different options on how to distribute password cracking tasks between multiple nodes. The research covers scheduling, applying different attack strategies, and centralized control of a cracking network.

Secondly, checking all possible combinations is not always necessary. Let us return for a while to the example with the mechanical lock. If the child knows any clue like "there is number one in the first position" or "the combination contains number 7", he or she may finish much faster. If the lock is configurable and a particular person chose the combination, the child may take this as a benefit. Research shows that human beings choose passwords that are easy to remember [30, 63], and often use the same password for multiple purposes [52]. In our use case, trying combinations like "1234" or "4444" could be a smart strategy. The child may also utilize knowledge about the password's creator. Using passwords with the owner's favorite numbers, year of birth, and other likely combinations may have a higher probability of success than just a mechanic guessing. Computer cracking may benefit from the same principles. Therefore, the thesis also aims at "smart" password guessing methods based on formal models like grammars and Markovian chains. Employing statistical analysis and mathematical probability allows guessing passwords more precisely [136, 213, 114]. In the thesis, I show various improvements to existing techniques and propose how to use them in a parallel and distributed password cracking trial.

1.1 Background

The need to deal with cryptographic protection is undisputable. To identify the most significant challenges in digital forensics, Al Fahdi et al. performed a survey undertaken by 42 forensic experts from law enforcement, industry, and academia. The 3 top identified issues included anti-forensics and encryption [11]. In a survey from Harichandran et al. with 99 participants, mostly from North America and Europe, encryption was identified as one of the three most crucial challenges. The growing importance of encryption is noticeable from the surveys performed by Forensic Focus that asked about the biggest challenge forensic investigators face today. In 2015's survey with five hundred respondents, encryption was the second most frequent answer [64]. Three years later, encryption and anti-forensics techniques moved to the top [65]. Luciano et al. report similar observations after analyzing both qualitative and quantitative data from twenty-four cyber forensics expert panel members at the 2017's National Workshop of Redefining Cyber Forensics (NWRCF) [113]. Garfinkel also mentions a growing interest in anti-forensic methods involving cryptographic file systems, encrypted network protocols, and program packers [68]. Roussev discusses pervasive encryption as one of the six major issues in today's forensics [176]. Casey et al. report the increasing use of full disk encryption (FDE) and mention concrete cases where the encryption blocked further investigation [40]. Apparently, cryptographic protection creates a significant obstacle in forensic investigations. I assume such an obstacle should not be contemned and requires further attention.

1.1.1 Early Password Protection

The first documented use of password protection on computers dates back to the 1960s. MIT used passwords for securing user accounts in the Compatible Time-Sharing System (CTSS). The same system also encountered the first security breach. In 1966, Allan Scherr used a loophole in the system to print out the contents of the password file. Since the passwords were stored in a plaintext form, Scherr gained access to all user accounts [122].

MIT Bell Labs' Multiplexed Information and Computing Service (MULTICS) secured the stored passwords through a one-way transformation. The square root of a password was modified by the "AND" operation with a pre-defined mask to discard some bits. In 1972–1974, Paul Karger and his team from the US Air Force performed a security assessment and breached 90 percent of user accounts because their owners never changed the default password [204].

Robert Morris wrote the "crypt" program for encrypting files in UNIX systems. The file encrypter first appeared in the 3rd Edition of UNIX, and its improved versions are still used today. McIlroy states that the explicit intention was to stimulate code-breaking experiments, and Morris himself was able to break crypt by hand [121]. From the 6th Edition of UNIX, the crypt application encrypted user passwords. The implementation simulated the M-209 cipher machine from World War II. To eliminate its known weakness, creators used an inverted design. Instead of using the password as the plaintext, the password served as the encryption key. In 1978, Robert Morris published a study of possible attacks with a PDP-11/70 computer. A brute-force attack on a 5-character alphanumeric password would take a maximum of 318 hours. However, the same attack on a 6-character one could take over two years. Morris also suggested using a dictionary attack and proposed multiple hints like checking "a list of first names" or "all valid license plate numbers in your state" [131].

1.1.2 DES and Password Hashing

In 1976, the US National Bureau of Standards approved DES cipher as a federal standard authorized for encrypting all unclassified data [202]. Despite its original purpose being encryption, it is possible to employ DES for one-way hashing [123]. The 7th Edition of UNIX supported two implementations of the crypt. The first, inspired by the German Enigma machine, but in a simplified, single-rotor version. And the second based on DES [69]. In the DES-based version, the first eight characters of the user's password serve as the encryption key, and the algorithm encrypts a constant. The encryption is performed 25 times, and the resulting 64 bits are repacked to become a string of 11 printable characters. The password entry application was also modified to force users to choose less-predictable passwords. Morris also proposed using cryptographic salt as a 12-bit pseudorandom value obtained by the real-time clock [131]. At the time, hardware chips for computing DES became commercially available. Unfortunately for potential attackers, the "E bit selection table" [202] was wired into the chip. To prevent hardware-accelerated attacks, Morris proposed changing the table according to the 12-bit random number [131]. In 1980, Martin Hellman introduced the possible chosen-plaintext attack on DES using the precomputation of data [78]. It is the first known time-space tradeoff attack. In 1982, Ron Rivest improved the concept with the method of distinguished points that reduced the number of necessary lookup operations [174]. In System V and BSD 4.3 in 1988, developers introduced password shadowing, which restricted standard users from accessing password hashes [69].

Microsoft also used DES for calculating Lan Manager (LM) hashes. Passwords were converted to uppercase and filled in by zeroes to have 14 characters. The result was split into two chunks, each used as a DES key to encrypt a specific string. Such protection was very easy to crack since the attacker can exclude all lowercase characters. Moreover, passwords longer than eight characters can be cracked in two separate chunks [125]. In 1996, Windows NT 4 introduced the successor of LM, the NTLM, which uses the MD4 hash algorithm [171]. The new version uses Unicode encoding, but in contrast to UNIX, does not employ salt or multiple iterations [179, 125]. The LM or NTLM password hashes reside inside a Security Account Manager (SAM) database. The database is a hive file of the Windows Registry [56]. Windows NT 4.0 introduced the optional Syskey function [210, 89] for encrypting the SAM file. In Windows 2000, XP and newer, the function is enabled by default [128], and the entire hive is encrypted using the RC4 stream cipher [89].

1.1.3 The First Cracking Tools

The first publicly available password cracking tools were released in the early 1990s. Dan Farmer proposed the Computer Oracle and Password System (COPS) [62], the first vulnerability scanner for Unix systems. COPS contained the „pwc" utility for cracking weak user passwords. Alec Muffett decided to improve its memory management to increase performance and released the legendary Crack program [205]. Its 1991's version contained a programmable dictionary generator and even supported network distributed password cracking [133]. An alternative UNIX password cracker created for DOS and OS/2 was Cracker Jack, named after the author, Jackal. Both Crack and Cracker Jack were able to use values from the GECOS field in the password file [93].

To prevent cracking attacks, in the early 1990s, BSD Unix extended the crypt program with the support for longer passwords, multiple iterations up to 275, and the salt of 24 bits instead of 12. In 1994, FreeBSD's crypt was the first operating system to use the MD5 algorithm [172] for password hashing. The implementation used 1000 iterations and

a 48-bit salt. In the following years, most Linux distributions and the Cisco IOS adopted the same concept of iterated MD5 [160].

The AccessData Corporation, founded in 1987, is probably the first distributor of commercial password cracking software [129]. In 1995, the company offered five utilities for recovering document passwords [3]: WRDPASS for WordPerfect, DataPerfect, and Professional Write, LTPASS for Lotus 1 to 3, Symphony, and Quattro Pro passwords, PXPASS for Paradox and Symantec's Q&A, WDPASS for MS Word, and XLSPASS for MS Excel and MS Money. The utilities were running under a DOS or Windows environment. Each was available for $185 [2].

The year 1996 showed even more tools for cracking DES. The PaceCrack95 was created as a Windows 95 replacement for Cracker Jack. Qcrack published by the "Crypt Keeper" was probably the first available tool that supported the precomputing of hashes [160]. The tool offered high performance but required a lot of space for precomputed tables since each candidate password was hashed with 2^{12} possible salts. For each password, it was necessary to store an additional 4 kB of data. Therefore, 5000 precomputed passwords took about 20 MB of disk space [92].

In the same year, Alexander Peslyak, better known as the „Solar Designer," created John the Ripper tool as a replacement for Cracker Jack that was not maintained anymore and missed optimizations for x86 CPUs newer than 386. Peslyak completely redesigned the single crack attack mode. In addition to Cracker Jack's features, John the Ripper had an incremental brute-force attack mode [159]. In 1997, Peiter Zatko, better known as „Mudge" released L0phtcrack for Windows under L0pht Heavy Industries. It was the first tool capable of cracking NTLM hashes. Unlike John the Ripper, L0phtcrack had a graphical user interface [111].

1.1.4 The Revolution in Symmetric Cryptography

In 1998, The Electronic Frontier Foundation (EFF) presented a machine called EFF DES Cracker, nicknamed "Deep Crack," based on ASIC chips. In the same year, EFF broke DES cracking record in the RSA-sponsored DES Challenge II-2 by decrypting a DES-encrypted message after 56 hours of work. Six months later, in collaboration with distributed.net, the EFF won DES Challenge III by decrypting another message in 22 hours and 15 minutes. The main weakness of the DES is a short 56-bit key [203, 179]. As a response, in 1998, NIST proposed Triple DES (3DES), a block cipher that applies the DES cipher algorithm three times to each data block, supporting longer keys [102]. In the following year, however, Daemen and Rijmen proposed the Rijndael cipher [49] that, in 2001, NIST approved as Advanced Encryption Standard (AES) [50]. The new standard supporting encryption keys of 128, 192, or 256 bits quickly replaced DES and 3DES and is probably today's most widely used algorithm for symmetric cryptography. In 2018, NIST eventually deprecated using 3DES [22].

Today, AES is probably the most widely used algorithm for symmetric cryptography. The Windows 10's Anniversary Update finally changed the encryption of SAM files with NTLM user password hashes [89]. AES replaced the previously-used weak RC4 stream cipher. Operating system user passwords are not the only domain. Encryption is often used to secure documents, archives, disk volumes, and other media with sensitive content. In Microsoft Office 2007, AES replaced RC4 as well [214]. The same happened to Portable Document Format (PDF), starting from version 1.6 and Adobe Acrobat 7 [7]. The algorithm is also necessary for password verification and decryption of ZIP archives created by both

WinZIP [45] and PKWARE's SecureZIP [163]. AES replaced the original PKZIP stream cipher that was cracked by Biham et al. in 1995 [29]. AES is used for encrypting both RAR v3 [6] and v5 archives [175]. Another use is for disk encryption with BitLocker or Mac Disk Utility [183]. TrueCrypt and VeraCrypt also support AES as one of the multiple encryption options [218].

While AES serves for encryption of data, storing and verifying passwords is usually performed via one-way hash functions. In 1995, NIST published the Secure Hash Algorithm 1 (SHA-1), a more secure alternative to the existing MD5 [137]. The new function produces a 160-bit hash value in contrast with the MD5's 128 bits. In 1999, Provos et al. proposed bcrypt, a hash algorithm based on the Blowfish cipher. The function uses built-in cryptographic salting to prevent rainbow table attacks. Moreover, it provides a variable number of iterations, which allows to strengthen it over time to remain resistant against attacks [167]. Some Linux distributions like SUSE Linux started to use it by default for hashing user passwords [48]. In 2001, NIST published SHA-2, a family of hash algorithm standards developed by the NSA. The SHA-2 functions are much more secure than the original SHA-1 and produce digests from 224 to 512 bits in length [138]. After discovering vulnerabilities for collision attacks in 2008, MD5 is considered cryptographically broken and unsuitable for future use [55].

We can also observe advances in the process of key derivation, i.e., the process of making a fixed-size encryption key from a password. Formerly, the widely used method was the same DES-based scheme that Morris proposed for hashing UNIX passwords [131]. A noticeable change occurred when RSA Laboratories released the Public-Key Cryptography Standards (PKCS) #5 v2.0, also published by IETF, with PBKDF1 and PBKDF2 key derivation functions. To produce an encryption key from a password, the function needs five inputs: a two-input pseudorandom function, the password, cryptographic salt, the desired number of iterations, and the desired length of the derived key [98]. An official example of the two-input pseudorandom function is the Hash Message Authentication Code (HMAC), which requires a message, a key, and a cryptographic hash function like SHA-1 or SHA-2 [76]. The MD5-based HMAC is not recommended [199]. For the salt, NIST recommends a randomly generated value of at least 128 bits [198].

1.1.5 Modern Password Cracking

With the forensic experts' need for recovering password-protected content, commercial distributors started to develop cracking software. The tools mostly provide a user-friendly graphical interface and focus on the document, archive, and application passwords. The AccessData Corporation merged all their password recovery utilities into a single Password Recovery Toolkit (PRTK, see Section 2.4.6). The version from 2002 was able to crack multiple encrypted media formats, categorized into three levels by difficulty. The easy recovery within minutes was possible for Microsoft Office 95 and older documents, plus a few other applications like Lotus 123 Organizer or Ascend QuickBooks. Paradox and WordPerfect passwords had moderate difficulty that was crackable in hours or two days maximally. The most difficult yet supported formats were Office 97 and 2000, PKZIP, and PGP. The tool contained various dictionaries and customized suspect profiles for guessing passwords [129, 38]. Another password cracking pioneer is an Estonian company called Passware. Their Password Recovery Kit (see Section 2.4.7) from 1998 was capable of cracking Office 95 and 97 documents. An improved version available in 2002 was also capable of decrypting PDF documents, WinZip archives, and Windows NT/NTLM passwords [67]. In 2006,

a Russian company named Elcomsoft applied for two US patents: Fast cryptographic key recovery system and method [116], and Password recovery system and method [99].

A game-changer in hash cracking arrived in 2003 when Philippe Oeschlinn presented rainbow tables, inspired by the cryptanalytic time-memory tradeoff from Hellman and Rivest. Unlike classic lookup tables used in Qcrack, rainbow tables use the concept of chains and hash reduction that decreased the necessary storage space dramatically [142]. In the same year, Zhu Shuanglei started Project RainbowCrack [184]. The goal is the development of a general-purpose implementation of Oechslin's technique, precomputing hashes and maintaining a publicly available rainbow table repository. The project is still active today (see Section 2.4.9) [185].

In 2009, Jens Steube, known as "atom", decided to fix the missing multi-threading support in John the Ripper's dictionary attack mode. Therefore, Steube created the hashcat tool, originally called "atomcrack." The initial version was a simple yet very fast dictionary cracker [191]. At the same time, Peslyak and his team continued with the development of the John the Ripper tool and released new versions with enhanced features and optimizations for various architectures [158].

The release of NVIDIA CUDA [141] in 2007 and OpenCL [134] from the Khronos Group in 2009 allowed using GPU units for general-purpose computing. General-purpose computing on graphics processing units (GPGPU) brought a revolution to the password area and following years revealed its true potential. In 2008, Elcomsoft company, a commercial creator of password cracking solutions (see Section 2.4.5), introduced GPU accelerated computing of some of the supported algorithms. The same year, Graves proposed a solution for cracking NTLM and MD4 hashes with rainbow tables on GPU. The proof-of-concept tool was named IseCrack [72]. In 2009, Kipper et al. presented an implementation of AES for GPU [106]. In the same year, Zonenberg created a distributed CUDA-based MD5 cracker [221]. The first wave of GPU crackers included other, mostly free but later abandoned, projects: GPU md5 Crack, Multihash CUDA bruteforcer, Extreme GPU Bruteforcer, ISHASHGPU, and Bars WF Bruteforce. All supported MD5, and most of them also NTLM hashes or other algorithms. Bakker et al. compared the performance of Extreme GPU Bruteforcer, ISHASHGPU, Bars WF Bruteforce, and the commercial solution from Elcomsoft. All four tools showed a massive speedup on GPU in comparison to CPU cracking. The difference in performance between the tools was minimal [20].

In 2010, Steube released cudaHashcat that provided CUDA kernels for GPU-accelerated hash cracking on NVIDIA cards. In the same year, he created oclHashcat, an alternative with OpenCL kernels instead of CUDA. The main advantage of OpenCL is that it was supported by GPUs from both NVIDIA and ATI/AMD. The tool offered GPU-accelerated cracking of MD4, MD5, SHA-1, SHA-256, and NTLM hashes, Domain Cached Credentials, and passwords for MySQL and vBulletin applications. The oclHashcat offered advanced dictionary-based attacks like the rule-based attack, toggle-case attack, or combinator attack. Unlike Cracker Jack and John the Ripper, the new tool applied word-mangling rules on GPU kernel, which dramatically reduced the number of necessary PCI-E transfers [193]. The in-kernel rule engine is a unique feature, and I have not found any other password cracker with this capability.

In 2011, Sprengers presented a CUDA-based MD5 cracker. It was 104 times faster than John the Ripper that was still CPU-only [187]. The situation changed in the same year. Peslyak added CUDA kernels to John the Ripper. In 2012, he also added support for OpenCL [158]. For more about John the Ripper, see Section 2.4.1. In the same year, AccessData added support for GPU acceleration to their PRTK 7.0 and DNA 7.0 tools [34].

Later, Steube abandoned CUDA and merged the original tool's features with oclHashcat, creating a unified OpenCL CPU/GPU named simply hashcat. In the past years, Team hashcat won several years of DEFCON and DerbyCon Crack Me If You Can (CMYIC) contests [220]. The tool is the self-proclaimed „world's fastest password cracker". For more about hashcat, see Section 2.4.4.

1.1.6 Advances in Cryptographic Protection

Despite the revolutionary GPU acceleration and massive improvements in password cracking performance, attacking state-of-the-art cryptographic protection is still very difficult. Novelty algorithms and the increasing difficulty of password verification procedures complicate the cracking. Hashing with many iterations is costly for attackers. Due to the configurable cost factor, the bcrypt algorithm from 1999 is still in the game. Cracking bcrypt hashes with a higher number of iterations is extremely challenging. The obstacle made by high iteration counts is noticeable with other algorithms as well. Verification of a single password for an encrypted Office 2013 and 2016 document requires to compute 100,000 iterations of SHA-512 [126, 214]. For non-system partitions, TrueCrypt used 2,000 for RIPEMD-160 [54], and 1,000 for SHA-512 or Whirlpool [188] algorithms. However, its successor, the VeraCrypt, employs 655,331 iterations of RIPEMD-160 and 500,000 iterations of SHA-2 [218, 196].

A noticeable obstacle for attackers is the spread of PBKDF2 key derivation scheme [98], especially when combined with a strong hash algorithm like bcrypt or SHA-512. The use of salt eliminates possible rainbow table attacks, and every iteration is costly. PBKDF2 is not only used in the above-mentioned VeraCrypt, but also in WinZIP [45], SecureZIP [163], RAR v5 [175], or for full disk encryption on Android systems [195]. In 2017, IETF recommended using PBKF2 for password hashing [130].

In 2015, NIST released SHA-3, a novelty family of hash algorithms, internally different from its predecessors [139]. SHA-3 is based on the Keccak [27] concept of cryptographic primitives and uses a sponge construction where the input is "absorbed" into the sponge, and the result is "squeezed out" [139]. In 2016, Percival et al. released the scrypt algorithm for hashing and key derivation. The algorithm was intentionally designed to have high memory requirements to prevent massively parallel attacks. Like bcrypt, scrypt is also configurable. Three parameters control the complexity of hashing: N sets the CPU difficulty, r controls the memory difficulty, and p defines the parallelization difficulty [157].

1.1.7 Possibilities for Further Acceleration

With the increasingly complex cryptographic algorithms [139, 167, 157, 130], higher iteration counts [126, 218, 175], and stricter password-creation policies [166, 207], forensic investigators often reach a dead end. Even a multi-GPU machine with state-of-the-art tools and optimized algorithms may not be enough.

To move from this point-of-failure, I see two possible pathways. One solution is to put together a grid or cluster of machines to achieve the desired performance. The related work on the distributed password cracking is summarized in Section 3.2. The second way is to reduce the number of password guesses. We may only verify the candidate passwords that are likely to be correct. The question is how to determine such a subset of passwords. The answer may lie in probabilistic passwords models. Thereby, I present the work related to this subject in Section 4.2.

1.2 Research Goals

The scope of my research is exploring ways to accelerate password cracking tasks. In other words, proposing techniques that allow finding passwords in a smaller amount of time. The related research shows two possible pathways: a) parallel and distributed processing, and b) smarter guessing techniques using existing knowledge and time-space tradeoff to decrease the number of candidate passwords. Therefore, the objectives of this work are to:

- Study the characteristics of existing cracking tools, supported attack modes, and features. Focus on the use of GPGPU-based solutions. Explore the possible techniques to distribute the workload amongst multiple nodes. Analyze existing frameworks for distributed computing in terms of their suitability for password cracking. Define the requirements for a general-purpose distributed password cracking solution.
- Use the observations to propose algorithms and strategies that allow the distributed processing of cracking tasks. Focus on the performance, efficiency, and scalability of attacks. The solution should work with existing computer networks without requirements for specialized hardware. Create a proof-of-concept cracking system that implements the proposed methods.
- Experimentally evaluate the solution by performing a series of experiments under various settings. Check if the results match the initial requirements. If possible, compare the novelty cracking system with existing software.
- Study the methods for the time-space tradeoff attacks and utilizing the knowledge about existing passwords. Explore the use of statistical analysis and mathematical probability to model users' password creation habits. Choose a method and analyze the possibilities for its parallel use. Identify the weak spots and obstacles that complicate practical use.
- Propose improvements that eliminate the identified obstacles. Design a way to use the method in a multi-node GPGPU-based network. Create a proof-of-concept tool to demonstrate the proposed principles and experimentally verify its usability.

1.3 Contribution

The thesis describes different frameworks for distributed computing and discusses their usability in the area of password cracking. I analyze and evaluate possible ways for workload distribution in a multi-node computing network. I propose a technique based on the dynamic assignment of work chunks and an algorithm for adaptive scheduling of workunits. The algorithm reflects the current state of the network and withstands abrupt changes in computing nodes' performance [81, 87]. Moreover, I introduce pipeline processing of workunits that minimizes the overhead for network transfer and delays for switching between individual pieces of work [84].

While there are multiple commonly-used attack modes, it is necessary to take their properties and requirements into account. Inspired by the arsenal of attack modes provided by the hashcat tool, I propose a convenient task distribution strategy for each of them [84, 87]. Moreover, I present distribution techniques for two additional modes with external password generators: the PRINCE and the PCFG attacks. All strategies aim to divide the

problem efficiently between the computation nodes to minimize the overhead and utilize the maximum of available hardware resources.

I utilize the above-shown methods to create a general-purpose high-efficiency GPGPU password cracking system called Fitcrack [87]. The proof-of-concept solution uses hashcat tool as a client-side computing engine, and the BOINC framework [15] to handle host management, network communication, and work assignment. The design reflects the requirements denoted in Section 3.3. I experimentally verify that the new solution is capable of performing distributed attacks reliably and efficiently. Moreover, I compare the software with the Hashtopolis tool, underlying pros, and cons of each solution [84].

The thesis further contains a detailed study of the „smart“ cracking methods based on time-space tradeoff and formal models. The principle is to utilize the knowledge of users' password creation habits. Inspired by the research from Weir et al., I concretely focus on cracking with Probabilistic context-free grammars (PCFG) [213]. Motivated to make the technique more utilizable for practical use, I identified factors that influence the time of generating password guesses.

Firstly, the thesis shows that removing specific rewrite rules leads to a massive speedup of password guessing without having a considerable impact on the success rate. Secondly, I propose methods of parallel and distributed password guessing. The concept uses preterminal structures as basic units for creating work and supports parallel generating of strings. I demonstrate the idea by creating a proof-of-concept tool that also natively supports direct cracking with hashcat. Similarly to Fitcrack, the solution uses adaptive work scheduling to reflect the performance of available computing nodes. I evaluate the techniques in a series of experiments by cracking different hash algorithms using different grammars, network speeds, and numbers of computing nodes. By comparison with the naive solution, I illustrate the advantages of the new concept [82, 85].

1.4 Structure of the Thesis

The thesis is structured as follows. Chapter 2 summarizes the essential principles of password cracking, including the methods for password guessing and verification. Chapter 3 focuses on distributed processing. It analyzes possible ways of utilizing multiple nodes concerning different tasks and attack modes. The chapter contains a design of a novelty distributed password cracker and its experimental evaluation. Chapter 4 aims at “smart” password guessing methods. It describes multiple enhancements to the state-of-the-art cracking with probabilistic context-free grammars, including a parallel and distributed solution. Finally, Chapter 5 concludes the thesis. Appendix A provides an overview of the most common password-protected formats and describes concrete procedures for password verification. Appendix B describes the contents of the attached storage medium.

Chapter 2

Password cracking essentials

Password recovery is a process of obtaining passwords for accessing protected content. When performed by force, the procedure is commonly referred to as *password cracking* [117]. This chapter describes the basic cracking workflow, shows different schemes for verifying password candidates, and mentions existing software solutions.

2.1 The Password Cracking Process

In a nutshell, password cracking consists of two phases: a) password generation and b) password verification. These two phases repeat in a cycle with a finite number of iterations. Figure 2.1 describes the workflow of the process.

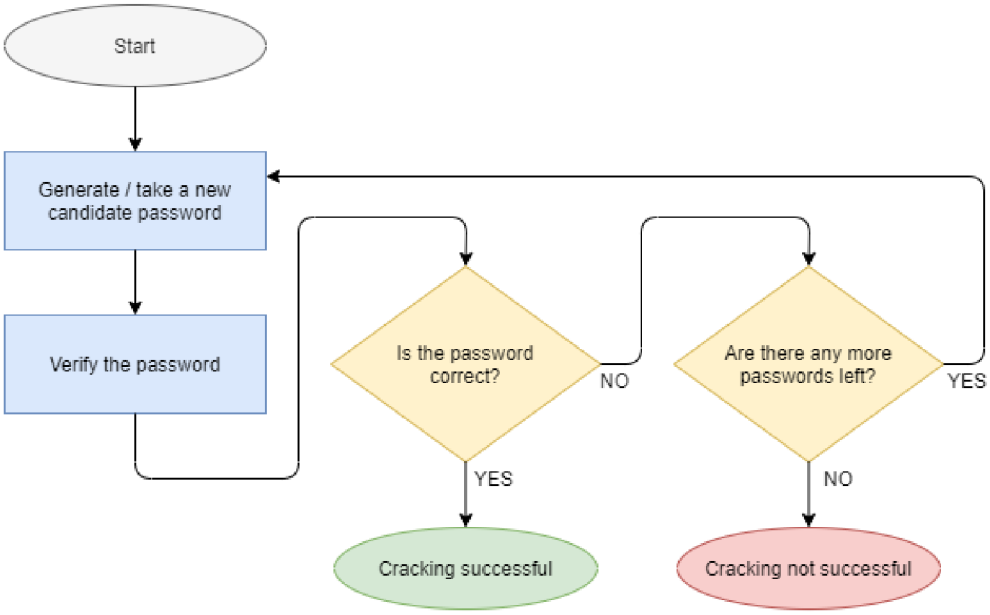


Figure 2.1: An illustration of the password cracking process

There are two types of password cracking attacks:

- **Online attack** - when attacking a live system, the attacker creates the candidate passwords, and the system serves as an oracle for password verification. The use-cases include attacking a website, a computer account, breaking electronic locks, or

guessing an ATM's PIN. The main drawback of such an attack is that the defender's security features can be active, e.g., the verification is temporarily disabled after reaching several attempts, etc.

- **Offline attack** - when the attacker has direct access to password hash or encrypted content, there is no need to communicate with the live system. Attackers can generate and verify the passwords on their own machines, e.g., using a GPU-equipped HPC cluster. An example is the password recovery of documents and archives found on a disk seized within a criminal investigation.

Both attacks have different strategies that could be employed, and in this thesis, I will focus mainly on offline attacks.

2.2 Password Generation

Password cracking examines a series of candidate passwords, also referred to as password guesses. An *attack mode* or attack type defines how these passwords are created. The password generation may utilize both existing string fragments or build entirely new ones from a pre-defined set of characters. The two approaches may also be combined. Smarter methods also introduce the use of mathematical probability and statistics to guess passwords more precisely. The *attack configuration* further specifies the guessing process. It may limit the length of strings, used alphabet, wordlists, and other details. The choice of attack mode depends on the attacker's decision and should reflect the situation. Is the password human-created or machine-generated? Was the password created to respect some policies: minimal length, at least one number, and special symbol, etc.? Do we know anything about the identity of the password owner, which may give us a hint? The attacker shall answer these questions before launching the attack.

2.2.1 Exhaustive Search

The exhaustive search is the core principle of brute-force attacks. The configuration contains one or more alphabets and a series of rules that define how to build strings from them. An alphabet is an ordered set of characters used for creating passwords. The classic incremental brute-force attack uses a single alphabet and creates every possible sequence of characters of a given length range. More advanced alternatives may utilize additional rules for specifying what characters are allowed in which position, etc. The main advantage of the exhaustive search is that, if appropriately configured, it eventually finds the correct password. The main drawback is usually the enormous number of candidate passwords. A more detailed overview of commonly-used techniques is in Section 3.8.3.

2.2.2 Dictionary-based Attacks

Dictionary-based attacks utilize existing wordlists of strings. The classic dictionary attack employs a single wordlist where each line represents a candidate password. In other words, generating password guesses is just reading a text file line by line. Some tools also support additional password-mangling rules that modify the strings before use, e.g., capitalize the first letter, swap or substitute some characters, etc. Concrete techniques are discussed in Section 3.8.1. Advanced approaches combine multiple strings. The combination may use fixed-position placements (see Section 3.8.2) or employ letter chains like the PRINCE

attack (see Section 3.8.6). Hybrid methods combine the dictionary-based approaches with exhaustive search. For example, a part of the password is from a dictionary while the other is generated using the brute-force technique. The hybrid attacks are described in Section 3.8.4.

2.2.3 Probabilistic Methods

Advanced state-of-the-art password guessing techniques often employ mathematical probability and use results of statistical analysis. Related algorithms are mostly based on formal models like Markovian chains or probabilistic grammars. Such methods are extremely efficient against human-created passwords since they can reflect the users' password-creating habits. They may utilize the knowledge obtained from previously-known passwords, the creator's country of origin, language, personal information, and other useful details. The entire Chapter 4 of this thesis is dedicated to probabilistic models and their improvements.

2.3 Password Verification

Once we get a candidate password, it is necessary to verify it for correctness. The verification procedure depends on the target of the attack. Cracking a root password to a MySQL[1] database requires an entirely different approach than breaking into an encrypted RAR archive [6, 175]. While proprietary applications often hide the internal implementation of password handling, open formats are usually well-documented, and the specification of necessary password verification steps are mostly publicly available. After analyzing dozens of password protection formats, I propose a classification into three commonly-used password verification schemes.

2.3.1 Hash-based Password Verification

The hash-based password verification is the most straightforward way in all scenarios where we can access the cryptographic hash of the correct password. Operating systems and web applications often store passwords in a hashed form instead of plaintext [10]. The motivation is to minimize the impact of a possible security breach so that the attacker does not instantly pick up passwords of all users. Once a user enters a password to authenticate, the application calculates its cryptographic hash and compares it with the stored one. If they match, the system grants the user access to given resources.

For password cracking, we can use the same verification procedure. Figure 2.2 illustrates the principle. The input candidate password serves as an input for cryptographic hashing. The calculation may require to compute one or more iterations of a single hash function or a combination of hash functions. Once we get an output, we compare it with the known hash, also referred to as the verification value. If they match, the password is considered correct.

This verification scheme is not limited to applications and operating systems only. Encrypted PDF and newer versions of MS Office documents (see Section A.1) and encrypted version 5 RAR (see Section A.2) archives also store the verification value. Therefore, for password verification, we do not need to decrypt the contents.

For cracking raw hashes easier, one may use the time-space tradeoff methods with precomputed hashes. In the case of shorter passwords, the lookup table [174] or rainbow

[1] https://dev.mysql.com/doc/refman/5.6/en/password-hashing.html

table [142] attacks allow for cracking the hashes almost instantly. Therefore, many formats employ cryptographic *salt* - a pseudorandom high-entropy value added to the password before calculating the hash. The salt makes the password longer so that there is a very low probability that it matches any pre-computed table. The salt needs to be stored together with the hash since it is necessary for the verification. NIST recommends the salt to be at least 32 bits long and arbitrarily chosen to minimize collisions among stored hashes [71].

A cryptographic *pepper* is another extra high-entropy value added to the password before hashing by some applications. Unlike salt, the pepper is stored secretly, separately from the hashes, typically inside the application that uses it. The use of pepper follows the NIST recommendations to use a secret value known only to the verifier. The recommended length is at least 112 bits [71].

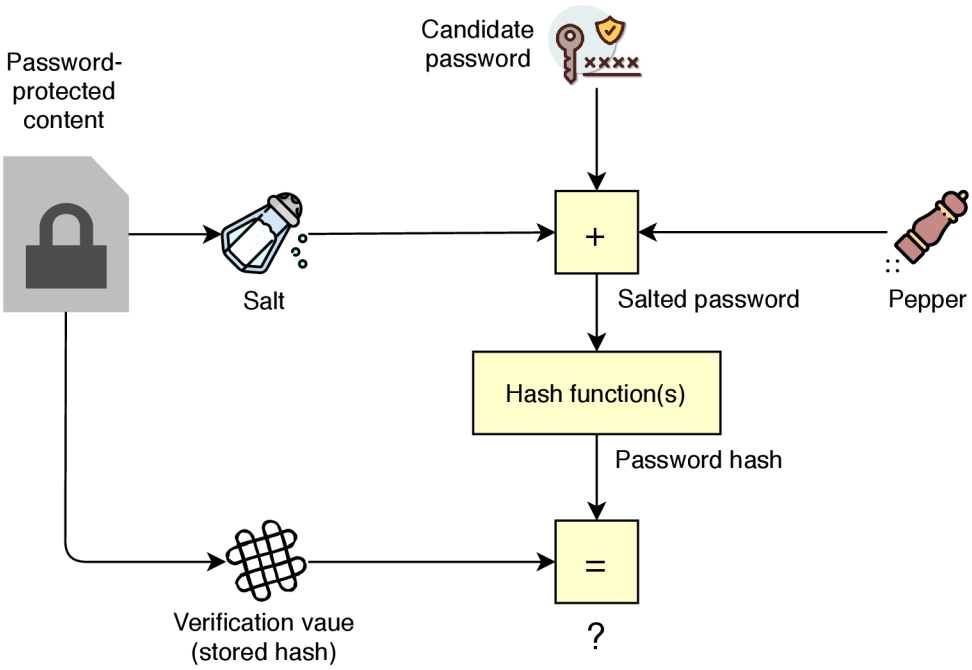


Figure 2.2: Hash-based password verification scheme

2.3.2 Decryption-based Password Verification

In many cases, there is no verification value with the stored password hash. In such a situation, the option we can perform a known-plaintext attack. First, we need to get an encryption key. The process is defined by the protected media format's manufacturer. A common practice is to use commonly-known key derivation functions like PBKDF2 [98, 130] that performs multiple iterations of a pseudorandom function. The key derivation function also often uses cryptographic salt or pepper. As the pseudorandom function, HMAC is typically used together with a hash function like SHA-2 or other [132]. Once we get the encryption key, we may decrypt the encrypted content. To perform the decryption-based verification automatically, we need to know a part of the plaintext. After the decryption, we check if there is the string we are looking for. If so, the password is correct. Where possible, we may decrypt only a part of the ciphertext. For example, with a block cipher

like AES [33], we can decrypt just the first block if we know what it should contain. The principle of the decryption-based password verification is illustrated in Figure 2.3.

An example of the decryption-based password verification is cracking disk volumes encrypted by TrueCrypt. For a candidate password, we generate an encryption key and decrypt the partition's header. Then, we check if the positions from 64th to 67th byte contain the word "TRUE" [219]. With VeraCrypt, the process is similar, but we look for the string "VERA" (see Section A.3.2).

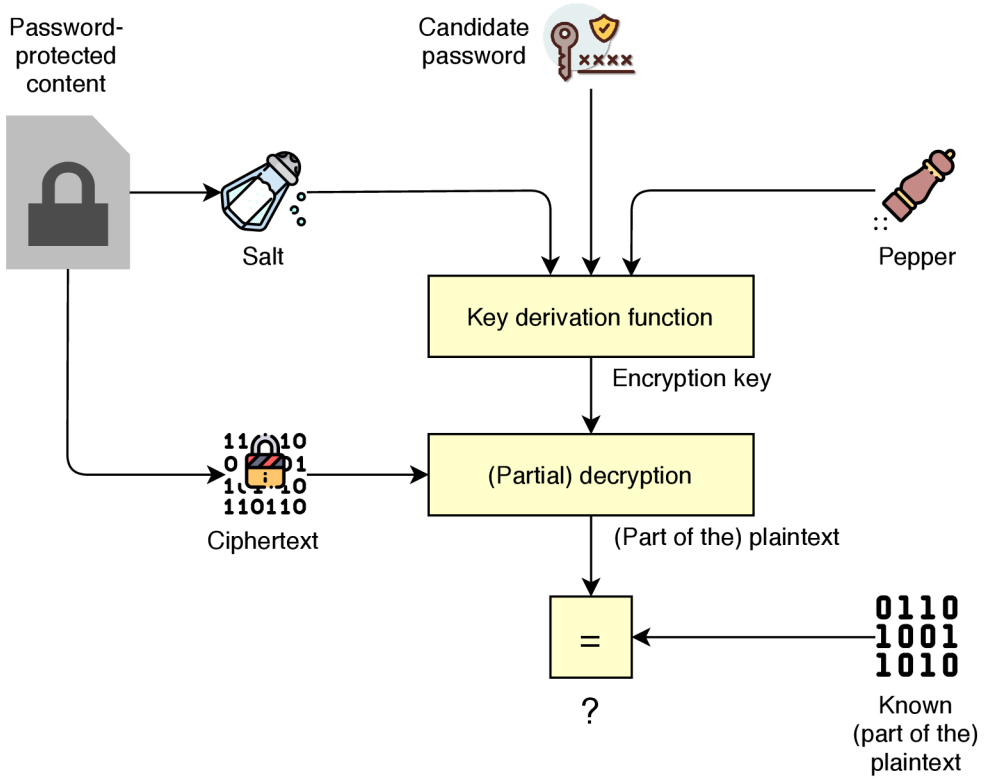


Figure 2.3: Decryption-based password verification scheme

2.3.3 Checksum-based Password Verification

In some cases, we have neither verification value nor a known part of the plaintext. Yet the protected media may include a checksum of its content or its part. In such a case, we use a checksum-based password verification, as illustrated in Figure 2.4. Like in the previous scheme, we first generate an encryption key. Then, we decrypt the content or its part. From the plaintext, we calculate a checksum using CRC or other error correcting code function [161]. The result is compared to the known checksum. If they match, the password is correct.

A typical example is a ZIP container encrypted with the original PKZIP stream cipher [163]. For each file inside the archive, there is a header with a CRC checksum of the file. For recovery, it is necessary to decrypt and decompress the data of at least one of the files inside the archive. Then, we need to compute a CRC checksum from it. For verification, we compare the computed checksum with the archive's checksum. If they match, the password is more likely correct. In this concrete case, there is, however, a risk of *false positive*

passwords (see Section A.2.1) since the checksum is relatively short and thus two different inputs may produce the same CRC code.

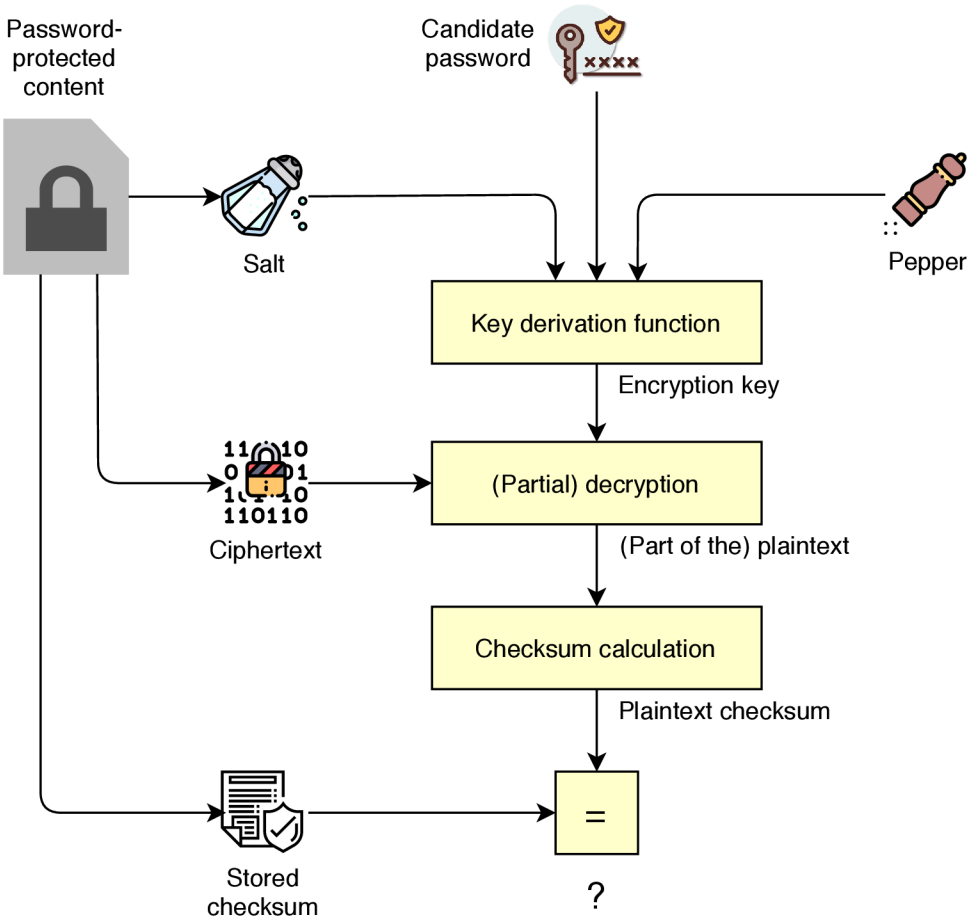


Figure 2.4: Checksum-based password verification scheme

2.4 Existing tools

This section lists some of the most popular solutions for password cracking. While few of the mentioned tools support distributed computing, the discussion on distributed cracking, including related work and existing solutions, is in Chapter 3. As mentioned above, I focus on offline cracking, so that I do not discuss popular tools for online attacks like THC Hydra[2] or NCrack[3].

2.4.1 John the Ripper

John the Ripper[4] (JtR) is a classic password cracking tool created by Alexander Peslyak[5], better known under the nickname "Solar Designer". Peslyak released the first version

[2] https://github.com/vanhauser-thc/thc-hydra
[3] https://nmap.org/ncrack/
[4] https://www.openwall.com/john/
[5] https://openwall.info/wiki/people/solar/bio

in 1996 as a drop-in replacement for the old Cracker Jack [93] tool for MS-DOS. John the Ripper supports four[6] attack modes: the wordlist mode, the single crack mode, incremental mode, and external mode [159].

The wordlist mode represents a classic dictionary attack that reads candidate passwords from a given wordlist. For this attack mode, John the Ripper provides the use of password-mangling rules that extend the repertoire of password guesses by modifications like letter capitalization, character swapping, and others. The mangling rules became so popular that they were adopted by other password cracking tools, including hashcat. The Fitcrack system, proposed in this thesis, also supports all of the JtR's mangling rules. The single crack mode builds password guesses from user names, GECOS / Full name fields from UNIX passwd files, and user home directory names. It also applies a large set of mangling rules. Successfully guessed passwords are also tried against all loaded password hashes just if more users have the same password. The authors suggest that „this is the mode you should start cracking with." The incremental mode is a brute-force attack that creates candidate passwords from frequency-sorted lists of characters. In the configuration, a user can define what character sets to use and the minimal and maximal password length. Finally, the external mode reads passwords from the standard input. Therefore, the user may employ an external password generator, connect it via a pipe, and use JtR as a backend cracker to verify password guesses. The tool is also frequently referenced in many scientific studies from the password cracking area [66, 109, 213, 211].

Starting from 2011, JtR provides GPU acceleration for a still increasing number of supported algorithms [158]. The 2019's version 1.9.0 jumbo can crack 287 different formats. Those include raw hashes, OS passwords, passwords for archives, documents, applications, and network protocols. From all supported formats, 88 utilize GPU-accelerated cracking.

John the Ripper is an open-source solution maintained by the Openwall Project. The source code is freely available on GitHub OpenWall repository[7] under a modified GNU GPL license. In addition to the freely-available distribution, there is a commercial Pro version with extended upgrades and enterprise support. It also contains a large multilingual wordlist for dictionary attacks. The Linux and MacOS versions of John the Ripper Pro are pre-built and distributed in native OS packages.

2.4.2 Cain & Abel

Cain & Abel is a popular free password cracking tool for Microsoft Windows OS with a user-friendly graphical user interface. The tool can recover passwords by sniffing on the network or performing dictionary, brute-force, and cryptoanalysis (rainbow table [142]) attacks. It can crack 26 different formats, including raw hashes like MD5 or SHA-2, Windows password hashes from LM to NTLMv2, passwords for network protocols like Kerberos or RADIUS, and various application passwords. Moreover, Cain & Abel incorporates many other existing tools into it. Therefore, other functionalities include recording VoIP conversations, performing an ARP cache poisoning attack and several man-in-the-middle attacks, decoding scrambled passwords, revealing password boxes, uncovering cached passwords, and even analyzing routing protocols. Thanks to these features and ease-of-use on Windows systems, the tool is supported by a relatively large fan-base [211].

Cain & Abel is not only a password cracking program but is also highly effective at collecting passwords and password hashes from targets on the local network. Despite the

[6] https://www.openwall.com/john/doc/MODES.shtml
[7] https://github.com/openwall/john

extra functionality, the actual cracking capabilities are too limited to withstand today's challenges. First, there is no GPU acceleration. Next, while the tool supports password-mangling rules, the only available modifications are case mangling, swapping characters, and appending characters to the end of each candidate password. The attacks based on frequency analysis, e.g., based on Markovian models, are not supported. On the other hand, it supports features that many other tools do not, r.g. creating rainbow tables [142], submitting password hashes to online lookup databases, etc.

The tool is distributed as closed-source freeware. The latest release 4.9.56 was published in 2014. Sadly, the project nowadays seems to be abandoned. The official website is empty, however the software is still publicly available[8].

2.4.3 L0phtcrack

L0phtcrack[9] is a commercial password auditing and recovery tool with a user-friendly graphical interface. The tool is designed mainly for penetration testers to perform security audits on company networks. It gained popularity in 2000 as the first password cracking tool capable of cracking MS Windows NTLM password hashes, but it also cracks Unix password files. The tool can retrieve the password hashes locally, remotely from a domain controller, or sniffing passwords off the network.

The initial release from L0pht Heavy Industries dates back to 1997 [111]. In 2000, the original manufacturer merged with ATstake, Inc. company. In 2004, the project was purchased[10] by Symantec Corporation, shut down, re-purchased by the original creators, and re-released.

L0phtcrack 6 supported a dictionary attack, brute-force attack, and substitution-based hybrid attack. For LM and NTLM hashes, the application also offered to use precomputed password tables. The included HashGen utility could calculate new hashes to extend the default set [165, 111] While the built-in cracking rules are more sophisticated than in Cain & Abel, L0phtcrack lacks the ability to define custom word mangling rules [211].

In 2020, Terrahash the L0pthrack was purchased[11] by Terrahash LLC, a company created by inventors of the popular hashcat tool and the creators of Hashstack, an enterprise distributed password cracking solution.

In L0pthcrack 7, released in 2019, the original cracking engine was replaced by the JTRDLL[12] fork of John the Ripper. Thanks to this change, the tool now supports GPU acceleration and the User info attack that corresponds to JtR's Single crack mode [110].

2.4.4 Hashcat

Hashcat[13] tool, initially created by Jens "atom" Steube in 2009, started as a high-speed freeware cracking solution with proprietary code. In 2015, the code was made publicly available under the MIT license, and it is nowadays an open-source project with a large fan base and a community of contributors. Originally, there were three tools: the legacy hashcat for CPU-based cracking, the oclHashcat with the OpenCL cracking kernels for NVIDIA/AMD cards, and the cudaHashcat dedicated for NVIDIA only. Very early versions

[8] https://github.com/xchwarze/Cain
[9] https://www.l0phtcrack.com/
[10] https://www.eweek.com/security/symantec-buys-security-consulting-pioneer-stake
[11] https://terahash.com/news/terahash-acquires-l0phtcrack.htm
[12] https://github.com/L0phtCrack/jtrdll
[13] https://hashcat.net/

of the oclHashcat also had two branches: oclHashcat-lite and oclHashcat-plus. The lite version was optimized for performance and supported only attack modes based on brute force. The plus version, on the other hand, was designed mainly for dictionary-based attacks. This changed[14] in 2016 when all three tools were merged into a single hashcat tool, version 3. The early versions are now considered deprecated and have been have removed from the official website. The cracking kernels are purely OpenCL-based and can be run on almost any OpenCL-compatible CPUs, GPUs, and even FPGAs, DSPs, and co-processors.

Unlike many other tools, hashcat does not have a graphical user interface and is therefore designed for advanced users. The only exception was "hashcat-gui" for the old 2010 to 2012 versions. While commercial tools regularly accept encrypted documents and archives as a direct input, hashcat does not. The tool is just for hashes and supports over 300 different algorithms. While hashcat can crack Office documents, ZIP, 7z, and RAR archives, encrypted disk volumes, or even cryptocurrency wallets, the user needs first to employ external utilities or "scraper" scripts to extract all necessary metadata from the password-protected medium. The extracted so-called "hash" has a common format and, in addition to the raw hash, may also contain other values like iteration count, cryptographic salt, version, key length, or even part of the encrypted and/or compressed content [95]. This is probably the main difference from out-of-the-box commercial password solutions where the user can just drag and drop the encrypted file and let it crack. With hashcat, you first need to extract the hash. Many of hashcat's hash formats are also compatible with John the Ripper tool.

The main advantage of the tool is its cracking performance. Hashcat is a self-proclaimed "World's fastest password cracker" and team hashcat won 7 of 11 of *Crack me if you can.* (CMIYC) contests[15] in the past 10 years [220]. Hashcat supports the following attack modes: the straight mode - a classic dictionary attack, the combination attack, the brute force attack, and hybrid attacks. In Section 3.8, I discuss each of these attack modes in detail. If a user chooses the straight mode and does not specify a concrete wordlist, the tool reads password. guesses from the standard input. This feature allows hashcat to use an external password generator. For dictionary attacks, hashcat supports password-mangling rules that are fully compatible with JtR [192]. Nevertheless, there is a difference in their application. While JtR modified the dictionary words on the host machine's CPU, hashcat's rule engine is implemented inside the GPU kernels. This way, hashcat can offer significantly higher performance for rule-based attacks. The performance and the range of supported formats motivated me to use hashcat as the "cracking engine" for Fitcrack distributed password cracking system that I propose in this thesis. Fitcrack supports all the hashcat's above-mentioned attack modes. For details, see Section 3.8. In 2020, hashcat developers announced a new mode called "association attack," which should be similar to JtR's single crack mode.

Hashcat is currently developed on GitHub, and the repository[16] has over 70 contributors. In addition to the source code, hashcat developers provide pre-compiled binaries for both Linux and Windows systems.

[14] https://hashcat.net/forum/thread-5559.html
[15] https://contest.korelogic.com/
[16] https://github.com/hashcat

2.4.5 Elcomsoft Password Recovery

ElcomSoft[17] Co. Ltd. is a manufacturer of password recovery solutions, mainly for governments, military and law enforcement customers. The company offers over 15 different tools for MS Windows. Each tool is dedicated to a single purpose: recovering Office passwords, cracking encrypted archives, PDF documents, and others. In addition to the set of single-purpose tools, Elcomsoft Distributed Password Recovery (EDPR), released in 2006, is an all-in-one solution for distributed cracking [59]. The company claims EDPR to have "linear scalability with low bandwidth requirements and zero overhead on up to 10,000 computers."

The tools support exhaustive search and dictionary-based attacks. For dictionary-based attacks, Elcomsoft supports several password-mangling rules like case mangling, letter replacement, or appending strings. The rules are partially compatible with John the Ripper tool [61]. Yet, it is not possible to define custom rules. The tools can perform brute-force attacks based on letter frequencies but do not support Markovian models or targeted brute-force attacks [211].

The main advantage of the Elcomsoft password recovery tools is a very user-friendly graphical interface. A high level of automation makes tools easy-to-use. Most of the crackers provide GPU acceleration, but not all of them. And also, not all support both NVIDIA and AMD cards. The benchmarks, however, show noticeably lower performance than with the hashcat tool [60]. In my early research, I tested three Elcomsoft tools. With my older OpenCL-based cracking tool, the Wrathion, I achieved higher performance for cracking Office, PDF, and AES-encrypted ZIP archives [83].

2.4.6 AccessData Password Recovery Toolkit

The AccessData[18] Group, Inc., founded in 1987, is mainly known for its Forensics Toolkit (FTK). The company, however, offers the password cracking solutions as well. The Password Recovery Toolkit (PRTK) contains dozens of modules for cracking different formats, including NTLM, Lotus, and Office password hashes. A Windows NT password recovery module can also reset password protection and secure boot options. AccessData is probably the first company to introduce distributed password cracking. In 2000, they released the Distributed Network Attack (DNA) for cracking Office Documents [1]. Newer versions provided hardware acceleration, originally FPGA-based. Since version 7, released in 2014, the software offers GPU acceleration using NVIDIA CUDA [34]. The software now supports over 70 different password-protected media formats. The GPU acceleration is, however, currently available only for MS Office and WinZIP [5].

The PRTK offers a detailed wizard for creating dictionary attacks and exhaustive search, including a targeted brute-force with Markovian models. It supports a wide and highly-customizable variety of password-creation rules. For creating word fragments, it offers multiple language profiles: European, Arabic, Russian, etc. Unfortunately, the PRTK can not use an external password generator [5].

[17] https://www.elcomsoft.com/
[18] https://accessdata.com/

2.4.7 Passware Kit

Passware Kit[19] is a commercial solution for the discovery and recovery of encrypted content. An early version from 1998 was capable of cracking MS Office 95 and 97 passwords [150]. The company later added support for other formats. In 2002, the solution was reference by Scott Gardener in IT World Canada [67]. The software provides a user-friendly graphical interface and is easy to use. It can perform a deep scan of the system to locate password-protected items and attempt to crack their passwords. For disk volumes encrypted by BitLocker, FileVault, or VeraCrypt, it can search through the operating memory and hibernation files for encryption keys.

Passware supports[20] six attack modes. First, it has a dictionary attack with advanced features like patterns and case modifications. The brute-force attack also supports patterns that resemble regular expressions. The proprietary Xieve attack is an optimized brute-force that uses letter frequency combinations and skips nonsensical sequences of characters. The mask attack is similar to hashcat's brute-force and allows specifying what characters to use at each position. "Known Password/Part attack" can be used in combination with other modes if the user knows part of the guessed password. Finally, "Previous Passwords attack" is simply a lookup for previously-cracked passwords [148].

There are multiple editions from Passware Kit Basic to Passware Kit Forensic with different pricing, capabilities, and supported formats. The most expensive edition supports over 280 different password-protected formats, including ZIP and RAR archives. Approximately half of them are crackable with GPU acceleration, while the rest is CPU-only. The Business and Forensic editions also support distributed computing where for each computing node called Agent, the user must buy a license. The Passware Kit also supports the rainbow table attack, but only for MS Office documents up to version 2003. Since 2010, Passware support distributed cracking with multiple machines. The main workstation runs the classic Passware Kit and the other stations connect using an application called Passware Kit Agent [151].

2.4.8 Ophcrack

Ophcrack[21] is an open-source password cracker from Phillipe Oechslin, the inventor of rainbow tables [142]. The tool can perform a rainbow table attack on Windows Lan Manager (LM) and NTLM hashes. It also offers a brute-force module for cracking simple passwords. From the tool's website, a user may download rainbow tables for Windows XP, Windows Vista, and Windows 7. As of September 2019, all tables are available free of charge.

The tool can dump and load hashes from encrypted Windows SAM files, perform an audit of passwords, and export the data into CSV. Moreover, the tool offers a graphical user interface with real-time graphs to analyze passwords. The graphs visualize the distribution of password complexity based on character sets, the distribution of password lengths, and overall cracking progress [143].

The sources are freely available under the GNU GPL license. Ophcrack runs Windows, Linux/Unix, and Mac OS X. It is available either as a standalone application or as a live CD

[19] https://www.passware.com/

[20] https://support.passware.com/hc/en-us/articles/115002145927-What-Password-Recovery-Attacks-can-I-use-

[21] https://ophcrack.sourceforge.io/

based on SliTaz[22] GNU/Linux. In contrast to RainbowCrack, Ophcrack does not support GPU acceleration.

2.4.9 RainbowCrack

Project RainbowCrack[23] was started in 2003 by Zhu Shuanglei, who created a general-purpose implementation of the rainbow table time-memory trade-off attack proposed by Phillipe Oschslin [142]. The project develops and maintains a series of tools and a large repository of rainbow tables.

The initial version supported only Microsoft Lan Manager (LM) hashes and contained three tools for 32-bit Windows: rtgen, rtsort, and rcrack. The *rtgen* serves for precomputing hash chains and generating rainbow tables. The *rtsort* tool can sort the tables. Finally, the *rcrack* is the cracker that can perform a lookup for a given hash [184].

Later releases introduced native support for NTLM, MD5, SHA-1, and SHA-256 hashes, computing on multicore processors, tables larger than 2 GB, and more compact `.rtc` format that reduced the required size by 50 % to 56.25 %. Shuanglei also provides utilities for converting the original tables to the new format. The tool also introduced support for 64-bit systems, not only Windows but also Linux. Users may either calculate rainbow tables on their own or buy precomputed tables for NTLM, MD5, or SHA-1. For both Windows and Linux, Rainbow Crack now supports GPU acceleration with NVIDIA CUDA and AMD OpenCL. Nevertheless, calculating on GPU only works with purchased tables. The Windows version also provides an optional graphical user interface [185].

Recent versions also allow for extending the range of supported hash formats. Users may create custom plugins for new algorithms. In the documentation, Schuanglei provides a simple tutorial and example sources [186].

[22] https://www.slitaz.org
[23] http://project-rainbowcrack.com/

Chapter 3

Distributed Password Cracking

Even with the most optimized cryptographic algorithms, password cracking with a single machine always has a boundary for achievable performance. Once we reach that boundary, the only way of getting over is by utilizing more nodes. As there are many options for distributed processing, it is necessary to choose methods that meet the requirements of a given task. Cracking passwords in a distributed environment requires a suitable strategy for distributing the workload and proper algorithms for controlling and scheduling the computing process.

The chapter starts with a brief introduction to parallel cracking that is essential to explain workload distribution principles. I discuss the limits of a single-machine approach to show the motivation for distributed processing. Next, the chapter contains an overview of related work from academia, hacker communities, and the commercial sector. I then define the requirements that a distributed password cracking solution should meet to withstand the current challenges. I study the usability of available frameworks for distributed computing, existing cracking tools, and possible techniques for the decomposition of cracking tasks.

Following the results of the analysis, I propose a method for task distribution and algorithms for scheduling work in a multi-node environment. I apply the novelty methods to design a distributed password cracking system called Fitcrack, based on the Berkeley Open Infrastructure for Network Computing (BOINC) framework and hashcat tool. The decision for BOINC is motivated by its flexibility, robustness, and automation of various tasks like updating client-side files or negotiation of system capabilities. The hashcat tool as the "cracking engine" is selected mainly for its performance, the range of supported formats, attack modes, and the portability to different platforms. I present the system's architecture, necessary server daemons to work with BOINC, client modules, communication between nodes, and others. Moreover, the chapter focuses on various attack modes and options. For each attack mode, I propose a unique distribution strategy that allows controlling and utilizing available resources efficiently.

Finally, I verify the usability of the proposed methods by performing a series of practical experiments with a proof-of-concept implementation of Fitcrack. The experiments focus on efficiency, overhead, scalability, and overall cracking time achieved in different scenarios. Moreover, I compare my solution to another existing hashcat-based distributed cracker, the Hashtopolis.

3.1 Motivation and Parallel Cracking Sessions

Password cracking is one of the most computationally-extensive problems in the area of digital forensics. The amount of time required depends on the algorithms involved, the strength of the password itself, and the available computational power. For instance, using an exhaustive search (brute-force attack), the maximum time in seconds (t_{max}) to find a password can be computed by Equation 3.1. The min is the minimum password length, max is the maximum password length, A is the alphabet used, and p is the cracking performance in passwords per second (p/s). For cracking hashes, p is often in hashes per second (h/s).

$$t_{max} = \frac{\sum_{l=min}^{max} |A|^l}{p} \tag{3.1}$$

For instance, let us consider cracking a password for a WinZIP archive encrypted by AES. As discussed in Section A.2.1, verifying every single password candidate requires calculating 1,000 iterations of salted HMAC-SHA1 [45]. The cracking performance I measured using Elcomsoft Advanced Archive Password Recovery 4.54 (see Section 2.4.5) and a single Intel(R) Core i7 CPU 920 was p=8,102 p/s [83]. In the worst case, finding an alphanumeric password made of 7 characters, i.e., min=1, max=7, and A=$\{a..z, A..Z, 0..9\}$ can take almost 14 years. For a forensic investigator, after such a long time, the information stored inside the archive could be useless. However, if we improve the cracking performance (p), the worst-case time necessary for finding the correct password is decreased. Therefore, I show how parallel and distributed computing can increase the overall performance and help obtain results in an acceptable time.

3.1.1 Parallel Cracking

The entire idea behind password cracking is to take several password candidates and verify each one of them. As described in Chapter 2, the password verification algorithm represents a finite sequence of operations defined by the type of protected content. For different formats, the procedure consists of different steps and requires different inputs like the number of hash function iterations, cryptographic salt or pepper, padding, etc. However, for a single cracking session, the only parameter that changes is the password itself. All other input values remain the same. Since there is no mutual dependence between different password candidates, we can verify them separately. This way, it is possible to divide a complex cracking problem into smaller subtasks that can be resolved simultaneously.

Imagine a protected content with a verification value in the form of hash derived from the correct password and salt. Figure 3.1 shows how the password cracking can be performed in parallel using N processor cores. In the beginning, we store the verification value and salt in the memory of each processor core. In every run, each processor core loads a password and concatenates it with the salt. Then, it calculates the hash of the salted password and compares the result with the verification value. If they match, the password is considered correct, and the cracking task ends. If not, the process continues until all passwords are examined.

Let c be the number of processor cycles required for verifying a single password, including loading of the password, returning result, and synchronization. Then, for N cores with frequency f, the theoretical cracking performance p_t in passwords per second can be calculated as described by Equation 3.2.

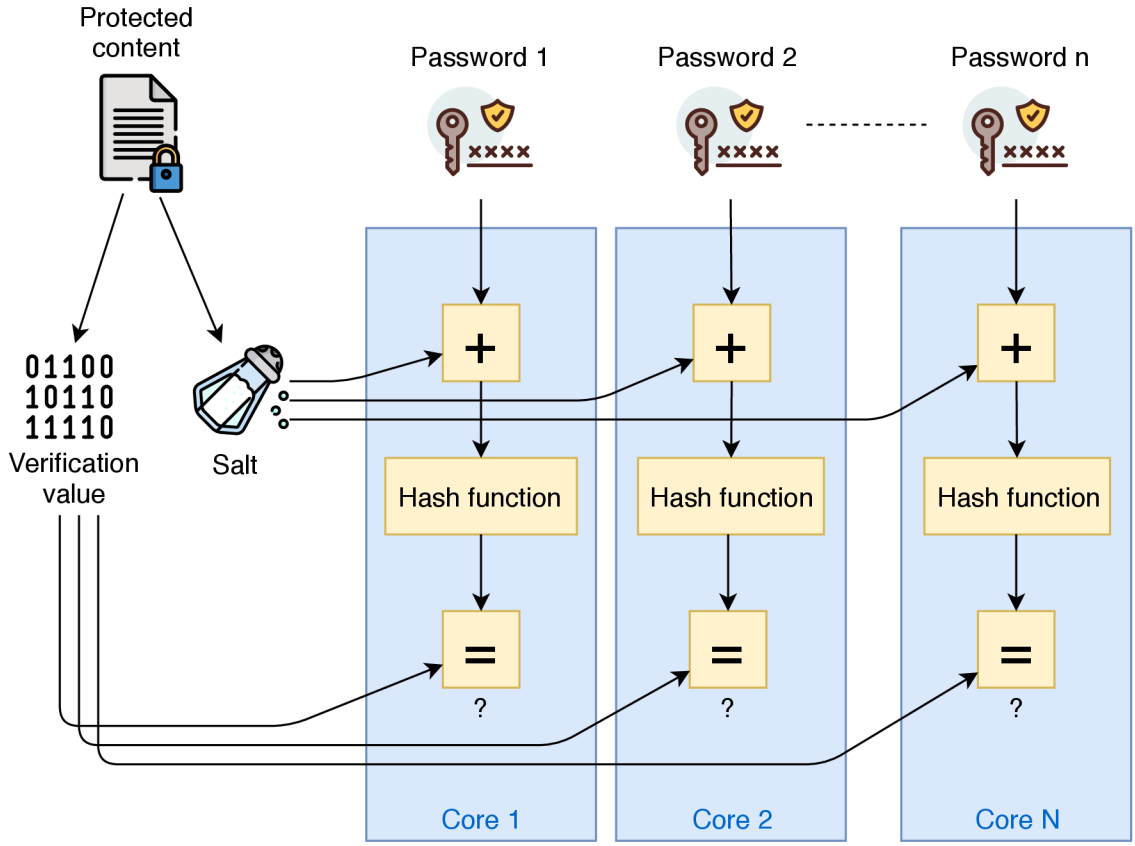


Figure 3.1: Parallel password cracking

$$p_t = \frac{f}{c} \cdot N \tag{3.2}$$

Please note that the calculation only represents a rough estimation since many other factors influence the actual performance. Those include the amount of memory, cache size, and others. While cracking raw hashes usually does not have high memory requirements, for formats where we need to decrypt part of the protected content (e.g., VeraCrypt, PKZIP, or RAR 3.0), the amount of available memory may become a bottleneck in the password cracking process.

Forensic investigators often face multiple cracking tasks at a time. In such a case, it may be helpful to use hashlists. A *hashlist* is simply a text containing a hash on each line while all hashes are of the same type (e.g., SHA-1). Many cracking tools accept hashlists as input and allow to perform a single attack on all the hashes inside. This approach is most beneficial if cryptographic salt is not in use. An example of parallel hashlist cracking is illustrated by Figure 3.2. For each password, the hash only needs to be calculated once. The result is then compared with every hash from the hashlist. Since the hash function's computation is typically the most complex part of the entire process, this technique may reduce the cracking time dramatically.

Unfortunately, with cryptographic salt, the solution is not that simple. For every hash, the hashlist contains a separate salt. Before the final comparison, we need to apply each salt to the password and recompute the hash function again.

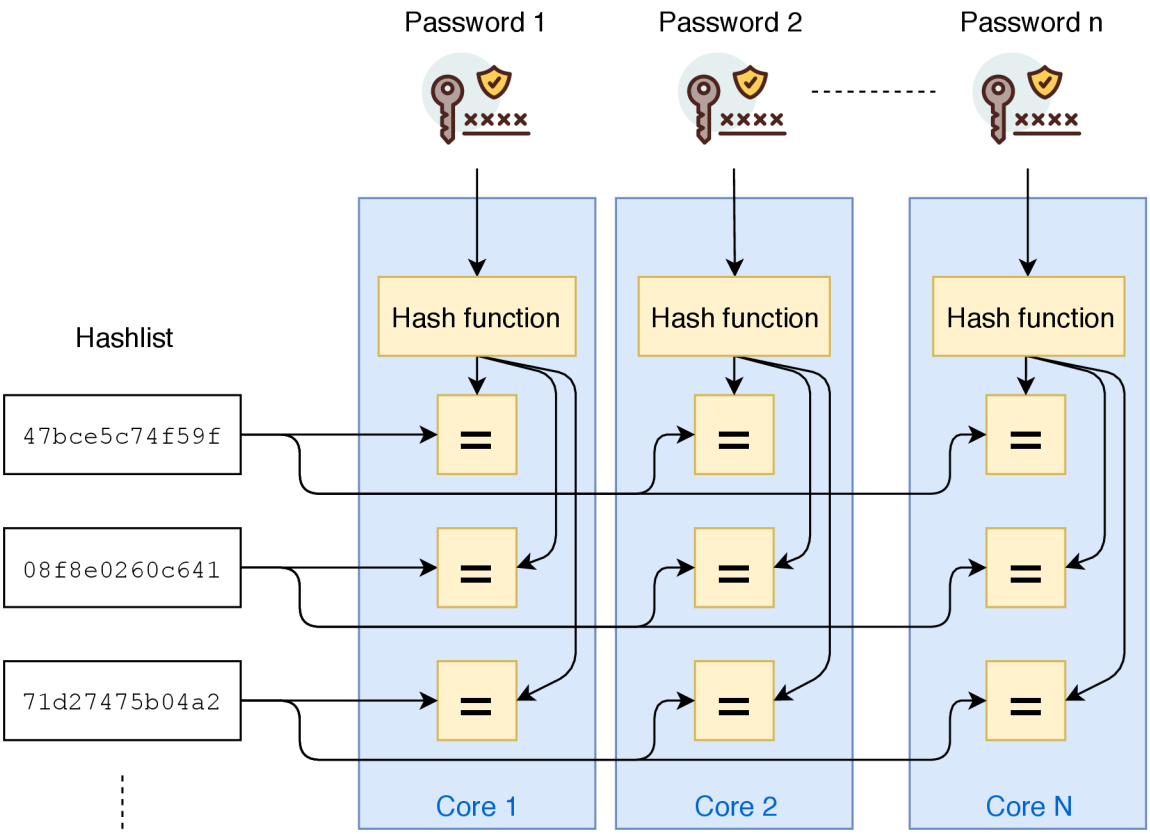


Figure 3.2: Parallel cracking of a password hashlist

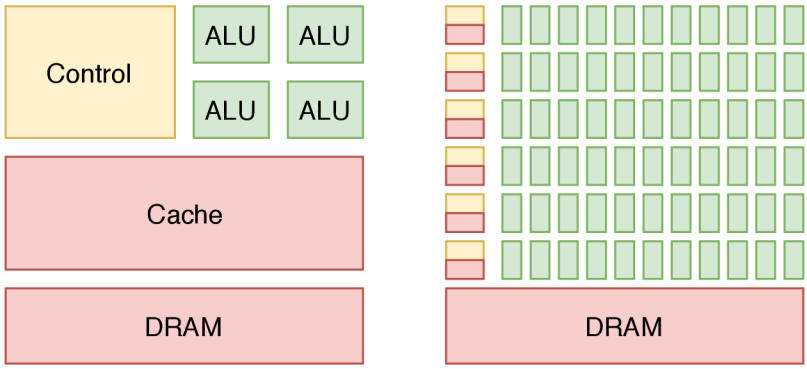
3.1.2 Utilization of GPGPU

Using parallel cracking, we can theoretically verify one password per each core at a time. Neglecting the impact of memory and caching, from Equation 3.2, we can state that the two main factors influencing the theoretical performance are the number of cores and processor clock frequency. Such observations may help forensic investigators choose a proper hardware solution for password cracking tasks.

Central processing units (CPU), designed to carry out instructions of an operating system and applications, usually contain few very fast cores. Today's desktop CPUs usually have 2 to 18 cores, while for server and high-end workstations, the number may be higher. Manycore processors like Intel® Xeon Phi™ have up to 72[1] cores. To boost up parallelization, some CPUs employ a multithreading technology like Intel Hyper-Threading. By duplicating registers that store the processor's architectural state, a single core may act like two virtual (or logical) ones, allowing to run two threads simultaneously [119]. In the past few years, the operating frequency of CPUs flew between 2 to 5 GHz and did not increase dramatically due to energy and thermal constraints [107].

Graphics processing units (GPU), on the other hand, were designed to render graphics, which requires to perform operations on large vectors and matrixes of values. Thus, GPUs have a lower operating frequency but contain thousands of cores. For example, the base

[1] https://ark.intel.com/content/www/us/en/ark/products/series/123588/intel-core-x-series-processors.html



(a) CPU (b) GPU

Figure 3.3: The difference between CPU and GPU

clock of popular NVIDIA GTX 1080 Ti[2] is 1,481 Mhz. However, the card is equipped with 3,584 CUDA cores. The architecture of a GPU allows performing parallel operations on large sets of data. Thus, despite their generally lower clock rates, GPUs may provide incomparably higher performance for specific tasks. Since a GPU is programmable, its use is not limited to graphics only. NVIDIA CUDA and OpenCL frameworks allow *General-purpose computing on graphics processing units* (GPGPU) [141, 134]. Employing GPGPU helps accelerate the computation of complex problems in various areas from linear algebra [115], through machine learning [190], to cryptographic algorithms [135, 14, 17].

Figure 3.3 shows the difference in the architecture of a CPU and a GPU. A typical CPU core, shown in Figure 3.3(a), communicates with DRAM through one or more cache layers, contains a controller, and a set of arithmetic logic units. In the case of a multicore CPU, each core has its own controller and can work independently. A GPU, depicted in Figure 3.3(b), is a programmable parallel multiprocessor, where groups of cores share the same controller. Each group can solve only a single task at the moment, creating a Single instruction, multiple data (SIMD) architecture. The group is called a CUDA Core on NVIDIA GPUs, or SIMD unit in AMD/OpenCL terminology [141, 134].

For password cracking, the number of cores richly compensates the slightly lower operating frequency of a GPU. Using the principles described in Section 3.1.1, we can benefit from the architecture of a GPU to verify masses of passwords in parallel.

In my preliminary research between 2014 and 2016, I evaluated CPU and GPU password cracking performance using different software [83]. The experimental machine contained Intel(R) Core i7 CPU 920 @ 2.67Ghz processor, 16 GB of DDR3 RAM, and two AMD Tri-X R9 290x GPU cards. The experiments show cracking of a WinZIP archive encrypted by 256-bit AES cipher [45], a MS Word 97/2000 document [127], a PDF document under security revision 4 [8], and a PDF document using security revision 5 [7]. I used the following tools: Wrathion[3], oclHashcat[4], John the Ripper 1.8.0 - jumbo 1 (John), Advanced Office Password Recovery 6.10 (AOPR), Elcomsoft Advanced PDF Password Recovery 5.06 (APPR), and Elcomsoft Advanced Archive Password Recovery 4.54 (AAPR). As oclHashcat does not have a non-OpenCL implementation of the cracking algorithms, I did not test the CPU

[2] https://www.geforce.com/hardware/desktop-gpus/geforce-gtx-1080-ti/specifications
[3] https://wrathion.fit.vutbr.cz/
[4] https://hashcat.net/wiki/doku.php?id=oclhashcat

cracking. Since the used version of AAPR has only CUDA-based acceleration, for this case, I used NVIDIA GeForce GTX 660Ti instead.

The experimental results are displayed in Table 3.1. As we can see, in every case, the GPU provided higher performance than the CPU. The difference is most noticeable with the ZIP format. For Wrathion, a single-GPU cracking was 30.28 times faster than using the CPU method. With the dual-GPU deployment, the cracking was 60.56 times faster than with the CPU. For John the Ripper, the GPU cracking was 39.38 times faster in comparison with the CPU. We can also notice the potential for scalability, especially with oclHashcat. After adding a second GPU, the performance almost doubled.

Tool	Processor	ZIP AES-256	DOC 97/2000	PDF Rev 4	PDF Rev 5
Wrathion	CPU	4,329	2,985,065	142,787	7,424,962
Wrathion	1x GPU	131,082	18,547,791	2,621,864	82,735,294
Wrathion	2x GPU	262,190	35,262,422	4,522,481	136,913,101
oclHashcat	GPU	n/a	7,766,401	2,596,223	144,074,187
oclHashcat	2x GPU	n/a	15,377,279	5,121,015	286,340,421
John the Ripper	CPU	376	114,740	n/a	n/a
John the Ripper	1x GPU	14,805	861,320	n/a	n/a
Elcomsoft	CPU	8,102	19,452	32,951	28,861,149
Elcomsoft	1x GPU	n/a	25,543	69,419	n/a

Table 3.1: Cracking performance in passwords per second and GPU acceleration using different tools on ZIP, DOC, and PDF [83]

In Section 3.1, there was an example of a WinZIP archive with a 7-character alphanumeric password that required 14 years to be cracked using brute-force. Using two GPUs and Wrathion tool, the same password can be cracked within 185 days, assuming the values in Table 3.1. The original measurement was performed in 2015. Through time, both hardware and cracking tools got improved. For instance, oclHashcat and cudaHashcat applications were merged into a single Hashcat tool, now containing the missing WinZIP support as well. Companies like Terrahash[5] LLC started to offer multi-GPU cracking solutions. The Brutalis™ appliance with 8x NVIDIA GTX 1080 Ti and hashcat 3.5.0 can crack over 13 millions of WinZIP passwords per second [96]. The same 7-character password can thus be found in 3 days.

3.1.3 The Limits of a Single Machine

While GPGPU shows the potential to accelerate password cracking massively, it resolves only a single part of the problem. No matter how powerful processors we employ, using a single machine always limits the achievable computational power. For a multi-GPU password cracking machine, we need to consider multiple factors, including:

- **The PCI-Express bus** - Every motherboard supports a limited maximum number of connected PCI-e devices. High-end workstations and gaming boards may offer four or more x16 slots. The phenomenon of cryptocurrency mining lead manufacturers like ASUS to create many-slot motherboards, e.g. H370 MINING MASTER[6] containing 20 PCIe slots. Through riser cards, we can connect GPUs to PCI-e x1 slots in

[5] https://terahash.com/
[6] https://www.asus.com/us/Motherboards/H370-MINING-MASTER/

exchange for lower bandwidth. Such solutions are cheap and feasible for cases where throughput is not critical. For HPC computing and professional GPGPU applications, manufacturers provide enterprise GPU servers. For example, Supermicro offers[7] GPU Systems with up to 20 GPUs per machine. Nevertheless, PCI-e slots cannot be extended infinitely. There is a maximum number of PCI-e lanes supported by the CPU. Even with additional PCI-e bridges, the (UEFI) BIOS has a 64 kB IOSPACE. Therefore, we can theoretically use the maximum od 16 bridges [77].

- **Size** - High-end GPUs are relatively big in dimensions, mainly due to the coolers. Building a multi-GPU cracking rig requires proper casing. Enterprise GPU servers from SuperMicro or Dell have cases and backplanes designed precisely for the cards to fit. An exception from size is processors installed directly to the board sockets like notebook graphics Mobile PCI Express Module (MXM) or NVIDIA HPC accelerators with the SXM form factor. They, however, require external cooling.

- **Heat** - With a higher workload, the GPUs produce a high amount of heat. Proper cooling is crucial, especially if the cards are close to each other. Enterprise server solutions have unified systems that allow using even GPUs with passive coolers. Water cooling can also be used as a quiet alternative.

- **Power supply unit (PSU)** - A multi-GPU system needs enough power to supply all cards. Servers from SuperMicro, HP, ASUS, or Dell employ modular architectures with up to 3,000 W per unit. Desktop-based solutions are more limited. First, the most powerful commonly-manufactured PSUs offer 1,600 W (Corsair, EVGA). Exceptions like 2,000 W (Super Flower) or 2,500 W (Spire Corp) are very rare. Desktop motherboards support only a single PSU, which is not enough for machines with six or more high-end GPUs. Enthusiasts thus often use third-party workarounds like Add2Psu[8] enabling to stack up to 4 PSUs together.

- **Price** - It is always to consider if investing in a multi-GPU single-machine solution is more advantageous than using multiple nodes. Moreover, a distributed solution may utilize existing computers.

For the sake of completeness, it is necessary to mention that some manufacturers specialize in designing hardware solutions dedicated to GPGPU password cracking. Terahash[9] offers 5 to 10-GPU apppliances from $15,949 to $31,699. All are two-CPU machines based on SuperMicro server boards with full PCIe-x16 support. Decryptum[10], powered by Passware, offers cracking rig withs with up to 12 water-cooled NVIDIA RTX 2080 Ti for $35,600. In constrast to Terahash, Decryptum machines are based on desktop motherboards, currently the H370 MINING MASTER, and thus the GPU connection is limited to PCI-e x1.

Based on the observations mentioned above, it is clear that there is always a limit to the performance that a single machine can provide. For reaching values beyond that limit, the only feasible way is to distribute the task between multiple physical nodes.

[7] https://www.supermicro.com/en/products/gpu
[8] http://www.add2psu.com/
[9] https://terahash.com/#appliances
[10] https://www.decryptum.com/

3.2 Related Work

Distributed password cracking is mostly the phenomenon of the last two decades. Sadly, there are only a few "bigger" currently-developed projects. Most related solutions are either abandoned, proof-of-concept academic research tools, or commercial applications. The first group contains innovative but mostly single-purpose programs that demonstrate concrete principles on a limited set of use-cases. The enterprise applications, on the other hand, are "black box" solutions. We may read the specification or buy and try them, but that's all. The detailed specification, their architecture, and employed algorithms are proprietary company know-how that is purposedly hidden from the public.

3.2.1 Early Work

The first known password cracking software that supported distributed processing was version 4.0a of the legendary Crack program, released by Alec Muffet in 1991 [205]. Crack was written for Unix systems, using the combination of Perl, C, and Bourne Shell. With the `-network` parameter, the tool allowed employing a network of heterogeneous workstations in a single cracking task. A distributed cracking session had a single master machine and multiple hosts, communicating via Remote Shell (RSH), Remote Copy (RCP), and optionally Network File System (NFS) [140]. In the `network.conf` configuration file, the administrator configured what host machines to utilize. For each computer, the configuration contained its hostname, the relative processing power, and a few other parameters. The workload was divided accordingly to the performance of nodes. The latest release of the Crack tool was version 5.0a from 2000 [133].

In 2001, Steiggner and Wilke showed a distributed password cracking use case to demonstrate the CoSMoS performance monitoring tool. The demonstration showed a distributed dictionary attack on UNIX password hashes. The proof-of-concept tool followed a master-worker scheme. The master process called, *cpw* (crack passwords), spawned a set of worker processes *cow* (check one word). At the start, the master received a password dictionary. During the cracking session, it supplied the workers with passwords, one at a time. Each worker checked the received passwords against the accounts in a given UNIX password file [189].

3.2.2 Cracking in HPC Clusters

Much of the related work is based on the famous John the Ripper (JtR) tool, described in Section 2.4.1. Up to this day, John's wiki enlists 15 different approaches on parallel and distributed processing with the tool. Some of them were later abandoned [12]. The first published academic work on the case was performed by Lim, who modified the sources by adding MPI support for the *Incremental brute-force* attack (see Section 3.8.3) mode [112]. The solution used a master processor and a fixed number of slave processors. The master processor divided the *keyspace* (the number of possible candidate passwords [180]) into a pre-defined number of chunks, while each slave processor received an equal chunk to solve. The principle of keyspace division was, with various alterations, adopted to many subsequent solutions. Due to the equality of chunks Lim's original technique is only feasible for a stable homogenous cluster environment [112].

Pippin et al. proposed a technique for the parallel *dictionary* attack on multiple hashes [162]. Instead of dividing keyspace, they assigned different hashes to each node while all nodes used the same password dictionary. Nevertheless, I consider the approach efficient for

large hashlists and simple cryptographic algorithms only. In my previous work [83], I found the biggest influence on the cracking time has the calculation of the hash from candidate passwords. The rest is a simple byte array comparison. Thus, if we crack multiple hashes of the same type, we can calculate the hash from each password only once and compare the result with all the hashes we are trying to crack. There is no need to recompute the hash of a single password twice, especially with complex algorithms like bcrypt [167] or SHA-3 [139].

Bengtsson showed the practical use of MPI-based brute-force and dictionary attacks using the Beowulf high-performance computing (HPC) cluster for cracking MD5-based Unix shadow files. Similar to Steiggner and Wilke's approach [189], both attacks were based on simple password-by-password keyspace division and demonstrated using a proof-of-concept application called *brutest*. The cracking network consisted of a *root* node responsible for creating and distributing work, and a set of *slave* nodes used for the actual cracking. Each task was defined by an *assignment vector* made of three parts: a *string base*, a *UID*, and a *passwd*. The *string base* told the node where it should start. For dictionary attacks, it contained the next word or phrase. For brute-force attacks, it was the initial sequence of characters used for generating passwords. The *UID* represented the username of the Unix user, while the *passwd* contained the password hash and the cryptographic salt used [26].

Apostal et al. introduced another enhancement to HPC-based password cracking. The *Divided dictionary algorithm* evenly split dictionary words between MPI nodes equipped with GPUs. Using CUDA, GPUs on each node locally calculated the hashes and compared them with the ones that should be cracked [17].

Marks et al. designed a hybrid CPU/GPU cluster of devices from different vendors (Intel, AMD, NVIDIA) [118]. The design included both hardware and software solutions. The distributed network consisted of *Management/storage nodes* that control the calculation and handle user inputs, and *Computation nodes* responsible for the cryptographic work. For interconnection, Marks used three different lines: 10 Gb/s Ethernet for data transfer, 1 Gb/s Ethernet, and InfiniBand for controlling the computation process. Marks also proposed a software framework called *Hybrid GPU/CPU Cluster* (HGPC) utilizing a master-slave communication model using an XML-based protocol over a TCP/IP network. A proof-of-concept implementation was able to crack MD5, SHA-1, and four versions of SHA-2 hashes. Experimental results of cracking on up to 24 nodes showed great power and scalability. However, I suppose using an optimized tool like hashcat of JtR could increase the performance even more. While cracking MD5 hashes on NVIDIA Tesla M2050, Marks achieved the speed of around 800 Mh/s, while hashcat users report[11] cracking over 1200 Mh/s using the same GPU.

3.2.3 Non-HPC Solutions

Previous solutions work well for a "classic HPC" system of a homogenous cluster with a static set of nodes. Since my use-case also covers grid computing with existing computers and employing heterogeneous networks of a possibly changing set of nodes, I would like to present existing related non-HPC solutions.

Using a simple text-based protocol, loosely modeled on HTTP, Zonenberg created a distributed solution for cracking MD5 hashes using a brute-force attack [221]. The architecture consisted of a master server and a set of compute nodes, which were either CPU-based or used GPU acceleration based on CUDA.

[11] https://hashcat.net/forum/thread-2084.html

Crumpacker [47] came with the idea of using the BOINC framework [15] to distribute work and implemented a proof-of-concept tool for distributed cracking of operating system hashes with JtR. Since Crumpacker uses BOINC, there are similarities to my solution, the Fitcrack. For instance, Crumpacker also had to create a custom design of the BOINC daemons, namely the Work generator, the Assimilator, and the Validator. Similarly to my proposal, he decided to standardize the size of the workunit (a BOINC term for a chunk) by counting out the number of passwords that one computer could check in the desired amount of time. Nevertheless, both systems are different in concept. While Fitcrack uses the idea of independent jobs, each having one or more input hashes, Crumpacker's server backend employs the JtR database as unified hash storage. Using the command line, users can load new password hashes to the database. Whenever the Work generator creates workunits, it loads the hashes from this database. The most significant difference is the use case. Fitcrack serves as the general-purpose cracking system and allows for cracking over 300 hashcat-supported hash formats using eight different attack modes. It can even automatically extract hashes from headers of encrypted documents and archives. Crumpacker's tool, on the other hand, is purely an OS password cracker. It supports three JtR's attack modes: single cracking mode, wordlist mode, and incremental attack mode. There are no additional features like integrated hash scrapers, user account management, or even a graphical user interface. The tool supports six password storage schemes: traditional DES-based secheme [202] and its extended BSDi version, MD5-based scheme [172], Blowfish-based bcrypt [167], Andrew File system (AFS), and LanMan (LM) [125].

Crumpacker also reports he could not properly distribute the Incremental crack mode since JtR did not track the starting and ending position of the generated password segments. He resolved the issue by modifying the JtR database, but at the cost of efficiency. Crumpacker later introduced the *batch concept* which divides passwords into groups called batches and tracks them during the entire cracking process, possibly using different hash types, and attack modes [47]. While Crumpacker's proof-of-concept tool offers only basic features, it shows BOINC's suitability for password cracking, supporting my choice for the framework.

Despite hashcat's performance, JtR still offers some advantages over hashcat. For instance, it supports some formats which hashcat does not, e.g., encrypted RAR3 archives with an unprotected header. Such formats use the decryption-based or checksum-based password verification procedures described in Section 2.3. They require a large piece of work performed on the CPU of the host machine. Still, hashcat's cracking password verification is mostly OpenCL-based. As described in Section 3.5, I studied the possibilities of JtR integration to Fitcrack and encountered similar problems with distributing incremental mode as Crumpacker reported [47]. Therefore, I postponed these efforts to future work and decided to focus on hashcat in this work.

Kasabov et al. performed research on password cracking methods in a distributed environment. The resulting technical report compares existing frameworks and describes different architectures and technologies for workload distribution [103]. Kasabov considers MPI combined with OpenCL as the best practical approach for setting up a password cracking GPU cluster, underlying the possibility to use a combination of MPI and OpenMP[12] to gain fine-grained parallelism [215]. However, the research is merely theoretical and provides no proof-of-concept tool or experimental results to support the conclusions. Kasabov mentions Zonenberg's Distributed Hash Cracker [221], but Crumpacker's BOINC-based solution [47]

[12] https://www.openmp.org/

is not discussed. Still, the report includes a brief study of BOINC, emphasizing its advantages in automation, including integrity checks, workunit replication, checkpointing, and other features that my Fitcrack system [87, 84, 81] and Crumpacker's JtR-based solution take advantage of [47]. I agree with Kasabov that BOINC is not an "out-of-the-box" solution for password cracking. However, I do not consider the actual creation[13] of a password cracking project to be an as difficult task as Kasabov describes. Moreover, the statement "BOINC API lacks functions for managing projects" [103] is, at the time of writing this thesis, not entirely true. Every BOINC project contains a project management website, allowing the administrator to observe and control the project's tasks, users, and other options. The rest can be added by writing a custom interface, e.g., the Fitcrack WebAdmin described in Section 3.7.6.

Veerman et al. aimed to create a scalable, modular, and extensible solution for password cracking using existing cracking tools [154]. Their preliminary research compares existing password cracking tools and discusses the possible use of BOINC or MPI for task distribution. Veerman does not consider BOINC to be an optimal choice due to "large deployment overhead and complexity", referring to Kasabov's research [103]. As Veerman states [154], the use of MPI requires the cracking tool to either support MPI or be modifiable for adding the MPI support. Since Veerman wants the solution to support both closed-source cracking software, such modification may not always be possible. The work describes the architecture of the proposed system consisting of three parts: the *Node Controller* which handles user requests, stores cracking-related data, and schedules work; the *Worker Node* responsible for cracking; and the *Website* serving as a user interface. Generally, the design is similar to the Fitcrack system's architecture, described in Section 3.7. The assignments entered by the user are divided into *subjobs* analogous to *workunits* in Fitcrack. The software output represents a proof-of-concept system based on PHP, MySQL, Apache, and SQLite. From all discussed cracking tools, the proposed PHP-based system could only use JtR with the default cracking configuration (first dictionary, then brute-force attack), and the cracking is limited to MD5 hashes only [154]. I consider the proposed architecture well-designed. Nevertheless, the software is no more than a proof-of-concept tool due to the limited functionality. Sadly, the project now seems to be abandoned, and the repository with the software is empty.

Kim et al. proposed a protocol for distributed password cracking, based on the distribution of password indexes, i.e., starting points and keyspaces of each workunit [105]. The paper clarifies the principle used in various existing tools, including my Fitcrack system [87]. Kim's specification is, however, very shallow and limited to a brute-force attack on a single hash.

3.2.4 Commercial Distributed Password Crackers

While the above-shown work is mostly from academic research and hacker enthusiasts, I would like to cover existing enterprise distributed cracking solutions from the commercial sphere as well.

In 2000, AccessData Corporation released the Distributed Network Attack (DNA) for CPU-based exhaustive key search on encrypted MS Office 97 and Office 2000 documents. The cracking network contained a DNA Supervisor and DNA Worker nodes [1]. The 2010's version supported Office XP, PDF, PKZIP, WinZIP, and RAR up to version 2.9 of WinRAR. The 2020's version 8.2.1 supports over 70 different types of password-protected media.

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

The Worker application runs on Windows, Linux, Mac OS X, and even PS3. The GPU acceleration, however, is only for Microsoft Office and WinZIP formats. Unlike hashcat and JtR, the DNA is for cracking concrete media formats and application passwords. While it supports MD5 and SHA-based crypt, the solution is not designed for cracking raw hashes and ciphers [5].

On the 7th of February 2006, Elcomsoft announced a new product: Elcomsoft Distributed Password Recovery (EDPR). The initial version supported cracking passwords for decryption of MS Office 97 to 2003 documents. A cracking network has three main actors: a server, a console, and agents. Each of them may run on a different computer. Users create cracking tasks using the console. The console sends each task to the server that distributes it to the machines with installed agents. The agents work on the assignment and periodically report their status to the server [59]. In the following years, Elcomsoft extended the EDPR with the features of their other password recovery tools. Like hashcat and JtR, and unlike AccessData or Passware, EDPR supports cracking raw and salted MD5 and SHA-2 hashes. Nevertheless, both the range of supported formats and the performance [59] is noticeably lower than hashcat's [96]. And there is no native support for complex algorithms like bcrypt [167], scrypt [157], or SHA-3 [139].

In February 2010, Passware released a beta version of the Distributed Password Recovery extension. The new feature, available for the Enterprise and Forensic editions of the Passware Kit, allowed to increase the cracking performance by utilizing multiple machines. On each additional node, the administrator installs an application called Passware Kit Agent. Each agent is a cracking client connected to a workstation with the Passware kit that acts as a server for assigning work [151]. The Passware Agent application runs on both Windows and Linux systems, 64-bit only. Passware Kit Forensic 2020 v3 supports over 280 different formats. The company shows the performance of cracking five selected formats with NVIDIA RTX 2080 Ti [153]. The measured values are comparable to hashcat on the same GPU [145]. From all supported formats, only 80 provide GPU acceleration. For instance, PDF does not support GPU at all. Unlike hashcat and JtR, Passware Kit also does not support cracking raw hashes and ciphertexts [152].

3.2.5 Hashcat-based Solutions

Since I decided to use hashcat as a cracking engine because of its performance and variety of supported algorithms (see Section 3.5), it is necessary to mention existing work, despite being out of the academic sphere.

Hashstack[14] is an enterprise solution from Sagitta HPC, a subsidiary of Terahash LLC founded by J. Gosney, a core member of Hashcat development team. The authors refer to the solution as the "hashcat on catnip" and claim it provides extreme scalability. It should support 375+ highly-optimized hash formats, six attack modes, multi-user support with granular access control lists, and API to automate workflows. Nevertheless, the solution is closed-source except for a few plugins available[15] on GitHub and distributed exclusively with Sagitta's GPU cracking appliances.

McAtee et al. presented the Cracklord[16], a system for hardware resource management, which supports creating job queues and contains a simple hashcat plugin. The plugin allows to remotely run a dictionary or a brute-force attack with a limited set of options

[14] https://web.archive.org/web/20201108092329if_/https://terahash.com/#hashstack
[15] https://github.com/stricture
[16] http://jmmcatee.github.io/cracklord/

[120]. However, the project seems to be updated very rarely, and the last supported version is hashcat 3.

In 2014, a Github user cURLy bOi from Prague, Czech republic created Hashtopus[17], an open-source distributed wrapper around oclHashcat, a predecessor of the current hashcat tool. In 2015, Jakub Samek, a student from Czech Technical University in Prague, used Hashtopolis system with cudaHashcat to create a virtual GPU cluster for his bachelor's thesis [178]. Sadly, the Hashtopus project ended in 2017, when the author announced leaving the hash cracking scene. Since most of the code was still usable, Sein Coray created a fork called Hashtopussy, later rebranded to Hashtopolis in 2018.

Hashtopolis[18] uses a network with a *server*, and one or more *agents* – machines used as cracking nodes. The server is a web application written purely in PHP. It provides a user-friendly administration interface and agent connection point. The user interface allows to create and manage cracking tasks, hashlist, and others. The tool uses a MySQL or MariaDB database as data storage. Computing nodes contain a Hashtopolis Agent, a client application in C#, or Python implementation. The communication uses a custom protocol based on JSON over HTTP(S).

Many of the features of Fitcrack and Hashtopolis are similar. Fitcrack and Hashtopolis offer many similar features. However, the philosophy of the system and the concept of cracking tasks are entirely different. Hashtopolis runs solely on an HTTP server and does not employ any other actively running server daemons. While Fitcrack's WebAdmin has a separate frontend and backend connected via a REST API, Hashtopolis is a monolithic application. Without modification, the only control of Hashtopolis is via the graphical user interface. In contrast, Fitcrack's backend API allows controlling the system even from an external application. Fitcrack distinguishes between different attack modes and offers a unique distribution strategy (see Section 3.8) for each one. Hashtopolis treats all attacks the same, and the configuration is up to the user who needs to specify hashcat's command-line arguments manually. The control of cracking sessions is thus more low-level, and the system does not provide that high level of abstraction as Fitcrack. For each task, the user selects a hashlist, one or more files (e.g., password dictionaries) to be transferred to the client, and an *attack command* in the form of hashcat program options. In contrast to Fitcrack, the user needs to define most attack-based hashcat options by hand. Like Fitcrack, Hashtopolis handles benchmarking, keyspace distribution into chunks, and automated download of hashcat binaries and other files necessary for cracking. Being the only well-known maintained open-source solution for distributed computing with the current version of hashcat, I consider Hashtopolis a state-of-the-art tool in my research area. Therefore, my experiments in this thesis also compare the proposed Fitcrack system with Hashtopolis under different attack options.

[17] https://github.com/curlyboi/hashtopus
[18] https://github.com/s3inlc/hashtopolis

3.3 Requirements for a Distributed Cracking Solution

Following the research goals stated in Section 1.2, I decided to propose a distributed password cracking system with the focus on the following aspects:

- **Performance** - The overall performance of the system is the most crucial factor. The tool should utilize GPGPU technologies like OpenCL [134] or CUDA [141] to accelerate cryptographic algorithms and acquire as high cracking performance as possible.
- **Efficiency** - In an ideal state, all available processors are utilized all the time during the entire task. Naturally, such a goal is impossible to be achieved in real situations. In distributed computing, there is always an overhead that computing nodes require for communication, i.e., the interchange of commands and data, synchronization, etc. Operations like benchmarking or performing database transactions add another overhead as well. The efficiency is defined as the percentage of the time that processors actually spend processing rather than communicating or idling [146]. In other words, describes how well the processors are utilized. The lower the overhead, the higher the efficiency.
- **Scalability** - For achieving higher performance, the user may increase the number of computing nodes. The scalability describes how the performance develops if new nodes are added [13]. In an ideal state, the scalability of a distributed system is linear. Assume that all nodes have the same performance. If the scalability is linear, then doubling the number of nodes doubles the performance as well. The closer to the linear scalability, the better.
- **Adaptability** - For the purpose of my research, I assume a potentially changing computing environment. Especially in computer grids, the nodes do not necessarily have to be dedicated to the password cracking task only. Another applications may occupy the processor and temporarily decrease the performance. Parts of the network may even go offline as well as new nodes may appear during the computation process. The adaptability describes how well the system can withstand such changes and adjust the scheduling strategy properly.
- **Robustness** - Solving a task may take hours and days. If the computation of a partial result fails, the system should recover from the failure and arrange the continuance of the process.
- **Security** - While data exchanged between computational nodes may contain confidential information, the transfers have to be secure, and in some cases, only trusted nodes may be allowed to join the network.
- **Compatibility with commonly-available systems** - Computing nodes should not necessarily require specialized hardware. They do not have to be a part of a dedicated HPC cluster. The system should run on personal computers with consumer-grade GPUs and commonly used operating systems like Microsoft Windows or Linux.
- **Open-source code** - For research purposes and further development, all parts of the system should be publicly available under open-source licenses. Meeting this condition allows smooth reproduction of experimental results and enables any enthusiast to contribute to the system by creating additional modules and improvements.

3.4 Frameworks for Distributed Computing

With the limits of the single-machine approach, described in Section 3.1.3, distributed computing is often the only way to achieve higher raw cracking performance. The basic idea is to make multiple physical or virtual machines cooperate on a common problem by putting their computational power together. Different models describe the conceptual design of distributed systems. We can categorize them by the network topology (centralized/decentralized, star, ring, hierarchical, etc.), by the type of communication (synchronous vs. asynchronous), and other aspects [108].

Following the requirements from Section 3.3, I decide to design a distributed system that is not limited to HPC clusters but can use existing computer networks, including the Internet. Cracking tasks may run not only in local area networks (LAN) but also in larger computer grids with nodes in different locations. The system may utilize both dedicated GPU servers and personal computers. For instance, a company may use the computers for office work during the daytime, and at night, they automatically switch to password cracking mode. While it is possible to create a custom protocol from scratch, existing solutions cover various underlying operations and simplify creating new applications. In the following sections, I analyze different frameworks for distributed computing and their usability for password cracking tasks.

The selection of frameworks is inspired by the previous work of Kasabov et al., who compared BOINC with MPI and CLara [103]. In addition to these three frameworks, I also discuss VirtualCL and Apache Hadoop. Naturally, there are other existing solutions that are not included in the comparison for various reasons. Concretely, Apache also offers Spark[19] and Flink[20] frameworks. Both are conceptually close to Hadoop but use different computing and data flow models. Spark is part of the Hadoop ecosystem and is based on micro-batch processing. Flink uses a continuous flow, operator-based streaming model. Like Hadoop, both systems aim at Big data processing, which is far different from password cracking. Hadoop supports porting to many languages, including C and C++, that allow for creating compiled and highly-optimized applications. Spark and Flink support only Java, Scala, Python, and R. Microsoft Orleans[21] is another robust and scalable framework for distributed computing. Nevertheless, it is not designed for tasks whose processing lasts long as there is a limit in seconds for each function call. Moreover, it only supports programming in .NET languages. Another framework for creating concurrent and distributed systems is Akka[22]. The computing model uses actors and streams. Actors are computing nodes that do work, while streams serve for communication between individual actors. The portability is, however, limited since Akka only supports Java and Scala languages.

3.4.1 MPI

Message Passing Interface (MPI) is a protocol specification and a library providing an efficient way of coarsely dividing work between different processes. Depending on the configuration, an MPI application may run on multiple processor cores or multiple physical machines. The MPI was standardized by the MPI Forum in cooperation with researchers from various organizations, mainly from the United States and Europe, including the major vendors of concurrent computers, universities, and government laboratories [208].

[19] https://spark.apache.org/
[20] https://flink.apache.org/
[21] https://dotnet.github.io/orleans/
[22] https://akka.io/

The computing model uses the concept of communicators. A *communicator* is an object describing a group of processes that can communicate with each other by sending messages. Message passing is performed by calling functions named *MPI routines*. Every call must correspond to a specific communicator. MPI includes routines for both point-to-point as well as for collective communication. The routines exist in both blocking and non-blocking versions so that the programmer can select a proper variant depending on the desired use case [209]. The current MPI-3 standard supports shared, distributed, and hybrid memory models [124].

Kang et al. claimed MPI shows an excellent performance for computational-intensive tasks with a moderate amount of data [100]. Password cracking fits well into this class of problems as it uses relatively little data but may require high computational power. This property may be why some researchers eventually decided for MPI as the framework for their distributed cracking solutions. For instance, Apostal et al. employed the MPI in the CUDA-based password cracker [17]. Bengtsson also used MPI for cracking MD5-hashes of user passwords on the Beowulf HPC cluster [26]. However, it seems the MPI may not be an obstacle even for larger amounts of data. Reyez-Ortiz et al. used MPI on Beowulf cluster for supervised learning processing big datasets. They declared MPI showed great scalability and was ten times more powerful than Apache Hadoop on Spark[23] cluster [170].

There are multiple existing implementations of MPI, both open-source like OpenMPI[24] or MPICH[25], and commercial, e.g., Intel MPI[26]. OpenMPI includes various fault-tolerance techniques: local or distributed checkpoints, network failure detection, etc. In Section 3.3, I defined the proposed cracking system should support adding new computing nodes during runtime. The original design of MPI did not consider such an option. However, version 2.0 introduced the `MPI_Comm_spawn()` function that starts a new process, even on a new node. Unfortunately, new nodes are not detected automatically, so the programmer needs to create an extra process that detects new connections.

Although MPI implementations support distributed computing over the Internet, MPI-over-Internet is still a challenge today due to its volume and complexity [74]. Programming of interconnected cluster applications with MPI often requires advanced communication paradigms that increase the programming effort for code writing. Moreover, MPI is a low-level solution and does not natively support encryption or authentication. All such security mechanisms would have to be implemented manually by the application programmer.

3.4.2 Apache Hadoop

Apache Hadoop[27] is a distributed computing framework, primarily designed for processing big data in a cluster-based environment. Individual problems are described with simple programming models, mainly using the *MapReduce* model. The paradigm works as follows. The input data is processed by the `Map()` function, which provides filtering and sorting. Concretely, it transforms couples $(K_1, value_1)$ to $(K_2, value_2)$ where K_1 and K_2 are keys. The `Reduce()` function then merges all values from the result which have the same K_2. Finally, the system collects all results of the `Reduce()` operation, optionally sorts them by K_2, and produces the final result.

[23] https://spark.apache.org/
[24] https://www.open-mpi.org/
[25] https://www.mpich.org/
[26] https://software.intel.com/content/www/us/en/develop/tools/oneapi/components/mpi-library.html
[27] http://hadoop.apache.org/

The framework is distributed as open-source software under Apache License 2.0 with the code mostly written in Java programming language. Hadoop-based projects should scale well, even on thousands of machines, while each node can use its local computation storage. The framework uses its own Hadoop Distributed File System (HDFS), supporting encryption and data replication. It also offers failure-recovery, useful analysis tools, and adding new nodes when the process is already running[28].

Apache Hadoop is also used in the area of digital forensics. Using Hadoop, Chen et al. designed and implemented the Collaborative Network Security Management System (CNSMS), a network forensics solution for collecting and analyzing high amounts of network data in parallel. The system can detect, find, and trace evidence of malicious activity like phishing attacks, spam scattering, and network worms in real-time. Chen et al. claim CNSMS can make a network resistant even to a DDoS attack [42]. Cho et al. also recommend Hadoop for forensic purposes and even proposed guidelines for Hadoop-based cloud forensics [43].

On the other hand, Roussev et al. found weaknesses when comparing Apache Hadoop with the MPI MapReduce[29] (MMR) which is an open implementation of MapReduce processing model. They reported MMR having better scalability and performance than Apache Hadoop [177]. Reyez-Ortiz et al. also criticized Hadoop for showing much lower performance than MPI-based solutions. However, they admit that Hadoop is easier-to-use and could be preferred because of the included features. The main discussed advantages contain the distributed file system with integrated data replication management, adding new nodes on-the-fly, and a set of useful tools for data analysis and management [170].

Even though showing noticeably lower performance than MPI, Apache Hadoop is an excellent solution for problems on which we can apply the MapReduce processing. It is easy-to-use, well-designed for working with Big Data, and offers a set of useful integrated tools for data analysis and system management. Moreover, it natively supports encryption, data replication, and is robust thanks to integrated fault-tolerance routines. In contrast to MPI, Hadoop allows adding new nodes at runtime. Although there were approaches[30] on distributed computing of hashing algorithms using Hadoop and MapReduce, I do not the password cracking process an appropriate candidate for the MapReduce model.

3.4.3 VirtualCL

The philosophy of VirtualCL (VCL) is completely different from the previously-discussed frameworks. VCL works as a wrapper over OpenCL. It allows applications to use OpenCL devices located on different computing nodes in the same way as they were connected locally to the computer. Such use does not require any additional modifications to the native OpenCL application [21]. VCL uses a custom protocol over TCP/IP, and the network latency is the main limiting factor. A set of SHOC benchmarking tests showed that the overhead of VCL is nonnegligible, especially with bigger chunks of input data [51]. The latest version 1.25 of VCL was released in 2017, and the project does not seem to be develop anymore.

Nevertheless, I intended to experimentally evaluate its potential benefits to distributed password cracking. Therefore, I decided to test VCL with the hashcat tool, even though the

[28] http://hadoop.apache.org/docs/
[29] http://mapreduce.sandia.gov/
[30] http://en.cnki.com.cn/Article_en/CJFDTOTAL-TXBM201111026.htm

authors state[31] that "VCL is no longer recommended for oclHashcat clustering, and most likely will not work if you attempt to do it anyway". I deployed VCL 1.25 to a computer network of nodes equipped NVIDIA GTX 1050 Ti. However, the NVIDIA drivers did not seem to be supported anymore since the cards were not visible from the host machine. After switching to AMD R9 290X, the remote GPU became accessible from the host machine, but it was impossible to use the device with hashcat version 2 or higher. I consider the only option is to use an old deprecated `oclHashcat-plus-0.15` or choose a different solution.

In addition to the compatibility issues, password cracking with VirtualCL adds an undesirable amount of overhead compared to running a standalone password cracker remotely. Assume a network with a main controlling node that supervises the cracking task and a series of GPU-equipped working nodes. Utilizing a tool like hashcat or JtR only requires the controlling node to specify the attack settings, e.g., in the form of command-line arguments. The working node may then work independently, and the only other necessary communication is at the end of the task when the worker reports the result to the controller. VirtualCL, on the other hand, performs all OpenCL calls over the network [21]. As I detected, practically all existing GPU-supported password crackers require a periodical low-latency host-to-GPU communication to synchronize the work, monitor the device, verify the success of password verification, etc. Even the Wrathion that I proposed in my preliminary research [81] works in iterations, each verifying a vector of candidate passwords. The cycle is controlled by the host machine that actively checks the previous iteration results and defines the instructions for the next one. Hashcat uses a similar concept with two nested cycles, where the *base* loop (see Section 3.6.4) runs on the host's CPU, while the *modifier* loop is implemented within the OpenCL GPU kernels. Performing host-to-GPU communications over the network instead of the PCI-e bus increases the latency dramatically and introduces a significant overhead.

3.4.4 CLara

Similar to VirtualCL, CLara is a framework that allows accessing graphic processors over IP networks. Compared to VCL, the concept of CLara is more general as it uses the "many-to-many" communication paradigm where any computer in the network may access an OpenCL device to any other computer. The central point of the system is a reverse proxy server called *clarad*. Computers connected to the server are providers and consumers. A provider is a computer that offers its OpenCL devices, e.g., one or more GPUs. With the OpenCL platform provider application *clarac*, the machine connects to the proxy server. Finally, a consumer is a computer with an OpenCL application linked to the *libclara* library. Through the proxy server, the library provides access to the connected OpenCL devices [31].

I assume a password cracking system built over CLara could potentially run multiple cracking sessions initiated at different points of the network. Sadly, the project has been abandoned since 2015, and the last version is compatible only with the obsolete OpenCL 1.0 standard.

Nevertheless, even cracking with the old 1.0 standard faces the same issues described in Section 3.4.3 for VirtualCL. All OpenCL-related calls are transferred over the network instead of a fast, low-latency PCI-e bus. For cracking tasks, this adds a significant overhead that is neither wanted nor necessary.

[31] https://hashcat.net/wiki/doku.php?id=vcl_cluster_howto_original

3.4.5 BOINC

Berkeley Open Infrastructure for Network Computing (BOINC) is a platform for distributed computing that natively supports a dynamic number of nodes connected over the Internet [15]. It is an open-source project developed by U.C. Berkeley Space Sciences Laboratory. The primary use-case of BOINC is public-resource computing. Volunteers participate in solving various scientific problems, including the analysis of RNA molecules, simulation of proteins, weather prediction, and many others. Though the original purpose of BOINC is public-resource sharing, it offers excellent options for grid computing.

The architecture resembles a client-server model where the network consists of a server and one or more hosts. The *server* is responsible for the management, planning, and scheduling of tasks. *Hosts* represent computing nodes that perform the actual work. Each host contributes to the process by providing its computational power.

A problem or a set of problems is described as a *project*. Within the project, the server creates chunks of work called workunits. A *workunit* is the smallest piece of work that can be processed by a host. Once a project is created, hosts can connect and participate in the computing process by sharing their power for solving the project's workunits. The project server is responsible for creating and scheduling tasks, keeping track of clients, maintaining data storage, etc.

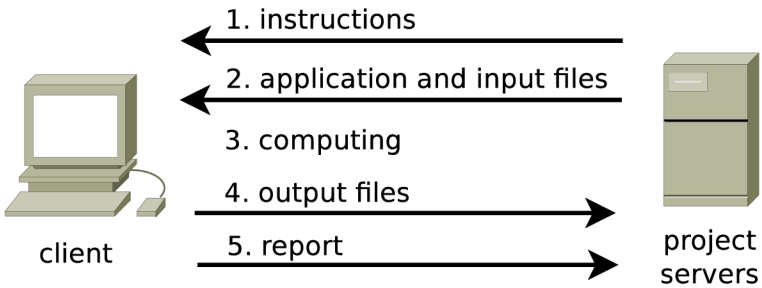


Figure 3.4: Solving a single workunit using BOINC

Figure 3.4 illustrates the communication between the project server and a client that is required for solving a single workunit. At first, the client receives instructions describing the task. Task assignment is client-specific, i.e., the server takes client architecture, operating system (OS), and hardware specification into account. BOINC allows the scheduler to create a task tailored precisely to the client's abilities.

For solving the task, a client may need one or more applications. BOINC supports automatic distribution and update of application binaries with respect to the client's architecture and OS. It also distributes input data of workunits. Once the client receives all the necessary data, it starts computing, which may take minutes, hours, or days. When the client completes its task, the output files and the task report are sent back to the server. After reporting the result, the client may then ask for a new workunit.

With BOINC, it is possible to run literally any application on the host machine as long as it is supported by the architecture and OS. The framework provides two options for running client-side programs: i) using BOINC Wrapper, ii) using BOINC API. The wrapper[32] is an application supplied by BOINC that allows executing any program on the host machine. Using BOINC Wrapper is easy, straightforward, and does not require the modification of any code. The communication is possible via redirecting the program's input

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

and output or reading its return value once the program ends. BOINC API, on the other hand, is an application interface proposed by Anderson et al. that allows programmers to create "native" BOINC applications using the runtime BOINC Client Library (`libboinc`). The communication between the BOINC core client and the app uses shared memory and message passing. Compared with the wrapper, the API provides many advanced features like progress reporting, checkpointing, critical section handling, atomic operations, pausing, or using time handlers [16].

Unlike MPI or VCL, BOINC is designed to distribute tasks over the Internet. It provides the optional use of built-in security mechanisms for untrusted environments: authentication, user account management, digital signatures, and public-key encryption. Clients can dynamically connect to and disconnect from a running project. Users can even specify the percentage of CPU or GPU power assigned to a BOINC task, network upload or download limits, disk, and memory size utilization assigned to the computing. It is also possible to define at which time the computing should start, restrict the computing to concrete days in week, etc.

Crumpacker et al. showed BOINC can be used for password cracking and created a simple proof-of-concept tool for distributing attack with John the Ripper tool [47]. Kasabov et al. emphasized BOINC's reliability and robustness and suggested the possible use for password cracking if a proper server software is developed and a GPU-accelerated client application is used [103]. Apart from the password cracking, DistrRTgen[33] was a volunteer BOINC-based project created for distributed computing of freely-available *rainbow tables* [142]. However, the official website appears to be defunct, and the project is currently marked as *private*.

3.4.6 Summary

Inspired by Kasabov et al. [103], I provide a table-based feature comparison of all the discussed frameworks. Table 3.2 compares the above-described solutions based on criteria that I consider crucial for distributed password cracking to meet the requirements from Section 3.3.

While the MPI shows outstanding performance and scalability, the use in dynamic non-HPC environments is problematic due to many missing features. These include authentication, encryption, and detection of new nodes. Hadoop, on the other hand, provides native support for computing over the Internet, contains failure-recovery routines, and much more. The primary advantages are working with big data and the MapReduce model, which is not the case. VirtualCL and CLara are low-level solutions for sharing OpenCL devices over the network. However, the projects are abandoned and mostly are not compatible with today's hardware and software. Moreover, VCL and CLara introduce high overhead for cracking tasks, as discussed above.

I eventually decided to use BOINC since it fits my needs the most. The framework offers auto-negotiation of hardware and software specifications, downloading and updating client binaries, failure recovery, authentication, and encryption, and much more. The only slight drawback of BOINC was the communication overhead and delays between reassignment of workunits. In the Fitcrack system, I later managed to resolve the issue using the concept of pipeline processing described in Section 3.6.5 that practically eliminated the overhead.

[33] http://boinc.berkeley.edu/wiki/DistrRTgen

-	MPI	Hadoop	VirtualCL	CLara	BOINC
overhead	low	moderate	high	high	moderate
cracking without additional app.			•	•	
non-LAN support	•	•	•	•	•
adding new nodes on-the-fly		•			•
fault-tolerance routines	•	•			•
native support for authentication		•			•
native support for encryption		•			•
maintained and updated	•	•			•

Table 3.2: Comparison of cracking-related properties of the analyzed frameworks

3.5 The Choice for the Cracking Engine

A mechanism for computing the cryptographic algorithms is likely the most critical part of every password cracker. To achieve high performance, hardware acceleration based on GPGPU was a clear choice. Solutions based on OpenCL [134] or NVIDIA CUDA [141] technologies should provide broad support for commodity hardware with consumer graphic cards, as required in Section 3.3. Designing custom GPU kernels from scratch, as I did for the Wrathion tool, is unnecessary since there already are existing tools with well-optimized implementations of the password verification routines. For choosing a proper cracking engine, I focus on the following criteria:

- **Multi-OS support** - The solution should support at least both Windows and Linux
- **Support for integration into external software** - The solution should provide an interface over which the password cracking system can control it. Concretely, to run and manage a cracking session over an API, over a CLI with arguments that specify the attack settings or offer an externally-editable configuration file.
- **Performance** - The solution must provide GPU acceleration.
- **Format support** - The solution must cover a wide-enough range of data encryption and hashing algorithms. Those include cracking raw hashes, OS passwords, or password that secure encrypted archives and documents.
- **Attack modes** - The solution should support not only the dictionary and brute-force attack but should also cover advanced attack with password-mangling rules, word combinations, etc.
- **Maintenance and distribution** - The solution should be maintained and updated by the creator. Licensing terms should allow integration of the tool into a larger password cracking system. Open-source free-of-charge licensing is preferred.

In Section 2.4, I described existing password cracking tools. Table 3.3 compares their features. Those include OS and integration support, GPU acceleration, supported formats and algorithms, attack modes, development, and licensing. The comparison does not include L0phtcrack since it uses John the Ripper (JtR) as the cracking engine. I also omit Ophcrack because it is a single-purpose tool for LM and NTLM hashes only. Note, the number of supported algorithms is based on the list published by the software's creator.

-	JtR	Cain	hashcat	Elcomsoft	AccessData	Passware	RBcrack
Version	1.9.0 j	2.0	6.1.1	EDPR 4.40	PRTK 8.2.1	2021 v1	1.8.0
Windows support	●	●	●	●	●	●	●
Linux support	●	-	●	-	○*	○*	○**
Integration support	●	-	●	-	-	●	●
* – only for agent (worker), ** – without GUI							
Cracking support							
GPU acceleration	○	-	●	○	○	○	○
Algorithms	287	26	323	69	64	289	5
Algorithms (GPU)	88	-	323	44	10	79	4
Raw hashes	●	●	●	●	-	-	●
OS passwords	●	●	●	●	●	●	●
Network protocols	●	●	●	●	-	●	-
Archives	●	-	●	●	●	●	-
Documents	●	-	●	●	●	●	-
Applications	●	●	●	●	●	●	-
Attack modes							
Brute-force	●	●	●	●	●	●	-
Dictionary	●	●	●	●	●	●	-
(Word mangling)	●	○*	●	●	●	●	-
Word combination	-	-	●	●	●	●	-
Hybrid attacks	-	-	●	●	●	●	-
Rainbow table	-	○	-	○	○	○	●
* – only reverse, case mangling, two-digit appending							
Development and distribution							
Maintained	●	-	●	●	●	●	●
Latest release	2019	2014	2020	2020	2018	2021	2020
License	GNU GPL*	Freeware	MIT	commercial	commercial	commercial	BSD
● - full support, ○ - partial support, * - modified license with relaxed terms							

Table 3.3: Comparison of password crackers

Cain & Abel and commercial tools like Elcomsoft or AccessData software are monolithic applications with proprietary API and communication protocols. Their source code is not publicly available, and the only way of controlling their operations is via the provided graphical user interface. The only exception is Passware Kit Forensic that offers a .NET SDK that allows integration of the Kit into third-party applications. Another obstacle is licensing issues. The license agreements of AccessData, Elcomsoft, and Passware forbid the creation of any derivative works based on the licensed software [4, 58, 149]. From the open-source software discussed, the only feasible candidates are JtR, hashcat, or RainbowCrack. Since RainbowCrack is a single-purpose application purely for rainbow table attacks, the final choice lies between JtR and hashcat.

Both tools are powerful and provide excellent features and performance. Hashcat offers a broader range of supported formats. Moreover, it is the only tool from the list that offers full GPU support. Recent versions of hashcat have all cryptographic algorithms implemented within OpenCL kernels, allowing GPU-accelerated cracking but does not restrict using OpenCL-compatible CPUs. The rule engine for mangling dictionary words is, unlike in any other tool, implemented for GPU as well. John has the OpenCL support for many formats, but not for all of them. On the other hand, John still offers some features that hashcat did not have in the current 6.1.1 release: the Single crack attack mode or support for RAR 3 archives without an encrypted header. It seems, however, hashcat's developers are already

working on improvement. The newly-announced Association[34] attack mode is inspired directly by John's Single crack mode. The development version already includes support for RAR3-p hash mode for cracking both compressed and uncompressed RAR version 3 archives without the encrypted header [95]. Moreover, hashcat has a native support for GPU-based combination and hybrid attack modes. Performing the same with JtR requires an external password generator.

The final criterium for selecting the cracking engine is the support for the decomposition of cracking sessions. Sadly, JtR does not track the starting and ending positions of generated password segments, and thus the options for workload distribution are very limited. It is necessary to either use the internal MPI support, define the keyspace splitting manually in the configuration file, or use an external password generator [12]. Crumpacker reported the same issue and had to modify the JtR's database to distribute the incremental attack mode correctly, but at the cost of efficiency [47].

In contrast, hashcat's `--skip` and `--limit` parameters allow to precisely define the range of candidate passwords that are verified within a given task. This integrated feature allows to easily create chunks of work and distribute them between the computing nodes. The only obstacle is hashcat's optimization of several attack modes, where the keyspace numbers do not directly correspond to the password guess count. I further discuss this phenomenon in Section 3.6.4. Fortunately, the distribution strategies that I propose in Section 3.8 resolve this issue [47].

All hashcat's features work on both Windows and Linux. The tool is maintained and improved over time, well-documented, and is surrounded by a large community of supporters that frequently discuss its features on web forums[35]. Hashcat is open-source and distributed under the MIT license. Therefore, integration into the password cracking system is possible without any limitations.

For the reasons discussed above, I eventually decided to use the hashcat tool as the cracking engine of my distributed password cracking solution. The following sections document the concrete techniques, algorithms, and strategies to perform cracking sessions in a distributed environment.

[34] https://hashcat.net/forum/thread-9534.html
[35] https://hashcat.net/forum/

3.6 Workload Distribution in Cracking Tasks

This section discusses the principles of dividing workload amongst multiple cracking nodes. First, I describe the general principles and unify the terminology. Then, I propose a classification of distribution options into three schemes. Finally, I contribute with a proposal of job processing with BOINC and hashcat utilized in the Fitcrack system. My concept covers benchmarking of hosts, scheduling jobs with an adaptive calculation of processing time, and optimizations like pipeline processing.

3.6.1 Essentials

Let P be a finite ordered set of all candidate passwords for a given attack. Assume that every candidate password $p \in P$ is a string over Σ alphabet, thus $p \in \Sigma^*$. Naturally, $P \subset \Sigma^*$. The total number of candidate passwords within a cracking job is called *keyspace*. The National Institute of Standards and Technology (NIST) defines keyspace as "the total number of possible values that a key, such a password, can have" [180]. Therefore, the keyspace of an attack equals the cardinality $|P|$ of P.

Definition 1 (Keyspace) *A keyspace is the number of all password candidates for a given attack. Keyspace $k = |P|$ where P is the set of all password candidates.*

Based on the definitions above, It is possible to define a *password generator* function $g(i) : N \mapsto P$, where $i \in \langle 0, |P| - 1\rangle$ and i is called a *password index*. The function maps every single index from 0 to keyspace $-$ 1 to a concrete password candidate. What candidate passwords P contains and wow the indexes are mapped is defined by an attack mode.

Let us consider a simple incremental *incremental brute-force attack* (see Section 3.8.3), where we want to generate all password of lenghts between 1 and 3 over alphabet $\Sigma = \{a, b, c, \ldots, z\}$. Then:

$$\begin{aligned} g(0) = a,\ \ldots,\ g(25) = z, \\ g(26) = aa,\ \ldots,\ g(701) = zz, \\ g(702) = aaa,\ \ldots,\ g(18277) = zzz. \end{aligned} \tag{3.3}$$

We can see, the domain of g corresponds to the keyspace $k = |P|$ of the attack. For a dictionary attack (see Section 3.8.1) with a wordlist containing "alpha", "bravo", "charlie", the $g(i)$ is defined as follows:

$$g(i) = \begin{cases} \text{alpha} & \text{if } i = 0, \\ \text{bravo} & \text{if } i = 1, \\ \text{charlie} & \text{if } i = 2. \end{cases} \tag{3.4}$$

Definition 2 (Attack mode) *An attack mode is the technique how candidate passwords are created. Every attack mode may have additional parameters (settings) that need to be specified. Together with all necessary parameters, the attack mode defines the set of all password candidates P, the password generator function $g(i)$, and the keyspace k.*

Definition 3 (Password index) *A password index is an integer that uniquely identifies a candidate password for a given attack mode and settings. Every password index i belongs to the domain of the password generator function g: $i \in Dom(g) \subset \mathbb{N}_0$.*

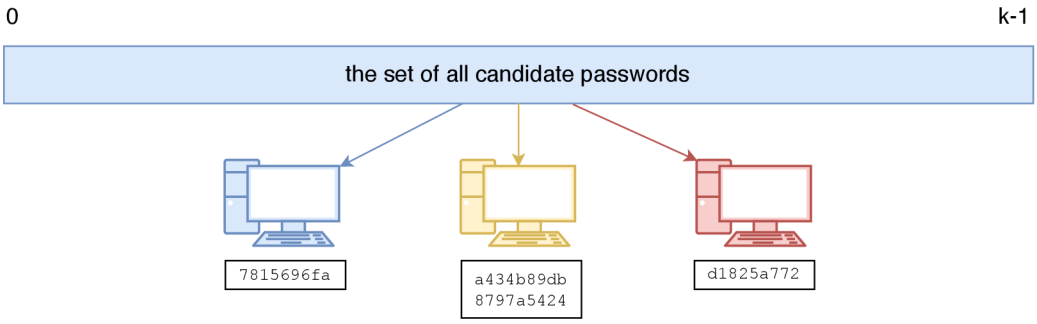
3.6.2 Distribution Schemes

In the terminology of Fitcrack, a *job* represents a single cracking task added by the *administrator*. Each job is defined by an attack mode (see Section 3.8), attack settings (e.g., which dictionary should be used), and one or more password hashes of the same type (e.g., SHA-1). The type of the hash algorithm is sometimes called the *hash mode*. After studying related work and analyzing possibilities, I see three common ways of distributing a password cracking job over multiple nodes. I define these approaches as follows:

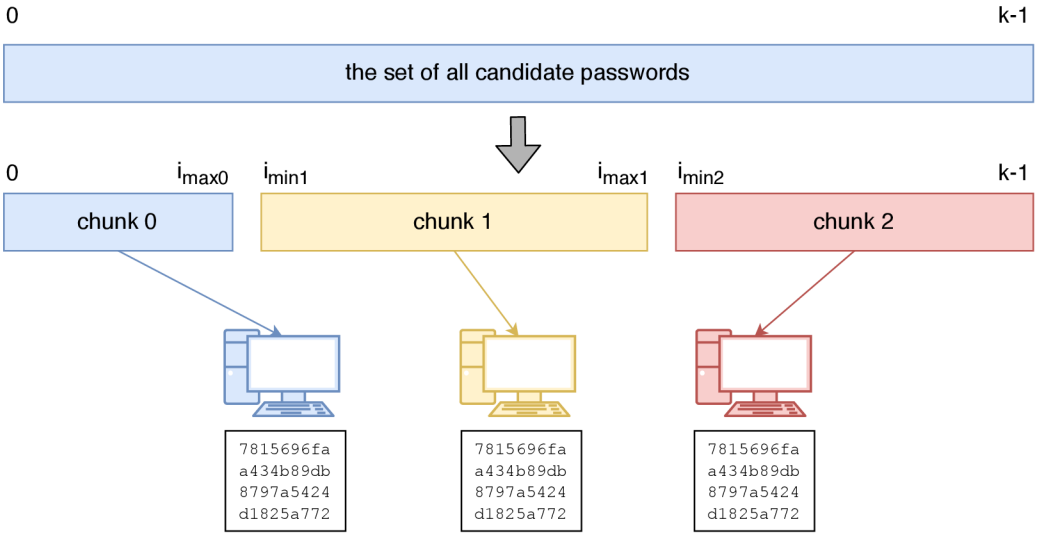
- **Hash distribution** described by Pippin et. al. [162] uses the same candidate passwords on all nodes, however each node is cracking a different hash. The calculation of the password hash is usually the most computationally complex part. Since tools like hashcat or John the Ripper are capable of cracking multiple hashes for each candidate password while the candidate hash is only generated once, I do not consider hash distribution to be an efficient method for dividing the workload. I suppose it only makes sense if cryptographic salt is used or if one deals with exceptionally large hashlists. Otherwise, there is no need to calculate the hash more than once from each candidate password. An example of hash distribution is shown in Figure 3.5(a). In the model situation, three computers receive the entire set of candidate passwords, but each computer has different hashes assigned.

- **Static chunk distribution** introduced by Lim et. al [112] divides the set of all candidate passwords into a number of *chunks* and assignes a chunk to each client. The division is done only once at the beginning. The method has low overhead, but cannot handle changes in cracking network. If a chunk is lost, it has to be recomputed from the beginning, if no method of checkpointing is implemented. Figure 3.5(b) shows the same attack with static chunk distribution. In contrast with the previous case, every computer receives the entire hashlist. The set of password candidates is, however, split into three chunks with sizes set accordingly to the performance of each computer. In the example, yellow and red computers have higher computing capacities than the blue one, so they receive bigger chunks.

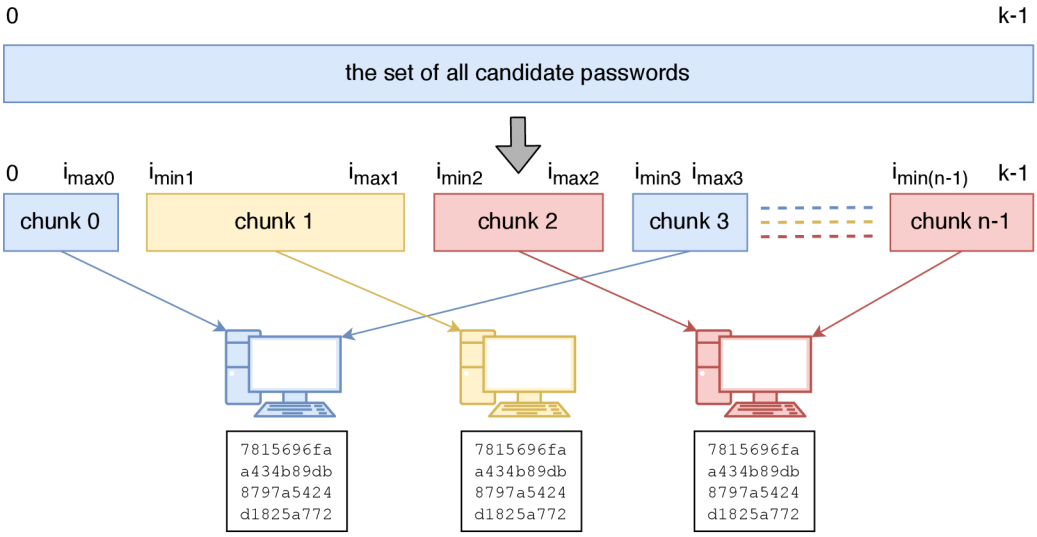
- **Dynamic chunk distribution** does not divide the entire set of candidate passwords at start. Instead, it generates and assigns smaller chunks called *workunits* progressively. This method is used in Fitcrack since it better handles dynamic and unstable environment. The dynamic approach allow to create workunits which are fine-tailored for the current client speed. Moreover, losing the result of a workunit has lower impact due to its size. A similar method with different scheduling algorithm is used by the Hashtopolis tool that also divides each task into smaller chunks. An example is illustrated in Figure 3.5(c). First, the blue computer receive a little chunk taken from the beginning of the set. Then, the yellow computer gets a next chunk of passwords. The chunk is bigger since the yellow computer has a higher performance than the blue one. Next, the chunk is assigned to the red one. When the blue one is finished with "chunk 0", it receives another one. The same follows until eventually the entire set is processed.



(a) Hash distribution



(b) Static chunk distribution



(c) Dynamic chunk distribution

Figure 3.5: Common workload distribution schemes

3.6.3 Workunits in Fitcrack

In BOINC, the chunks of work assigned to computing nodes are called workunits, and the computers that process them are called hosts. Fitcrack adopts the same terminology. As mentioned above, the system uses the dynamic chunk distribution scheme. The division of the candidate password set is based on password indexes. Therefore, each workunit is defined by the range of indexes: i_{min} a i_{max} while

$$0 \leq i_{min} \leq i_{max} \leq (|P| - 1). \tag{3.5}$$

The index ranges are also denoted in Figure 3.5. Each workunit (chunk) has the minimal and maximal password index. The first one (chunk 0) begins with index 0 and the last workunit (chunk n-1) ends with index $k - 1$ where k is the keyspace of the attack.

The actual work lies in trying ever possible passwords given by the generator function $g(i)$ where $i \in \langle i_{min}, i_{max} \rangle$. The workunit may end in two ways:

- **One of candidate passwords is correct** (or more, if we crack multiple hashes) - the client informs the server that it has found the correct password. If all hashes are cracked, the client stops.
- **No candidate password is correct** - client tried every password within the range, but none of them was correct.

The entire *job* may end in two possible ways:

- **Sucess**, if the correct password was found within a workunit.
- **No sucess**, if all workunits were processed, however the correct password was not found.

In Fitcrack, the creation of *workunits* is handled by the *Generator* module (see Section 3.7.1) which specifies the range of indexes for each workunit. The size of the workunit is calculated using the *adaptive scheduling algorithm* described in section 3.6.5.

3.6.4 The Keyspace in Hashcat

Since Fitcrack uses the hashcat tool as the internal cracking engine, it is necessary to consider its design and properties. Theoretically, the index-based distribution described in Section 3.6.1 should work with any password guessing subsystem that allows defining where to start and how many password candidates to generate. In hashcat, however, the understanding of a password index is slightly different from other tools. To design distribution strategies, I consider it necessary to emphasize this unique property.

Hashcat tool used for the actual cracking is controlled by *Runner* subsystem on the client side. The range of indexes defined above can be set by `--skip` and `--limit` parameters. While `--skip` corresponds to i_{min}, `--limit` defines the keyspace to be processed within a workunit, i.e. should be equal to $i_{max} - i_{min}$.

Whereas for a dictionary attack without the use of *password-mangling rules* (see Section 3.8.1), hashcat's keyspace equals the actual number of candidate passwords, for other attack modes, it may not match. This unexpected behavior is used by the internal optimization of hashcat. The hashcat's cracking process is implemented as two nested loops: i) the *base loop* and ii.) the *modifier loop*. While the base loop is compute on host machine's CPU, the

modifier loop is implemented within OpenCL GPU kernels. Hashcat's keyspace is equal to the **number of iterations of the base loop** and **could be different** from the actual number of password guesses.

For example, assume a brute-force attack using mask (see Section 3.8.3) `?d?d` which stands for two digits. We can generate 10 different digits on each position, so the keyspace of the mask should be $10 * 10 = 100$, however in hashcat, it is only 10 since it computes 10 iterations within the base loop, and the other 10 within the nested modifier loop. In that case, running hashcat with `--limit 1` causes to try 10 passwords, not only one. To overcome this obstacle, we let hashcat calculate the keyspace on the server before the actual work is assigned to the clients. And in our database (see Section 3.7.8), we store both hashcat's keyspace which is used for distributing work, and the actual keyspace, to inform the user about the actual number of passwords processed.

Moreover, the calculation also depends on the concrete cryptographic algorithms calculated with OpenCL kernels. For instance, mask `?d?d?d?d?d?d` defines all passwords made of 6 digits. The actual number of password guesses is thus 10^6. For cracking encrypted MS Office 2013 documents (hash mode 9600), hashcat's keyspace is 10^5, while for MS Office older than 2003 (hash mode 9700), hashcat's keyspace is 10^4. See the command line output:

```
./hashcat64.bin -a 3 ?d?d?d?d?d?d --keyspace -m 9600
100000
./hashcat64.bin -a 3 ?d?d?d?d?d?d --keyspace -m 9700
10000
```

I assume the optimization is integrated to reflect different time and space complexities of the algorithms involved. Older MS Office versions used the RC4 stream cipher and MD5 [172] or SHA-1 hash algorithms [97]. MS Office 2013 uses AES [50] cipher and SHA-2 hashing [138]. For details see Section A.1.2 and Section A.1.3.

Password-mangling rules even complicate the situation since the actual guess count is multiplied by the number of applied rules. Other deviations occur with combination and hybrid attacks, described in Section 3.8.2 and Section 3.8.4. Hashcat creates candidate passwords from two parts: left and right. Each is either a dictionary word or a string generated from the mask. The `--skip` and `--limit` parameters only allow to control the left side. Hashcat may thus generate hundreds of password guesses even with `--limit 1`. For better or worse, these deviations from a traditional password index concept should be taken into account. In Section 3.8, I propose distribution strategies that work even with the above-described optimizations.

3.6.5 Adaptive Scheduling

In previous sections, I described the essentials of dynamic chunk distribution usable for performance-based scheduling of workunits. This section shows how to utilize this principle in BOINC and calculate the proper size of workunits. One of the main reasons for choosing BOINC is the integrated technique called *targeting* that defines which workunit is assigned to which host. The framework supports two types of workunits:

- **non-targeted** - the workunit is created without targeting, and will be assigned to **any** host who asks the server for work;
- **targeted** - the workunit is created **for** a specific host, and will be assigned to this host only. This approach is used in Fitcrack, and will be described in the following paragraphs.

In a dynamic heterogeneous environment, working nodes may have different performance, based on their hardware. They can also dynamically join and leave the computing. In addition, the performance of a node can change over time. My goal is to propose a distribution technique that maximizes efficiency of the computing process. Concretely, to fulfill the following objectives:

- Utilize as many available hosts as possible — ideally, all of them.
- Make all hosts utilized most of the time — ideally, all the time.
- Adjust the size of each workunit equally to hosts' current performance.
- Preserve some work for newly connected hosts.

Therefore, I use the targeted workunits and propose an algorithm [81] for adaptive calculation of workunit size. The algorithm's idea is to estimate how much time it would take to verify the remaining candidate passwords on all the active clients. From such estimation (and various other settings), it chooses the desired processing time for a workunit. Based on this time, we assign an appropriate part of remaining keyspace to a host asking for work. Its size depends on the node's current performance (cracking speed). This means that the higher-performance clients receive larger workunits than the lower-performance clients.

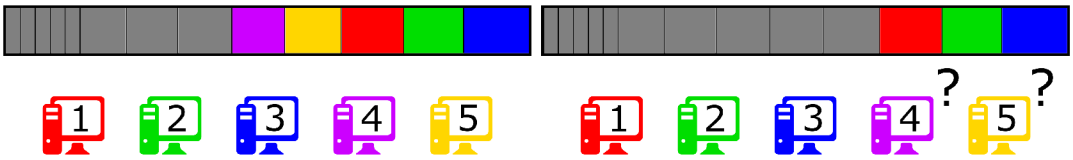
Let $P_R \subseteq P$ be the set of all remaining password candidates that need to be verified. Next, let t_p be the desired workunit processing time in seconds described above. Finally, let v_i be the current performance (cracking speed) of node i in passwords per second. Then, the size s_i of a new workunit assigned to node i is calculated as $s_i = min(t_p \cdot v_i, |P_R|)$. Speed v_i is determined from previously solved workunit as $v_i = \frac{s_{prev}}{t_{prev}}$ where s_{prev} is the size of a previous workunit assigned to the node, and t_{prev} is the time spent by its processing. The problem is how to choose v_i for a newly connected client. The solution used in Fitcrack and Hashtopolis is to run a *benchmark* on the client to calculate its performance.

Workunit Processing Time

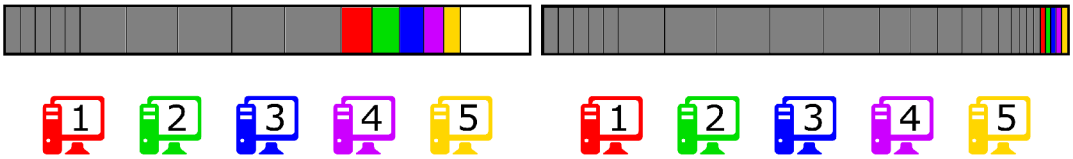
The above-shown description purposedly omits an important step — choosing the workunit processing time t_p. This critical variable affects the distribution's granularity. By specifying t_p, we define how long we want a host to process a workunit. Concretely:

- Lower t_p means more smaller workunits. Such a setting is more suitable for an unstable environment where clients are more likely to fail, frequently disconnect or change their performance. And thus, the impact of a lost workunit is lower, and the task can be assigned to another client. On the other hand, lower t_p implies higher overhead because more communication between the server and clients.
- Higher t_p results in a less number of larger workunits. It decreases communication overhead and clients spend more time by computing. In case of lost connection, recovery is longer. Higher t_p also causes less effective task distribution, namely at the end of the project. E.g., suppose 20 clients where only 10 nodes are computing. These active nodes will be computing for another hour while others stop working since there is no more task assigned to them.

In Hashtopolis, the t_p is defined purely by a user and is constant throughout an entire task. The setting is called the *chunk size*. Having a fixed workunit size may, however, cause a series of undesirable phenomenons.



(a) End of a job without ramp-down



(b) End of a job with the ramp-down

Figure 3.6: Illustration of the ramp-down

Firstly, the initial benchmark is often inaccurate. As I experimentally detected, Hashcat's measurement with the `--benchmark` option gives a theoretically achievable cracking speed. The actual performance is lower and depends on the attack mode and other settings. We can get more precise values using the `--speed-only` parameter. An alternative, currently used in Fitcrack, is running a live cracking session for a short time. This approach also respects the use of cryptographic salt. Both alternatives, however, require to specify additional settings, some of which are not known. The workunits are created on-demand so that the system can not predict the future. For example, we do not know what exact masks or dictionary fragments will be assigned to what nodes. Having smaller-than-desired workunits is not a big obstacle since the system gets precise performance info and quickly adapts. Creating enormously large workunits is a much more significant problem. Imagine a user specifies 30-minute workunits, and the system creates a 10-hour one. Not only the user may get annoyed, but the system may also terminate the process due to an exceeded deadline that is set for each workunit. Therefore, Fitcrack's scheduling algorithm creates smaller workunits at the start of each job to ensure the system adapts appropriately. The technique is called *ramp-up*. The technique also serves as a fail-safe mechanism to minimize the implications of unexpected host behavior. Potential failures range from GPU overheating, lack of memory, through network problems up to possibly compromised nodes. This „suspicious“ property of the algorithm does not let the host process full-sized workunits before it proves the ability to resolve smaller ones.

Secondly, if the hosts are not assigned to another running job simultaneously, they may not be utilized well at the end. An example is described in Figure 3.6(a), which illustrates the keyspace distribution. There is a total of five hosts, each having assigned a workunit of its color. Hosts 4 and 5 finish their workunits before the others. Since there is no more keyspace left to distribute, they do not receive any work. Even if it takes hours for hosts 1 and 3 to finish, the other nodes are not employed. The computing resources of hosts 4 and 5 remain idle. Fitcrack's scheduling algorithm creates progressively smaller workunits at the end of the job. The solution is called *ramp-down*, and the goal is to ensure all hosts compute most of the time. The solution is illustrated in Figure 3.6(b) and allows for more efficient utilization of network resources.

When a user creates a new job in Fitcrack, the user specifies the *seconds per workunit* value. This number has a similar meaning as the chunk size in Hashtopolis. The adaptive

scheduling algorithm, however, applies the principles of ramp-up and ramp-down described above.

To properly choose t_J for each workunit, I define function $proctime(t_J, |P_R|, k)$ that adaptively computes expected process time t_p till. The parameters are defined as follows: t_J is the elapsed time of current job, $|P_R|$ is the number remaining passwords guesses, and k is the number of active hosts that participate on the computing. Parameters t_J, s_R and k change over time. The function *proctime* is computed using Algorithm 1. Based on remaining time t_p, each node will be assigned appropriate keyspace $s_i = min(t_p \cdot v_i, |P_R|)$. Therefore, the remaining keyspace will be distributed among working nodes according to their performance. In an optimal case, all nodes complete their tasks in t_p as estimated.

Algorithm 1: Adaptive calculation of t_p

Input: $t_J, |P_R|, k$
Output: t_p

1: $v_{sum} = 0$
2: **forall** $client_i \in \{0, \ldots, k\}$ **do**
3: **if** $client_i$ *is active* **then**
4: $v_i = \frac{s_{prev}}{t_{prev}}$
5: $v_{sum} = v_{sum} + v_i$
6: $t_p = \frac{|P_R|}{v_{sum}} \cdot \alpha$
7: **if** $t_p < t_{pmin}$ **then**
8: $t_p = t_{pmin}$; `// minimal workunit time`
9: **else if** $t_J > t_{pmax}$ **then**
10: $t_p = min(t_p, t_{pmax})$; `// maximal workunit time`
11: **else**
12: $t_p = min(t_p, t_J)$; `// ramp-up`
13: **return** t_p

Lines 2 to 5 of the algorithm compute the entire speed of all active nodes. Line 6 is a bit tricky. Normally, we would have calculated t_p as $t_p = \frac{|P_R|}{v_{sum}}$. Here, I multiply the value by parameter α called *distribution coefficient* that ranges from 0 to 1. This parameter ensures that only a fraction of the remaining keyspace is assigned each time. E.g., $\alpha = 0.1$ means that maximally 10% of the remaining keyspace $|P_R|$ is assigned. Firstly, it guarantees that if additional hosts connect to the network, there is always a piece of work for them. Secondly, it assures the ramp-down of workunit size. The workunits get progressively smaller as we are close to the end of the job. The motivation for the ramp-down was discussed above and illustrated in Figure 3.6.

The value of t_p is also limited by t_{pmin} and t_{pmax}. Parameter t_{pmin} states, that the computing shorter than this value is ineffective in distributed environment, so the minimal task time is t_{pmin}. Similarly, t_{pmax} defines the maximal task time so that also slower nodes can participate in the computing. Based on my experiments, I recommend t_{pmin} to be at least one minute and t_{pmax} to be about 1 hour. When creating a new job in Fitcrack WebAdmin (see Section 3.7.6), the administator can specify t_{pmax} as the *seconds per workunit* option. The t_{pmin} and α are system-wide parameters and can be modified through WebAdmin's system settings via the "System preferences" tab. Finally, line 12 performs the ramp-up in an initial stage of the job. This stage is defined as the period

before the elapsed time reaches the desired workunit time. For example, the desired time for a single workunit is 15 minutes. But the full-size workunits are not created in the first 15 minutes of the job. The algorithm reserves this time for stabilization to withstand any benchmark inaccuracies or unexpected host reactions. The actual impact of the algorithm is shown by experiments in Section 3.9.2.

Improved Benchmarking

As described above, the performance obtained by hashcat using the benchmark mode are often far different from the actual cracking performance. The reason is that it only calculates a theoretically achievable number of calculated hashes per second. Yet, there are many other factors in the game that influence the actual performance. Those include the settings of the attack, disk i/o speed, amount of available memory, utilization by different running processes, and others.

Therefore, the newer versions of Fitcrack come with a redesigned benchmarking scheme. Instead of specifying just an algorithm or attack mode only, the system runs an actual cracking session for a short time. With dictionary-based attacks, the actual performance also depends on the length of passwords in a given fragment. At the beginning of the job, we do not know what particular dictionary fragments will the hosts receive. Therefore, Fitcrack introduces the `pwd_dist` utility that calculates the distribution of password lengths when users upload the dictionary. For the benchmark session, the host creates a dummy dictionary that matches the given distribution, as described in [87]. Experiments in Section 3.9.2 show that the modified solution provides far more accurate results.

Pipeline Processing

During the experiments with dictionary attacks (see Section 3.9.3), I detected that wordlist distribution has high requirements for network bandwidth. Especially with larger wordlists and less complex hash algorithms, the overhead is extensive. The hosts literally wait for new candidate passwords so that they can verify them. The efficiency of such attacks is very low [86, 84].

I found a solution in the form of *pipeline workunit processing*. The concept is quite simple. Setting the `max_wus_in_progress` parameter in the BOINC project configuration to 2 makes the server send not one, but two workunits to every host. At the time the host is processing a workunit, it can be downloading another one. Once the host finishes the first one, it can immediately switch to the next one without waiting. The only challenge was to prevent the BOINC client from starting more than one instance of the Runner and hashcat. The first solution was to specify the `max_concurrent` value in `app_config.xml` file and set it to 1. This configuration file is, however, created automatically after connecting to a project's server. Therefore, this is not an out-of-the-box solution. The user either has to configure the client manually, or Fitcrack would need to use a modified version of the BOINC client. In 2020, together with my fellow researchers, I later discovered an alternative solution. A new global mutex introduced to the Runner application prevented to start another instance of hashcat. Even if multiple Runner processes are started, there is maximally one hashcat process.

The most recent version of Fitcrack has the pipeline processing on by default since it almost eliminates the overhead for workunit distribution. To see the comparison of the original and improved versions, visit Section 3.9.3.

3.7 The Architecture of Fitcrack

This section briefly describes the architecture of Fitcrack[36] - a BOINC-based distributed password cracking system that I originally proposed in 2016 [81]. The initial release[37] utilized custom OpenCL and CUDA kernels for cracking PDF, ZIP, 7z, RAR, and MS Office up to version 2003. For the reasons discussed in Section 3.5, I later decided to replace my kernels with the hashcat tool. Therefore, I redesigned the system completely, and the new proof-of-concept implementation was made by the Fitcrack team[38] under my supervision. In this section, I focus on the novelty hashcat-based solution. Due to the system's complexity, I use a high level of abstraction. Detailed documentation of Fitcrack is available in a separate technical report that I will refer to [87].

Figure 3.7 illustrates the architecture of the Fitcrack system. Similarly to other related projects [221, 47, 105, 154], the solution consists of a *server* and a *client* part. The server and clients are interconnected by a TCP/IP network, not necessarily only LAN which makes it possible to run a cracking task over-the-Internet on nodes in geographically distant locations. Clients communicate with the server using an RPC-based *BOINC scheduling server protocol*[39] over HTTP(S) [87, 84]. The two sides of the system play the following roles:

- **Server** - The server is responsible for managing cracking jobs and assigning work to clients. In our terminology, a *job* represents a single cracking task added by the *administrator*. Each job is defined by an attack mode (see Section 3.8), attack settings (e.g., which dictionary should be used), and one or more password hashes of the same type (e.g., SHA-1). Once the job is running, Fitcrack progressively partitions the keyspace into smaller chunks called *workunits*. In terms of the *client-server* architecture, the server provides a workunit assignment service since hosts actively ask for new work. Using the adaptive scheduling algorithm described in Section 3.6.5, the keyspace of each workunit is calculated to fit the performance of a host that should compute it.

- **Client** - The clients in BOINC are called *hosts*. In Fitcrack, hosts represent the actual cracking nodes. A Fitcrack host can be any machine with Windows or Linux OS, and at least one OpenCL-compatible device with proper drivers installed. The only piece of software that needs to be installed is the *BOINC Client* (see Section 3.7.9), and optionally the *BOINC Manager* (see Section 3.7.10) providing a graphical user interface to the BOINC Client. Once a host connects and authenticates to the server, it automatically downloads all necessary binaries before the actual work is assigned. The binaries involve two applications: *hashcat* as the "cracking engine" (see Section 3.7.12), and the *Runner* (see Section 3.7.11) which serves as a wrapper encapsulating and controlling operations with hashcat. Besides, the host may also download an external password generator if necessary for the given attack mode.

The following sections describe the essential subsystems of both server and client sides.

[36] https://fitcrack.fit.vutbr.cz/
[37] https://fitcrack.fit.vutbr.cz/download/download-archive/
[38] https://fitcrack.fit.vutbr.cz/team/
[39] https://boinc.berkeley.edu/trac/wiki/RpcProtocol

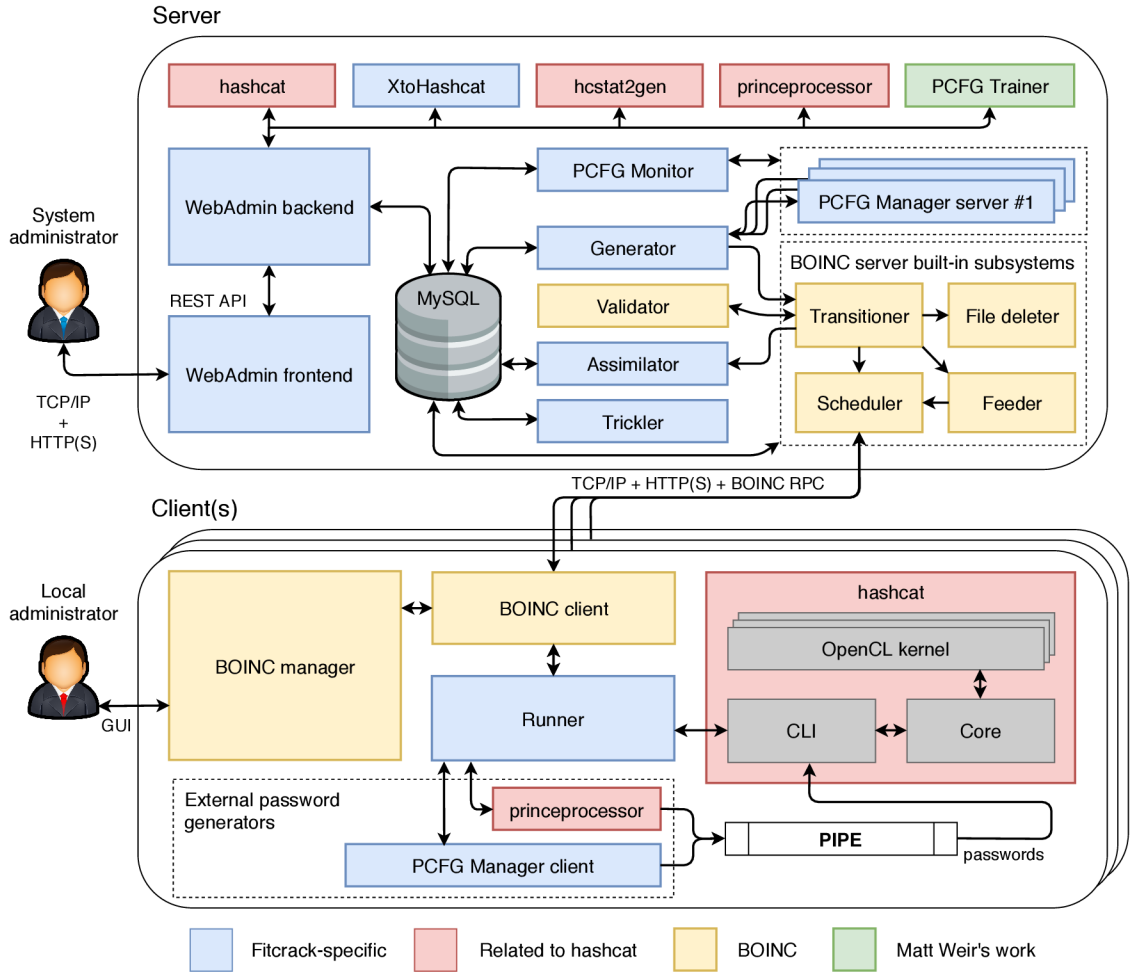


Figure 3.7: The architecture of Fitcrack server and client

3.7.1 Generator

The *Generator* is a server daemon responsible for creating new workunits and assigning them to hosts. To achieve an efficient use of network's resources, it employs the *Adaptive scheduling algorithm* [81] described in Section 3.6.5. The algorithm tailors each workunit to fit the client's computational capabilities based on the current cracking performance, which could change over time. To get the initial speed, at the beginning of each cracking job, the clients receive a *benchmark* job which measures their cracking speed for a given hash type. For the attack modes, the Generator implements the distribution strategies described in Section 3.8. For example, it performs the fragmentation of dictionaries, calculates the appropriate password index boundaries, loads the preterminal structures for the PCFG attack, etc.

There are three types of workunits: a) regular cracking tasks, b) benchmark workunits, and c) a complete benchmark. *Regular workunits* are used to verify a set of candidate passwords. For each such workunit, the Generator calculates an appropriate keyspace and prepares all necessary input data and files [87]. The *benchmark workunits* are created at the start of a job. The goal is to measure each host's actual performance for a given cryptographic algorithm and attack mode. The *complete benchmark* is an optional feature

that can be turned on or of in Fitcrack's settings. It is performed automatically at the time a completely new host connects. The complete benchmark is run only once for the entire existence of the host in the system. It measures the achievable performance for all supported hash algorithms. The results serve, above all, for estimating the cracking time of a new job.

status	name	description
0	ready	Job is ready to be started.
1	finished	Job is finished, one or more hashes cracked.
2	exhausted	Job is finished, no password found.
3	malformed	Malformed due to incorrect input.
4	timeout	Job was stopped due to exceeded time schedule.
10	running	Computation is in progress.
12	finishing	All keyspace assigned, some hosts still compute.

Table 3.4: Job *status codes* in Fitcrack

status	name	description
0	benchmark	The host is waiting for, or working on a benchmark.
1	normal	The hosts is working on a cracking job.
3	done	The host has finished all work on the given job.
4	error	The host encountered an error during the computation.

Table 3.5: Host *status codes* within a job

Each job in Fitcrack goes through a series of states. All possible states are enlisted in Table 3.4, while each has a unique numeric identifier from 0 to 12. Numbers above 10 mean the job is not running. Historically, not all are numbers are used; some are reserved for future use. The lifetime of a job is illustrated in Figure 3.8. Once created, the job is in the Ready state. Clicking the start button changes the state to Running. The following transitions depend on the conditions. If there is at least one non-cracked hash and no error or user action occurs, the job eventually proceeds to the Finishing state. In this state, the entire keyspace is distributed, but some hosts are still processing. Once all hosts are done, the job switches to either Finished or Exhausted state, depending on the results. A user may optionally specify a deadline for the job. If exceeded, the job ends in the Timeout state. In case of a non-recoverable error (e.g., the database gets corrupted), the job switches to the Malformed state.

In Fitcrack, three subsystems are allowed to change the state of the job: the Generator, when a host asks for a new workunit, the Assimilator if a workunit result is received, and the Webadmin at an event of user's action. Every job starts in the *ready* state, created by a user, and added to the database by the WebAdmin backend. Once the user launches it, the job switches to the *running* state. All hosts assigned to a job also have status codes defining the stage of their participation. The host codes are shown in Table 3.5.

The Generator daemon runs in a loop illustrated by Algorithm 2. It takes care that for each running job, there is always at least a single workunit assigned. And if possible, all participating hosts have a workunit. Each participating host needs to perform a benchmark workunit first. Once benchmarked, the host may receive regular workunits. If possible, there

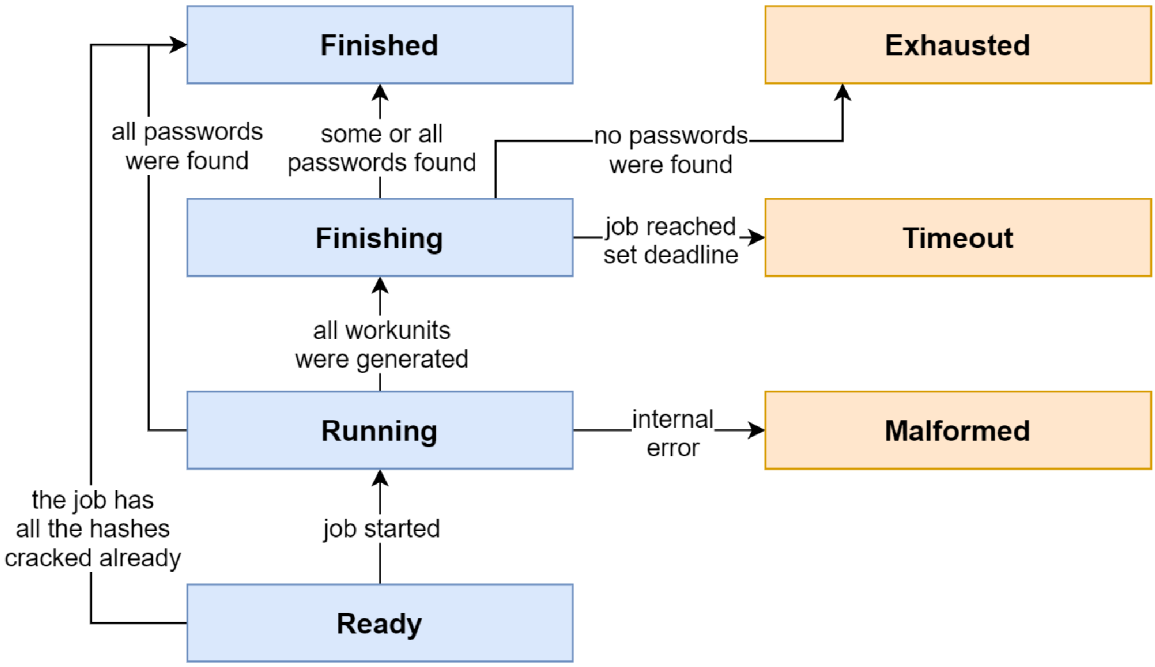


Figure 3.8: State diagram of a job in Fitcrack system

are always two cracking workunits ready for each host within the job. One is sent to the host. The second one is generated beforehand, so the host can start working on it right after the first one is completed. The goal is to reduce the overhead for creating new workunits. Moreover, Fitcrack supports the pipeline workunit processing. If the `max_wus_in_progress` parameter of the BOINC Server is 2, two workunits are sent to the client. After completing the first one, the BOINC Client switches to the second one immediately. At the time the second workunit is processed, another one can be transferred over the network.

The daemon also deals with disconnected hosts and computation errors. When a host delivers an incorrect workunit result, or when the processing reaches a pre-defined deadline, the workunit is tagged with *retry* flag, and a the Generator reassigns it by creating a copy of the original workunit. Another feature is job purging. When the user chooses to purge the job, revert all the work done, and delete the progress, the Generator sends a special signal to all connected clients to abort the current task. When the purged job is running, the Generator also reverts its state back to *ready*.

3.7.2 Validator

The *Validator*[40] is a tool that verifies the syntax of all incoming workunit results from clients before they are passed to the Assimilator. The subsystem also checks if each result contains all necessary output files. If the job *replication* is active, i.e., a single workunit is assigned to more than one host, the Validator verifies if the replicated results match. The technique helps in an untrusted network where we expect hosts may be compromised produce intentionally incorrect results. In Fitcrack, the replication is by default disabled since it reduces the computational power by 50% or more, as discussed in [81].

[40] https://boinc.berkeley.edu/trac/wiki/ValidationIntro

Algorithm 2: Generator daemon algorithm

```
1  while (1) do
2      // Iniciatization
3      if Any Jobs reached deadline then
4          Set them to Finishing status (12).
5      foreach Running Job (status ≥ 10) do
6          Process the purge requests. Load all corresponding masks or dictionaries.
7          // Benchmark
8          foreach Host in Benchmark status (0) do
9              if Benchmark is not planned then
10                 Plan a benchmark.
11         // Cracking
12         foreach Host in Normal status (1) do
13             if Number of planned workunits ≥ 2 then
14                 Continue to next Host.
15             if Job is in Running status (10) then
16                 Generate a new workunit or reassing a retry workunit.
17                 if No workunits could be generated then
18                     Set Job to Finishing status (12)
19             if Job is in Finishing status (12) then
20                 Reassign a retry workunit if exists. Otherwise, set Host to Done (3).
21         // Job finished
22         if Job status is Finishing (12) and no Jobs are generated then
23             Check the end conditions and set job to
                 Finished/Exhausted/Timeout/Paused.
24     Wait a short time interval before the next iteration
```

3.7.3 Assimilator

The *Assimilator*[41] is a server daemon that parses the workunit results sent by hosts and checked by the Validator. It decides what to do when a host completes a workunit. Depending on the result, the Assimilator can modify the database or even cancel running workunits. Algorithm 3 describes a simplified functionality of the Assimilator. There are three options, how a workunit could end:

- **Successful benchmark** of a node - The Assimilator saves the node's cracking performance to the database.
- **Finished regular job** - The Assimilator updates the job progress. If the host cracked one or more hashes, the passwords are saved to the database. If all hashes are cracked, the entire job is considered done, and all ongoing workunits are terminated. If the whole keyspace is processed, the Assimilator changes the job's status to either finished or exhausted.

[41] https://boinc.berkeley.edu/trac/wiki/AssimilateIntro

- **Computation error** - The Assimilator sets the *retry* flag to the workunit to perform the failure-recovery process, as described in [81].

Algorithm 3: Assimilator daemon algorithm

```
1  while (1) do
2      Read the result type
3      switch type do
4          case benchmark do
5              if Result is OK (code 0) then
6                  Read the power and save it to database
7              else
8                  Plan a new benchmark
9          case normal do
10             if One or more passwords found (code 0)) then
11                 Read the password(s) and save them to database
12                 if Any hashes remaining to be cracked then
13                     Switch the Job state to Finished (1)
14                     Cancel all running Workunits of the Job
15                     Set finished flag to all Workunits
16                     Read the cracking time and save it
17             else
18                 if No passwords found (code 1) then
19                     Modify the workunit size according to Algorithm 1.
20                     Update the current index used for planning
21                 else
22                     // Computation error
23                     Cancel host workunints
24                     Set Host status to Benchmark (0)
25         case bench_all do
26             if Result is OK (code 0) then
27                 Read the power list and save it to database
28             else
29                 Plan a new benchmark
```

3.7.4 Trickler

With the Generator, Validator, and Assimilator, the server knows what host has which workunit assigned. However, the only information about the workunit's progress the server obtains when the host finishes its work and sends a report. To provide the administrator with a more detailed overview, the Runner (see Section 3.7.11) uses BOINC Trickle message API[42]. Via this API, each host periodically sends XML-based *Trickle messages* that

[42] https://boinc.berkeley.edu/trac/wiki/TrickleApi

inform the server about partial progress. The goal of the *Trickler* daemon is to process these messages and updates the information in the database. The administrator may then visualize it in the *WebAdmin* application. This way, the server knows the state of cracking even with several hours long workunits.

3.7.5 BOINC Server Subsystems

Besides the previously denoted applications, Fitcrack uses the following subsystems[43] which are part of the BOINC:

- **Transitioner** - controls the state transitions of workunits and their results in order to keep the database synchronized.
- **Scheduler** - is a CGI [173] application running over the HTTP server. It handles the requests of all clients. The Scheduler communicates with the BOINC Client (see Section 3.7.9) using the XML-based BOINC scheduling server protocol[39].
- **Feeder** - allocates blocks of shared memory for saving all workunit-related data. It maintains a cache of jobs for the Scheduler.
- **File deleter** - deletes all unnecessary files remaining from older workunits.

3.7.6 WebAdmin

The *WebAdmin* is a web application for remote management of Fitcrack. It consists of two parts: frontend and backend connected interconnected via a REST API. The *frontend*, written in *Vue.js*, provides a graphical user interface for direct interaction with the personnel authorized to operate the system. Figure 3.9 shows a screenshot of the frontend user interface. The backend, written in Python 3, is based on Flask[44] microframework and communicates with Apache or NGINX HTTP server using the Web Server Gateway Interface (WSGI). The backend operates a MySQL database, which serves as a storage facility for all cracking-related data. Note, a complete user guide with screenshots is available on Fitcrack GitHub pages[45].

Features of WebAdmin

The WebAdmin consists of multiple sections that control various functions of the system. For each one, the frontend offers graphical components for user interaction, while the backend provides a series of endpoints that communicate with the database and implement the underlying operations. Fitcrack WebAdmin contains the following sections:

- **Job Creator** - provides a wizard for creating new jobs. Figure 3.10 shows how the user interface works. At first, the user enters the input hashes. For this purpose, the WebAdmin provides three options: a) manual entry, b) uploading an existing hashlist, or c) automated extraction from an encrypted file. Fitcrack can extract hashes from Microsoft Office and PDF documents, and ZIP, RAR, or 7z archives. The second step is the selection of an attack mode and attack options. While in the Hashtopolis tool, the attack settings need to be specified manually as hashcat's command line

[43] https://boinc.berkeley.edu/trac/wiki/BackendPrograms
[44] http://flask.pocoo.org/
[45] https://nesfit.github.io/fitcrack/#/

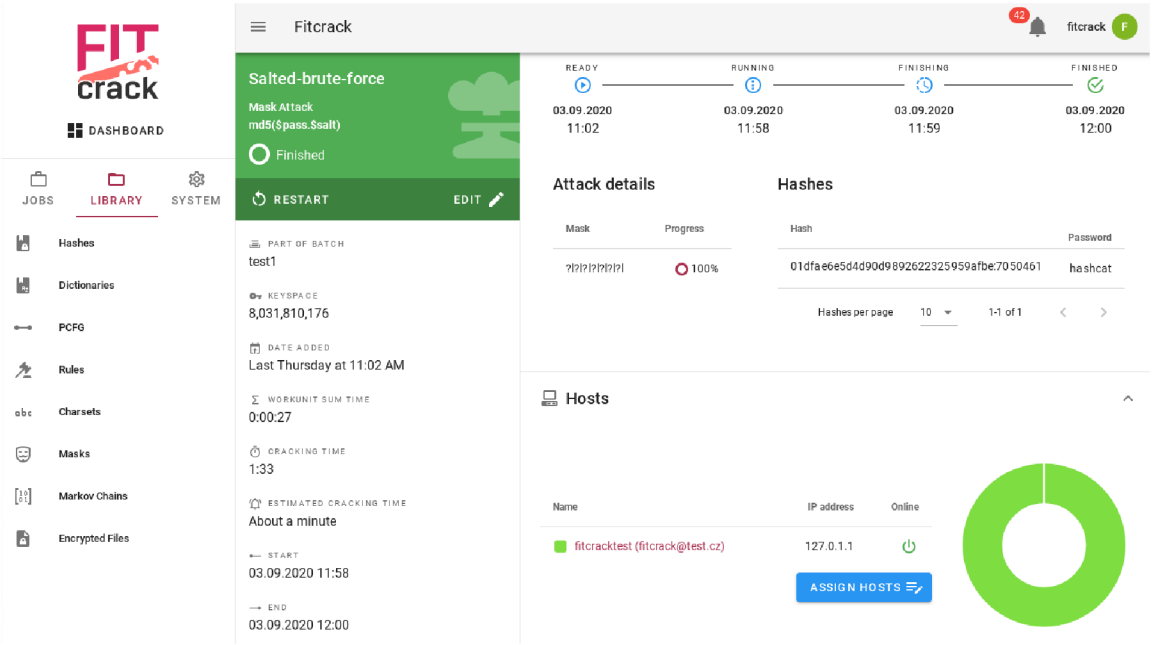


Figure 3.9: The interface of the Fitcrack WebAdmin

arguments, Fitcrack provides a unique set of configurable settings for each of the seven attack modes (see Section 3.8). The settings cover the selection of wordlists for dictionary-based attacks, password-mangling rules, masks for brute-force and hybrid attacks, custom character sets, Markov model parameters, and much more. As the user creates the job, the WebAdmin in real-time notifies the user about the current keyspace and estimated worst-case cracking time. Once the attack is configured, the user assigns one or more hosts to perform the attack. The final step allows specifying the desired workunit cracking time, choosing between an immediate or delayed start, and setting the job's deadline.

- **Job Management** - allows the user to observe, operate, and modify cracking jobs. For each job, the user sees the current progress and various statistics. Fitcrack allows assigning new hosts to already running jobs, changing the job's deadline, or displaying detailed information about individual workunits.

- **Job Batches** - represent a way of stacking multiple jobs together so that the user can control them as an individual job. A *batch* resembles a queue that starts with the first assigned job and continues with the others. Every time a job finishes, the next one starts automatically. Moreover, the WebAdmin displays batch-wide statistics like host contribution or workload distribution.

- **Job Bins** - help organize cracking jobs. The user may assign each job to one or more bins. For instance, the bins may refer to individual cases in terms of forensic investigation. The WebAdmin also allows creating a batch from a bin. Moreover, the entire bin can be exported for server-to-server data transfer.

- **Job Templates** - allow saving attack configuration for later use. Instead of creating a job from scratch, the user may select an existing template.

- **Hashes** - this section represents a system-wide hashlist. Each record contains the algorithm type, hash, timestamp, plaintext password (for already-cracked hashes), and a link to the corresponding job. This hashlist also serves as a lookup table. The lookup is conducted at job creation time. Users are notified if they attempt to create a job with hash that was already cracked.

- **Dictionaries** - provide an interface to manage and add password dictionaries. They serve for wordlist-based attacks: dictionary, combination, hybrid, and PRINCE (see Section 3.8) and optionally to create PCFGs or Markov chains [136]. Fitcrack supports three ways of adding new dictionaries: a) importing directly from the server; b) uploading new via HTTP; c) uploading using SFTP/SCP, if configured.

- **PCFG** - contains the repository of probabilistic context-free grammars [213, 80, 82, 85] for PCFG attacks (see Section 3.8.5). For each grammar, the system displays the achievable keyspace of all possible candidate passwords. The user may also browse all rewrite rules and their probabilities. To add a new grammar, the user may either upload an archive with existing rewrite rules or choose a training dictionary, and the system creates the grammar automatically.

- **Rules** - this section allows the user to manage **.rule* files with the password-mangling rules (see Section 3.8.1). The rules describe string transformations and allow to extend the number of candidate passwords for dictionary-based attacks.

- **Charsets** - extend the classic substitute symbols in password masks. The brute-force attack mode allows the user to utilize up to four user-defined character sets. This section allows the user to browse existing and add new ones.

- **Masks** - can be saved as text files and managed using this section. When users create new jobs, they may load the already-saved masks instead of manual typing.

- **Markov Chains** - are stochastic models that satisfy the Markov property [136]. They define the character creation order in the brute-force attack (see Section 3.8.3). For this purpose, the system uses **.hcstat2* files with per-position character statistics. This section allows the user to add a **.hcstat2* file either by uploading an existing one, or by generating a new one. The second option stands for an automated processing of a password dictionary using the *hcstatgen* tool.

- **Encrypted Files** - this section enlists the names of encrypted files that users uploaded as job input. Each record displays file name and the extracted hash.

- **Hosts** - this section contains an overview of all hosts connected to the system. For each host, the user may see its hardware and OS specifications. The WebAdmin also displays all jobs and workunits assigned to the host.

- **User Management** - allows to create, modify, and delete user accounts. Each account has assigned a role that defines permissions.

- **Server Monitor** - provides an overview of the entire system. It displays the status of all server daemons and the utilization of the server's resources.

- **Settings** - this section contains system-wide settings from appearance through benchmarking to advanced configuration of the scheduling algorithm (see Section 3.6.5).

- **Data Transfer** - this feature allows transferring jobs between two servers. The user may assign one or more jobs to an export. From these jobs the server creates a package with all necessary input data. The package can then be imported to another Fitcrack server.

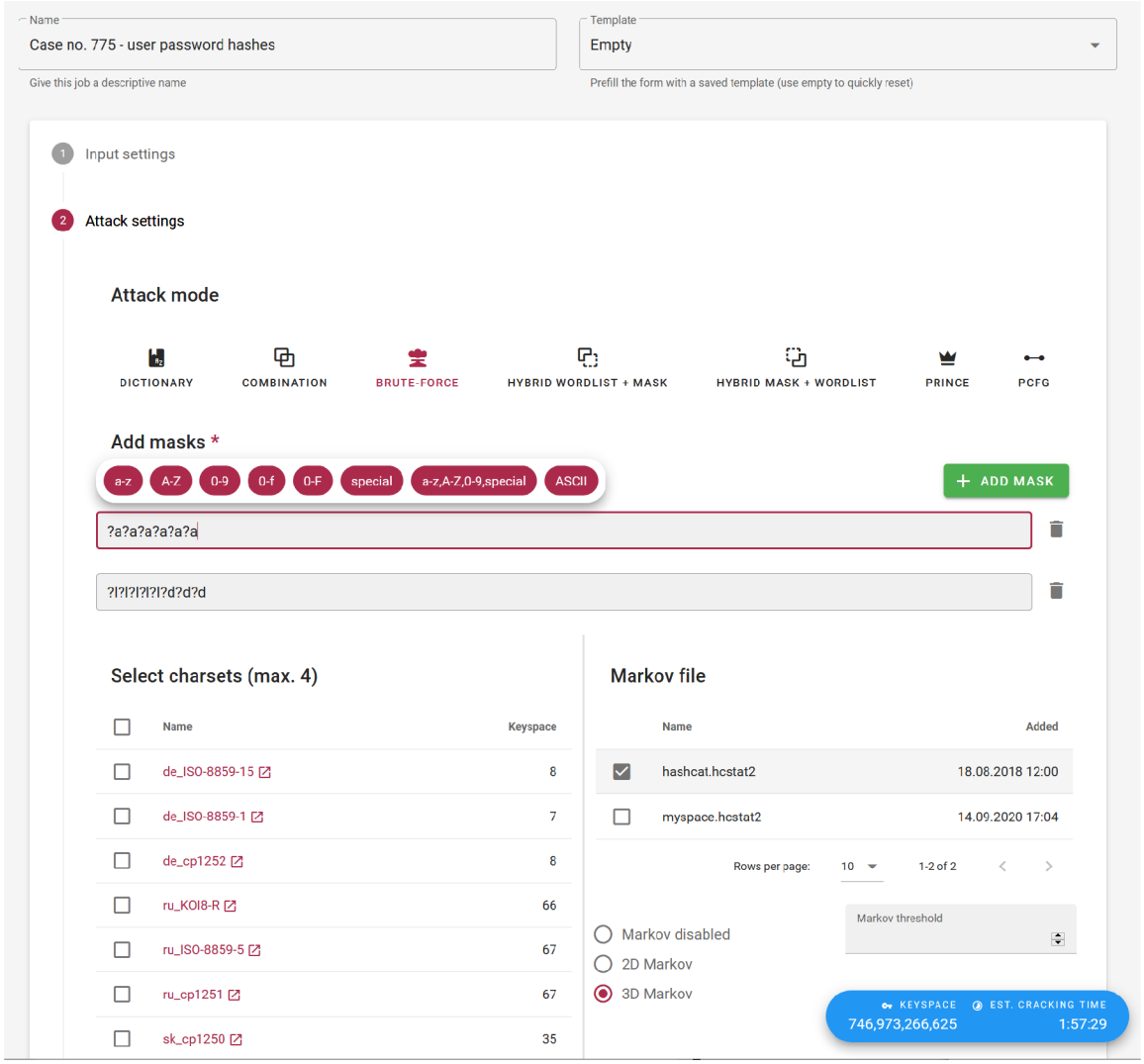


Figure 3.10: Job creation in Fitcrack WebAdmin

Utilities Used by WebAdmin

For some operations, the WebAdmin backend uses a set of external utilities:

- **Hashcat** is used on the server as well. The backend uses it to verify the format of input hashes and to calculate the keyspace of masks. This is important since hashcat's keyspace may not always correspond to the actual number of candidate passwords, as discussed in Section 3.6.4.

- **XtoHashcat** is a tool created for Fitcrack to detect the format of input media and automatically extract all necessary metadata, including cryptographic hashes, salts,

etc. For the media formats, where it is possible (e.g., ZIP and RAR archives, or Office documents), it detects the signature and contents of the file and calls one of the existing scraper scripts (e.g., office2hashcat.py) which extracts the hash.

- **Hcstat2gen** from the *hashcat-utils*[46] repository is used for generating **.hcstat2* files from existing password dictionaries. The files contains character statistics used for Markov-based password guessing in brute-force attacks, as described in Section 3.8.3.
- **Princeprocessor** on the server is used for calculating the keyspace of PRINCE attacks, described in Section 3.8.6.
- **PCFG Trainer** is a tool from Matt Weir that server for creating probabilistic context-free grammars [213, 211] from password dictionaries. The grammars are used in PCFG attacks, described in Section 3.8.5.

3.7.7 PCFG Monitor and PCFG Manager

For PCFG attacks, Fitcrack uses the distributed PCFG Manager proposed in Chapter 4. For each running attack, the server employs a single instance of the PCFG Manager server that creates preterminal structures [213] from the desired grammar. As described in Section 3.8.5, each workunit contains a chunk of these structures. Each assigned host runs the PCFG Manager client as an external password generator. The client creates the candidate passwords from the obtained preterminal structures. For the management of server instances, Fitcrack uses a daemon called *PCFG Monitor*. This tool ensures there is always a running instance of the PCFG Manager server for each running PCFG attack job.

3.7.8 MySQL Database

The MySQL database serves as the primary storage facility for all essential system data. The 2020's version 2.3.0 of Fitcrack uses over 70 different tables. About half consists of the BOINC framework's internal tables for storing data about hosts, client-side applications, data templates, and many others. The rest are tables of Fitcrack with all cracking-related data like jobs, workunit, masks, dictionaries, and others. The detailed specification of the database structure is available in the related technical report [87].

3.7.9 BOINC Client

The *BOINC Client*, also referred to as a *core client*, is an application that handles the communication between the client and the server. It is the only application that needs to be installed manually to a newly-connected host. The rest of the software is downloaded automatically from the server. Using the BOINC Scheduling server protocol[39], the client actively asks the server for work. Once a workunit assignment is received, it downloads all necessary input and output data. Besides that, the client also handles downloading and updating of all executable binaries required: Runner, hashcat, and all hashcat's OpenCL kernels and files that are needed. Depending on how the BOINC client is installed, it may run: a) in the background as a daemon; or b) start when an individual user logs in and stop when the user logs out.

[46] https://hashcat.net/wiki/doku.php?id=hashcat_utils

3.7.10 BOINC Manager

The *BOINC Manager* is an optional part of the client. It provides a graphical user interface for the administration of the core client. It allows the user to choose a project server, review progress on workunits, and configure various client settings. In BOINC Manager, the user can set "when to compute" by defining certain conditions, including times and days of the week, limits on CPU, memory, disk usage, etc.

3.7.11 Runner

The *Runner* is a wrapper of hashcat responsible for processing workunits. It communicates with the rest of the system using the BOINC API[47]. The Runner provides an abstraction layer for all Fitcrack workunits. It takes care of benchmarking as well as the regular cracking tasks. After each cracking session, it creates the report for the Fitcrack server. The Runner can optionally use a local configuration file, where the user can specify which OpenCL devices to use for computation, their workload profile, etc. The Runner also reports partial workunit progress using the Trickle messages processed by the server's Trickler daemon, described in Section 3.7.4.

3.7.12 Hashcat

Hashcat tool is the cracking engine of the Fitcrack system. It employs various OpenCL kernels that implement a GPGPU-based cracking of more than 300 different cryptographic algorithms supported by Fitcrack. Depending on the attack mode, it either uses an internal password generator or reads the candidate passwords from the standard input. From each candidate password, it calculates the cryptographic hash and compares it with the input hashlist. The hashcat is controlled entirely by the Runner application. In case of failure (e.g., GPU overheating, computation error), the Runner generates a report for the server. For more about hashcat, see Section 2.4.4.

3.7.13 Princeprocessor

Princeprocessor[48] is a standalone password generator created by Jens Steube, the author of hashcat. Fitcrack utilizes it for PRINCE attacks. The princeprocessor is located on both the client and server sides. On the server, the WebAdmin uses it to calculate keyspace. On hosts, the Runner uses it as an external password generator.

[47] https://boinc.berkeley.edu/trac/wiki/BasicApi
[48] https://github.com/hashcat/princeprocessor

3.8 Attack Modes and Proposed Distribution Strategies

As introduced in Section 2.2, an *attack mode* represents how candidate passwords are created. The two most commonly known attack modes are a dictionary attack and a brute-force attack. Other modes are mostly enhanced derivatives of these two attacks. Some advanced techniques employ the use of probability to guess passwords more precisely. For instance, by generating character sequences that resemble words from a given natural language.

The arsenal of available attack modes depends on the concrete cracking tool. While there is no unified naming convention, each software uses its unique terminology. Sometimes, such a situation may be confusing for a user. For example, a "hybrid attack" in Elcomoft tools [61] is a synonym for a wordlist mode with password-mangling rules in hashcat [192] or John the Ripper (JtR) [159]. The same term "hybrid attack mode" does exist in hashcat but with a completely different meaning, concretely, as a conjunction of a brute-force attack with a dictionary attack [192]. In addition to the integrated password generating mechanisms, some tools can read candidate passwords directly from standard input. This option allows the user to employ an external password generator and perform attacks that are not natively supported by the tool. In hashcat, this regime is called "stdin" mode [192]. JtR uses the term "external mode" [159].

In this section, I describe the principles and application of the attack modes included in the Fitcrack system [87]. Those include all attack modes provided by the hashcat tool plus a PCFG attack and a PRINCE attack. Whereas the described attacks are known, available tools like *hashcat* or *princeprocessor* exist only as single-machine solutions. Therefore, my contribution lies in the utilization of these methods in a distributed password cracking network. For each attack mode, I discuss possible strategies for work distribution and explain which one is used in Fitcrack and why. By *distribution strategy*, I concretely mean:

- How a password cracking job is decomposed into workunits. What data does the server send within each workunit.
- How hosts handle the workunits and how they control the hashcat.

For the design of the strategies, I focus on the following aspects:

- **Correctness** – The system must generate all candidate passwords that can be created with the attack's configuration. All generated passwords must be eventually verified. This condition is strict and must be met.
- **Precision** – The system should be able to specify workunit sizes as precisely as possible, ideally, in the units of individual passwords. This is crucial for the adaptive scheduling algorithm, described in Section 3.6.5.
- **Sensible network utilization** – With each workunit, the system should only transfer data that is necessary.
- **Server-friendly computing** – Complex computing operations should be performed by hosts, not the server. High utilization of server processors and memory is undesirable. Creating large temporary files on the server is also not optimal.
- **Fast start** – Ideally, the attack should start immediately at the moment the user launches it. Initial delays for precomputing, etc., are unwanted.

3.8.1 Dictionary Attack

A *dictionary attack*, also referred to as a *wordlist attack* or a *straight attack*, uses a text file called the *password dictionary*. The dictionary contains candidate passwords, each placed on a separate line. The cracking tool (hashcat) successively reads the candidate passwords, calculates their hashes, and compares the results with the input hashes, i.e., those we are trying to crack.

Such dictionaries may contain words from a native language or real passwords obtained from various web service security leaks[49]. One of the biggest well-known leaked dictionaries is *rockyou.txt* containing over 15 million passwords. The dataset origins from the end of 2009 when user account information from the RockYou portal leaked due to an attack[50].

Fitcrack supports the use of one or multiple password dictionaries. If more than a single dictionary is in use, the system processes them one by one sequentially. Once we get through all passwords from one dictionary, the processing of the next one begins. From the mathematical perspective, we can consider each dictionary as an ordered set D of strings, where the order is defined by the arrangement of passwords in the dictionary. For n password dictionaries, the keyspace k can be calculated as the sum of their cardinalities:

$$k = \sum_{i=1}^{n} |D_i|$$

where D_i is the i-th used dictionary.

Password-mangling Rules

The attack can be enhanced by the use of *password-mangling rules*. The technique was first introduced in John the Ripper tool, and further extended in hashcat. Password-mangling rules define various modifications of candidate passwords. Such alterations include replacing and swapping of characters and substrings, password truncation, padding, etc. Hashcat currently supports over 70[51] different rules. Table 3.6 illustrates their practical use on a few examples.

To use password-mangling rules, the user needs to specify a text file called *ruleset*. Several pre-defined rulesets are also present in hashcat's repository. Each line of the file contains one or more mangling rules separated by whitespace. Rulesets may also contain comments. The presence of multiple rules on a single line means that they should be used all together in a single mangling step.

The rules are applied to all candidate passwords in the following way: the first candidate password is modified by the rules on the first line of the ruleset; the result is used. Subsequently, the original password is modified by the rules on the next line of the ruleset. Eventually, the entire ruleset is processed. The number of lines in the ruleset signifies how many mangling steps will be performed on each password. The same password-mangling principle is applied to the next candidate password until we eventually reach the end of the password dictionary.

Using rules can rapidly enhance the repertoire of password guesses and provide a higher chance for success. On the other hand, the total keyspace of the job is multiplied by the number of rules. This is because every rule from the rule file is applied to each dictionary

[49] https://wiki.skullsecurity.org/Passwords
[50] https://techcrunch.com/2009/12/14/rockyou-hack-security-myspace-facebook-passwords/
[51] https://hashcat.net/wiki/doku.php?id=rule_based_attack

Rule	Description	Input	Output
l	Converts A–Z to lowercase	p@SSw0rd	p@ssw0rd
u	Converts a–z to uppercase	p@SSw0rd	P@SSW0RD
C	Uppercases first letter, lowercases rest	p@SSw0rd	P@ssw0rd
t	Makes lowercase uppercase and vice versa	p@SSw0rd	P@ssW0RD
r	Reverses all characters	p@SSw0rd	dr0wSS@p
o2X	Overwrites character at position 2 with X	p@SSw0rd	pXSSw0rd
$!	Appends character "!" to the end	p@SSw0rd	p@SSw0rd!
{	Rotates left by one position	p@SSw0rd	@SSw0rp
]	Deletes the last character	p@SSw0rd	p@SSw0r
k	Swaps last two characters	p@SSw0rd	p@SSw0dr
z2	Duplicates first character	p@SSw0rd	pp@SSw0rd

Table 3.6: Examples of password-mangling rules and their application

password. The total keyspace k is calculated as the sum of dictionary keyspaces multiplied by the number of rules in the rule file:

$$k = \sum_{i=1}^{n} (r * |D_i|)$$

where r is the number of lines in the ruleset.

Distribution Strategies for Dictionary Attacks

If the dictionary is pre-loaded on all nodes, the workload distribution is possible by setting different offset and guess limits to different nodes. An alternative may be a dictionary accessible via a shared network drive. Such a solution may work for HPC clusters with nodes interconnected by high-speed links. However, this is not always the case. In general, it is necessary to distribute the password candidates from the server to clients, i.e., the computing nodes. Unfortunately, this effort has significant overhead, and for less-complex hash algorithms could lead to an inefficient distributed attack [86, 84].

Hashtopolis tool lets the user specify files that are necessary for solving the given task. Before the attack starts, each client checks the presence of every file. If it does not exist locally, the client downloads it from the server via HTTP. The advantage is a simple and straightforward implementation. The main drawback is the time and space overhead since all nodes download and store all passwords, even if they need only a tiny portion. While there is no broadcast in HTTP, the server needs to send the same file multiple times using different connections. As shown in the experiments (see Section 3.9), for attacks with larger dictionaries, the overhead is enormous, and the scalability is very limited. There are use cases where this strategy may be beneficial. For instance, if the dictionaries are pre-loaded on all nodes or saved on network storage accessible via high-speed links in a computer cluster. But these are exceptions, and the strategy is not optimal in general.

To eliminate the overhead, yet make workunit size adjustments possible, Fitcrack uses a different strategy. For each host and each workunit, it only distributes a fragment of the original dictionary. The size of the fragment depends on the host's current computing power. The bigger fragment we create, the higher is the keyspace of the workunit. Moreover, the fragment size is not fixed and may vary in time, reflecting each hosts' performance changes.

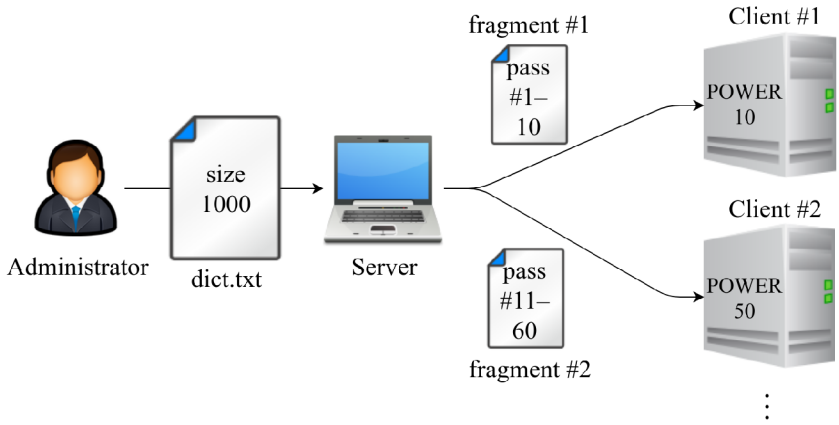


Figure 3.11: Example of dictionary attack distribution

With each workunit, hashcat is started normally in the wordlist attack mode and processes the entire fragment. A simplified scheme of this strategy is shown in Figure 3.11.

In this example, we have a dictionary with 1000 different candidate passwords. The computing network has multiple hosts. Client #1 can verify ten passwords per time unit (defined by the administrator), while Client #2 has five times higher performance. Hence, in the initial workunit, Client #1 receives a fragment with the first 10 passwords of the dictionary, while Client #2 gets the next 50 passwords. Both nodes shall process the assigned password in the same time interval.

Another challenge is the use of password-mangling rules. Its developers proclaim, hashcat has the "world's first and only in-kernel rule engine." On the one hand, this property brings significantly higher performance in contrast with other tool. On the other hand, it requires a modification of the distribution strategy. The use of rules increase the actual number of password guesses, but hashcat applies them in the modifier loop, and therefore hashcat's keyspace remains the same as if no rules were used. For a single keyspace unit, hashcat uses one word from the dictionary and consequently applies all existing rules to that word.

As mentioned in Section 3.6.4, Fitcrack distinguishes between hashcat's keyspace that is used for setting the program parameters and the actual keyspace that is used for calculating the size of a workunit for a concrete client. If password-mangling rules are applied, Fitcrack multiplies the real keyspace by the number of rules used. The estimation of computing time thus uses the actual number of password guesses. Such a correction prevents creating unequally large workunits.

For example, a host can verify 1,000 hashes of a given algorithm per second using hashcat's dictionary attack mode. The plan is to create a workunit for an hour. Therefore, Fitcrack would generate a workunit of keyspace 3,600,000. A user, however, decided to use password-mangling rules and selected a ruleset with 100 rules. From each dictionary word, hashcat creates 100 passwords. The hash algorithm thus needs to be calculated 100 times. Without the correction, resolving the workunit would require more than 4 days. With the correction applied, the final keyspace of the workunit is downgraded to 36,000, which is optimal for an hour.

3.8.2 Combination Attack

A *combination attack*, also referred to as a *combinator attack*, uses two separate password dictionaries: a *left* dictionary, and a *right* dictionary. Candidate passwords are crafted using a string concatenation: passwords from the left dictionary are extended by passwords from the right one. The goal is to verify combinations of all passwords in the two input dictionaries. An example of a combination attack is shown in Figure 3.12. Let D_1 be the left dictionary, and D_2 the right dictionary. The keyspace k can be calculated as:

$$k = |D_1| * |D_2|.$$

The combination attack in hashcat supports a single password-mangling rule for the left and right sides. Their syntax and semantics are the same as with the classic dictionary attack. Applying multiple rules is not supported, and thus the rules do not affect the resulting keyspace.

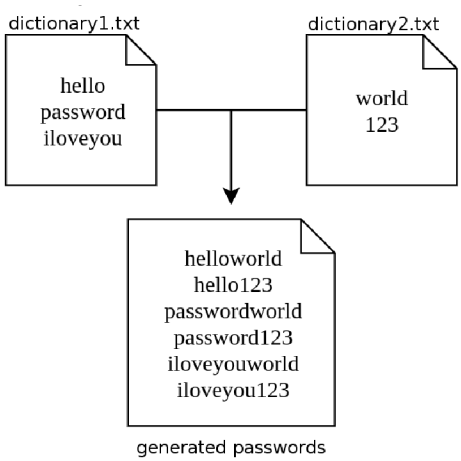


Figure 3.12: An illustration of a combination attack

Advanced Combination Attacks

Note, in addition to the basic combination attack, enhanced alternatives exist. The *hashcat-utils* toolkit contains a *combinator3* utility that can combine three dictionaries. If necessary, it can serve as an external password generator. For chaining multiple dictionary words, there is also an advanced combination attack called PRINCE that is decribed in Section 3.8.6.

Distribution Strategies for Combination Attacks

For combination attacks, hashcat's keyspace calculation does not consider the second dictionary. When asked to verify a single password using the combination attack, hashcat actually verifies $1 \times n$ passwords. For the example dictionaries in Figure 3.12, hashcat with `--limit=1` parameter generates two passwords instead of one: "helloworld" and "hello123". This property prevents creating workunits of the desired size. If the second dictionary contained hundreds or thousands of passwords, such behavior would represent a serious problem. Attacks with large dictionaries would be virtually uncontrollable.

Algorithm 4: Limiting the combination attack to the desired keyspace

Input: $k_{desired}, k_{left}, k_{right}, i_{left}, i_{right}$
Output: $i_{left}, i_{right},$

1: **if** $i_{left} > 0$ **then**
2: send i_{right}th password from the right dictionary
3: send `--skip` parameter with i_{left} as the value
4: **if** $k_{desired} < k_{left} - i_{left}$ **then**
5: send `--limit` parameter with $k_{desired}$ as the value
6: $i_{left} = i_{left} + k_{desired}$
7: **else**
8: $i_{left} = 0$
9: $i_{right} \mathrel{+}= 1$
10: **else if** $k_{desired} > k_{left}/2$ **then**
11: send $k_{desired}/k_{left}$ passwords from the right dictionary, starting with the i_{right}th.
12: $i_{right} = i_{right} + k_{desired}/k_{left}$
13: **else**
14: send i_{right}th password from the right dictionary
15: send `--limit` parameter with $k_{desired}$ as the value
16: $i_{left} = k_{desired}$

A naive solution for this problem is to generate all possible combinations, save them into a single dictionary, and perform a classic dictionary attack. I do not consider this an ideal way for a couple of reasons. First, generating such a emporary dictionary requires a noticeable amount of processor time and disk space. The larger the input dictionaries we have, the higher is the initial overhead. Second, it increases the amount of data that needs to be transferred over the network. Let m be the size of the left dictionary and n the size of the right dictionary. The space complexity in the sense of the transmitted passwords changes from linear, ideally $m + n$ passwords, to polynomial, $m \times n$. This may also rapidly increase the time required for transferring the data to all hosts.

To deal with this issue, Fitcrack uses a different solution. With the first workunit of a job, the entire left dictionary is distributed to all computing nodes. With each next workunit, it only distributes a fragment of the second dictionary. By modifying the size of this fragment, Fitcrack can control the number of right-hand strings applied within the workunit. Moreover, it may utilize the "skip/limit" arguments to reduce the number of left-hand strings if necessary. This strategy allows precise control of workunit sizes and keeps the linear complexity of data transfer over the entire attack.

Algorithm 4 describes the process of calculating workunit size. In a nutshell, it decides how many passwords from the right dictionary to send and whether to use the `--skip` and `--limit` parameters. The algorithm uses five inputs: $k_{desired}$ is the desired keyspace of the workunit, k_{left} and k_{right} is the keyspace of the left and the right dictionary, respectively. To indicate the current position in both dictionaries, it uses two indexes: i_{left} and i_{right}. The condition o line 10 is a heuristic that adds some tolerance to prevent over-fragmenting. For example, if the desired keyspace is 100 and there are 101 passwords remaining, it assigns the host all 101 in a single workunit instead of creating two workunits with 100 and 1 password. Essentially, there are three possible situations:

- The left dictionary is partially processed ($i_{left} > 0$). The algorithm uses the `--skip` parameter to skip already-processed passwords.
- The left dictionary is small enough to be processed entirely. Only the right dictionary is potentially fragmented.
- The left dictionary is too big for the desired keyspace. Then the algorithm uses the `--limit` parameter to specify the number of passwords to process from the left dictionary. From the right dictionary, it takes only a single password.

See that the proposed strategy iterates through the left dictionary over and over until all passwords from the right one are processed. With each workunit, the left dictionary is either processed entirely or partially, but when we reach its end, we start over by setting i_{left} to 0.

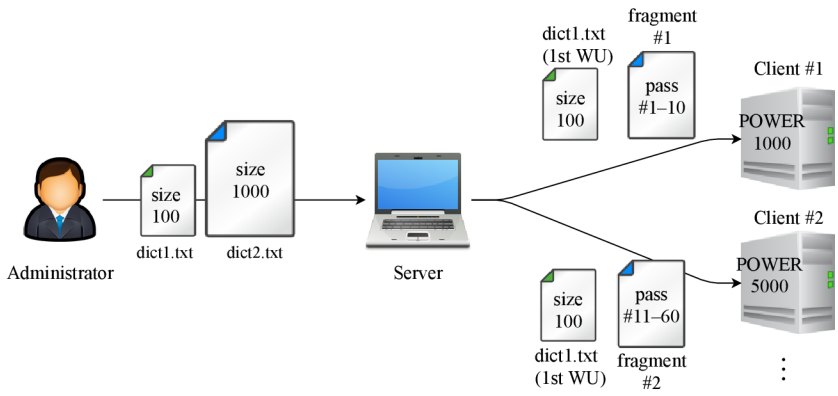


Figure 3.13: An example of combination attack distribution

Figure 3.13 illustrates an example of distribution with the proposed strategy. In this example, we have two dictionaries, the left dictionary contains 100 passwords, while the right one contains 1000 passwords. The first one is distributed to all nodes, while for the second one, Fitcrack uses fragmentation. Client #1 can verify 1000 passwords per time unit, and thus with the first chunk, the client receives the first 10 passwords from the second dictionary since $10 \cdot 100 = 1000$. Client #2 has five times higher performance and can verify 500 passwords per time unit. Therefore, it receives the following 50 passwords since $50 \cdot 100 = 5000$. In this case, everything was solved by fragmenting just the right dictionary. The `--skip` and `--limit` parameters were not necessary.

In contrast, assume the same dictionaries but the desired keyspace of 40 passwords. This is less than half of the left dictionary's keyspace, so that the algorithm: a) sends only a single-password fragment of the right dictionary; b) sets the `--limit 40` hashcat argument; and c) moves the i_{left} to 40. The next workunit is for a more powerful host and should have 1050 passwords. Since the left dictionary is fragmented, the host receives one password from the right dictionary along with hashcat argument `--skip 40`, the i_{right} increases to 1, and i_{left} is reset to 0. Assume the following workunit should contain 1020 passwords. The left dictionary is not fragmented so that this host can receive multiple passwords from the right dictionary (for a total of 1000 candidate passwords). The host gets the second through eleventh passwords from the right dictionary, and the i_{right} is set to 11. This way, Fitcrack eventually verifies the combinations of every password from the left dictionary with every password from the right dictionary.

3.8.3 Brute-force Attack

A *brute-force* attack is an exhaustive search for correct password(s) trying every possible password candidate that can be made from given characters.

Incremental Brute-force Attack

An incremental attack is a classic version of the brute-force attack with three parameters: the *minimal password length*, the *maximal password length*, and the *alphabet*, also referred to as the *character set*, or *charset*. The alphabet is an ordered set of characters that are used for generating candidate passwords. The order may be alphabetical, based on an ASCII value, or any other ordering. The order of password candidates may also be implementation-specific. The most straightforward way is to first generate all possible passwords for the minimal length and then consequently proceed through longer ones. Listings 3.1 to 3.3 show a classic incremental brute-force attack using lowercase Latin letters in alphabetical order with passwords length from 5 to 7 characters.

```
aaaaa
aaaab
.....
aaaaz
aaaba
aaabb
.....
aaazz
aabaa
aabab
.....
zzzzz
```

Listing 3.1: Length 5

```
aaaaaa
aaaaab
......
aaaaaz
aaaaba
aaaabb
......
aaaazz
aaabaa
aaabab
......
zzzzzz
```

Listing 3.2: Length 6

```
aaaaaaa
aaaaaab
.......
aaaaaaz
aaaaaba
aaaaabb
.......
aaaaazz
aaaabaa
aaaabab
.......
zzzzzzz
```

Listing 3.3: Length 7

The name "incremental" comes from the behavior of password guessing. In every step, we increment the value of a character at the last position. After trying everything from "a" to "z", we also increment the character at the last but one position and start over. Generating passwords in the incremental mode resembles incrementing numbers in a numeral set with symbols from the alphabet. The above-shown example is only one of the possible implementations. We may also start from the first character, or use a completely different logic.

The incremental attack mode is supported by many cracking tools, including John the Ripper tool, or Wrathion - the predecessor of Fitcrack that I used in my early research [83]. As mentioned above, different tools use different modifications of the password-guessing algorithm. More advanced techniques involve Markovian models (see Section 4.2.2) and the use of probability to generate certain sequences of characters first. For example, the Incremental attack mode of John the Ripper tool uses a modified Markov model based on 3-grams [57]. The keyspace k of an incremental attack can be calculated as:

$$k = \sum_{l=min}^{max} |A|^l$$

where min is the minimal password length, max is the maximal password length, and A is the alphabet.

Mask Attack

In hashcat, the brute-force attack is based entirely on password masks. Later versions of John the Ripper contain a "mask attack mode" as well, in addition to the classic incremental mode. Masks are patterns that describe the allowed syntax of candidate passwords, i.e., how candidate passwords "may look like". A user may define one or more masks for the attack. The cracking process then consist of generating every possible sequence of characters upon each mask.

A *password mask* is a template defining allowed characters for each position in the password. Masks have the form of strings containing one or more symbols. A password mask m of length n is defined as:

$$m = s_1 s_2 ... s_n$$

where s_i is the i-th symbol of the mask, and $i \in [1, n]$. Such a mask can be used to generate candidate passwords in the form of $c_1 c_2 ... c_n$ where c_i is the i-th symbol of the candidate password. Obviously, the candidate passwords have the same length n as the mask. For all i, the s_i symbol in the mask is:

- **a concrete character** (c_i) - which is directly used in generated candidate passwords at position i, or
- **a substitute symbol** (S_i) **for a character set** (C_i) - which defines the allowed characters at position i in the generated candidate passwords.

Symbol	Description	Characters in set
`?l`	lowercase Latin letters	`abcdefghijklmnopqrstuvwxyz`
`?u`	uppercase Latin letters	`ABCDEFGHIJKLMNOPQRSTUVWXYZ`
`?d`	digits	`0123456789`
`?s`	special characters	`(space)!"#$%&'()*+,-./ :;<=>?@[\]^_‘{|}~`
`?h`	hexadecimal digits with small letters	`0123456789abcdef`
`?H`	hexadecimal digits with big letters	`0123456789ABCDEF`
`?a`	all standard ASCII characters: `?l`, `?u`, `?d`, `?s`	
`?b`	binary - all bytes of values between `0x00` and `0xFF`	
`?1`	user-defined character set no. 1	
`?2`	user-defined character set no. 2	
`?3`	user-defined character set no. 3	
`?4`	user-defined character set no. 4	

Table 3.7: The substitute symbols and corresponding character sets

A *character set* (or simply *charset*) is an order set of characters. In masks, we use *substitute symbols*, each corresponding to a different character set. Table 3.7 lists the substitute symbols supported by *hashcat* with corresponding character sets. Besides the standard character sets (`?l`, `?u`, `?d`, `?s`, `?h`, `?H`, `?a`, `?b`), hashcat supports up to four user-defined

character sets (`?1`, `?2`, `?3`, `?4`). Custom character sets may contain both ASCII and non-ASCII characters - i.e., may be used in combination with various national encodings.

An example of generating passwords using a mask is illustrated by Figure 3.14. If there are concrete characters in a mask, the same characters at the same positions are used in the generated candidate passwords – i.e., if for all $i \in [1, n]$, if $s_i = c_i$, character c_i is used at the i-th position in all generated passwords. For substitute symbols, all possible characters from corresponding character sets are eventually used. If there is more than one one substitute symbol, candidate passwords are generated as a cartesian product of all used corresponding character sets.

For example, in mask `Hi?u?d?d`, the first two symbols are concrete characters $c_1 = H$ and $c_2 = i$. The rest is made of substitute symbols: $S_3 =$ `?u` which substitutes $C_u = \{A, ..., Z\}$, and $S_4 = S_5 =$ `?d` which substitutes $C_d = \{0, ..., 9\}$. Therefore, the prefix of candidate passwords is fixed (`Hi`), the rest is generated as $C_u \times C_d \times C_d$ or $\{A, ..., Z\} \times \{0, ..., 9\} \times \{0, ..., 9\}$. So that, the mask generates the following candidate passwords: `HiA00`, `HiA01`, ... `HiA09`, `HiA10`, `HiA11`, ... `HiA99`, `HiB00`, `HiB01`, ..., `HiZ99`. In a brute-force attack, the number of possible candidate passwords can be calculated as:

$$k = \prod_{i=1}^{n_s} |C_i|$$

where n_s is the number of substitute symbols in the mask, and C_i is the character set substituted by symbol S_i. For the previous mask `Hi?u?d?d`:

$$k = \prod_{i=1}^{3} |C_i| = |C_u| * |C_d| * |C_d| = 26 * 10 * 10 = 2600$$

we have 2600 possible password candidates.

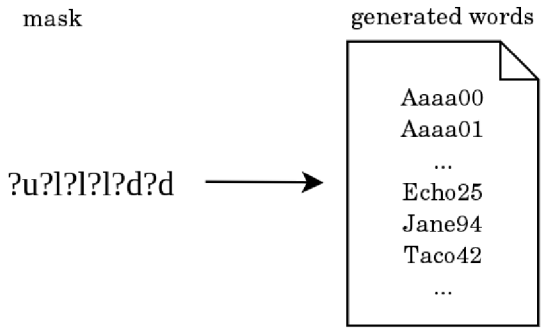


Figure 3.14: Illustration of a brute-force mask attack

Markov Chains

The principles of Markovian models and the history of their use for password cracking are narrowly discussed in Section 4.2.2. This section primarily focuses on the two models that are used in the hashcat tool since it serves at the cracking engine of the Fitcrack system.

In hashcat's brute-force attack mode, the candidate passwords are not generated by the lexicographic order of characters. Instead, it uses an algorithm based on *Markov chains* [136, 169]. The entire idea behind Markov chains is to use knowledge obtained by learning

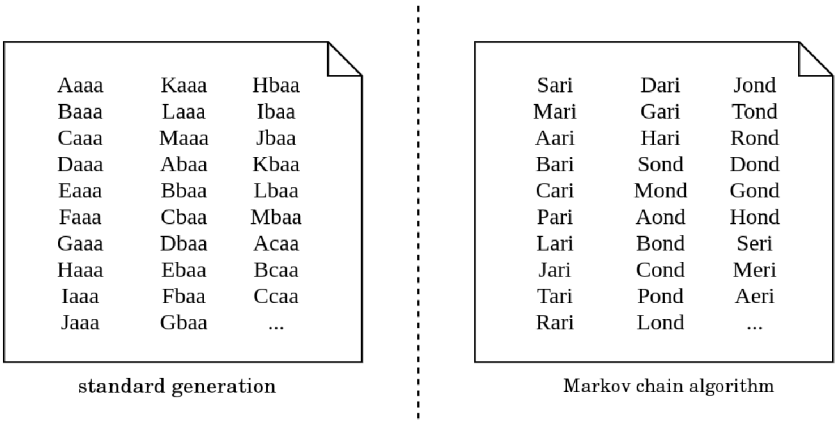


Figure 3.15: Candidate password order using Markov chains

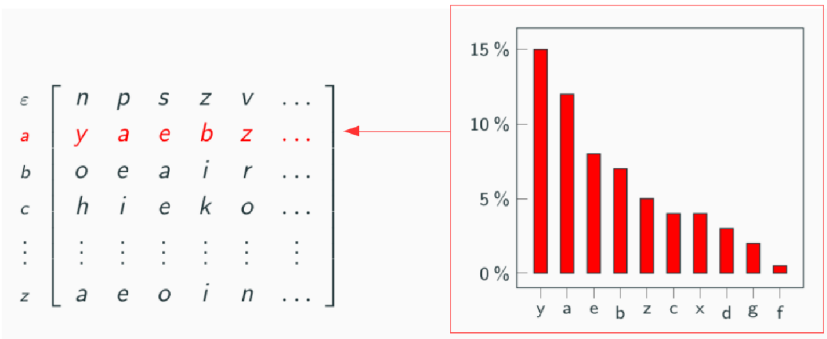


Figure 3.16: Markov chain probability matrix

on existing wordlists to generate more probable sequences of characters first. To illustrate the difference, Figure 3.15 shows an example of candidate passwords generated using the classic incremental approach and Markov chains. Hashcat uses a first-order Markovian model where the choice of a character depends on a previous one. Generating characters is based on conditional probability $P(A|B)$ that character A will follow after character B. The implementation does not calculate the raw probabilities. Instead, it uses a matrix of characters ordered by their probability. The matrix with these statistics is saved inside a *.hcstat* file. Starting from hashcat 4.0.0, hashcat uses[52] LZMA[53] compression and the extension changed from *.hcstat* to *.hcstat2*. With the `--markov-hcstat` option the user can specify what file to use. The default statistics files used for brute-force attack are `hashcat.hcstat` and `hashcat.hcstat2`, respectively.

Figure 3.16 shows an example of a Markov chain matrix. In each row, the matrix lists characters from the character set in order from the most probable, to the least probable. The first row entitled with ε shows characters on the first position in the password. In the example, the most probable character on the first position is "n", the second most probable is "p", etc. The other rows show characters which will most probably succeed after a certain character (entitling the row). In the example, "a" will be most probably followed by "y". The second most probable successor of "a" is "a", the third one is "e", etc.

[52] https://hashcat.net/forum/thread-6965.html
[53] https://www.7-zip.org/sdk.html

$$
\begin{array}{c}
\varepsilon \\ a \\ b \\ c \\ d \\ e \\ \vdots
\end{array}
\left[\begin{array}{ccc|cccc}
b & n & e & g & a & u & \dots \\
d & t & r & n & d & v & \dots \\
e & a & r & u & o & i & \dots \\
k & i & e & o & u & a & \dots \\
o & m & a & y & r & p & \dots \\
d & c & t & z & d & n & \dots \\
\vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \ddots
\end{array}\right]
$$

Figure 3.17: Example of Markov matrix with threshold set to 3

The matrix defines how the candidate passwords are generated. At the first position, characters from ε row are used. The order is defined by position in the matrix. In the matrix from Figure 3.16, the first sequence of candidate passwords would start with letter "n". Once all passwords starting with `n` are generated, the next sequence contains passwords starting with letter "p", etc. For each character "c" generated, the algorithm looks at the row entitled by "c", and the next character will be generated from that row.

In standard case, on each position, all possible characters are used, and the keyspace is calculated as shown in Section 3.8.3. In hashcat, however, it is possible to define a *threshold* value which can be used to limit the depth of character lookup. The threshold says how many characters from each row are used. Naturally, using the treshold affects the keyspace. If threshold is used, the least probable passwords are not generated. If many cases, thresholding can save processor time without bigger influence on success rate [136, 169].

For now, let us ignore the keyspace optimization used by hashcat, described in section 3.6.2 – i.e., assume the keyspace is the actual number of password candidates. Figure 3.17 shows a matrix with threshold set to 3. In case of mask `?l?l?l`, the keyspace would be $26 * 26 * 26 = 17576$, since $|C_l| = 26$. However, which threshold set to 3, the keyspace is $3 * 3 * 3 = 27$, since on each position, only three characters are used.
The candidate passwords for mask `?l?l?l` and threshold 3 are generated in the following order: `bed`, `bec`, `bet`, `bad`, `bat`, `bar`, ... Note that password `bez` is not generated since `z` is on the position 4 in *e*-row, and $4 > 3$. In hashcat, the threshold can be specified using the `--markov-threshold` option. For brute-force attack with Markov chains, hashcat supports two different models:

- **2D Markov model** (*classic*) - uses a classic first-order Markovian model with a single matrix for a character set, and works as described above. The technique is used if hashcat is run with `--markov-classic` option.

- **3D Markov model** (*per-position*) - a modified version that is used by default in hashcat's brute-force attack. It utilizes the idea that character probability is influenced not only by the previously generated character, but also by the position in the password. The model uses multiple matrices, one per each password position. If the first character is generated, the first matrix is used, for the second chracter, the second matrix is used, etc. Such an enhancement makes sense because users often follow specific password-creation patterns, e.g., numbers will more likely be at the end of the password than at the beginning [35, 212].

Distribution Strategies for Brute-force Attacks

The distribution of a brute-force attack highly depends on the implementation of the password guessing algorithm, concretely, how it represents its state. For example, commercial tools from Elcomsoft[54] employ the classic incremental brute-force (see Section 3.8.3) and provide the "Start from" and "End at" columns where a user can specify the range using concrete passwords. John the Ripper, on the other hand, does not track the starting and ending position of password segments at all [47]. In such a case, without modification of the program, the options are very limited[55], e.g., letting different nodes generate passwords of different lengths, etc.

Luckily, hashcat natively supports defining the starting and ending position using the `--skip` and `--limit` parameters. Thus, with each workunit, Fitcrack only distributes the mask, the range of indexes, plus the user-defined character sets if used. This strategy makes a brute-force attack, in contrast with the previously described attacks, very efficient in a distributed environment. The overhead to the attack is minimal since there is no need to transfer strings via the network. Moreover, hashcat's password generating algorithm is highly optimized and partially computed on-GPU.

Paradoxically, the optimizations also introduce other issues that complicate the workload distribution. One of the biggest challenges of distributing the mask attack in hashcat is the way hashcat computes the keyspace of each mask, described in Section 3.6.4. In the brute-force attack mode, part of the mask is processed on GPU and cannot be managed by users. Using command-line arguments, users can only control the part of the mask that is generated in the base loop on a CPU. In other words, the user may specify how many iterations of the base loop are calculated. The rest, processed in the modifier loop on GPU, cannot be changed externally. What specific part of the mask is processed in the base loop depends on multiple factors. Those include the syntax of the mask, the version of hashcat, and even the type of hash algorithm used. Practically, this means that with `--limit 1`, hashcat performs a single base loop iteration but may produce multiple password guesses. The keyspace reported by hashcat is thus always lower or equal to the number of password guesses.

To overcome this obstacle, Fitcrack lets hashcat on the server calculate its optimized keyspace. This value is used to determine the range of allowed password indexes. Moreover, this solution provides forward compatibility with future versions of hashcat. If the keyspace calculation changes for any hash format, Fitcrack will still receive correct values. In addition to hashcat's normalized keyspace, Fitcrack calculates the actual number of password guesses. This value serves for estimation of the cracking time, specifying workunit sizes, and informing the user. Both numbers are calculated before the attack even starts. Dividing the real keyspace by the hashcat's keyspace, we can determine how many real passwords are represented by a single hashcat index. With this knowledge, sending the mask with the corresponding index range to verify is no longer a problem.

For example, assume the cracking of SHA-3-512 hashes with mask `?l?l?l?l?l?l?l?l`. The real keyspace is $26 \cdot 26 \cdot 26 \cdot 26 \cdot 26 \cdot 26 \cdot 26 \cdot 26 = 26^8 = 208,827,064,576$ passwords. In hashcat, the SHA-3-512 algorithm corresponds to hash mode 17600. Running `./hashcat64.bin -m 17600 -a 3 ?l?l?l?l?l?l?l?l --keyspace` returns 11,881,376. Fitcrack calculates $\frac{208,827,064,576}{11,881,376} = 17,576$ so that a single unit of hashcat's keyspace corresponds to 17,576 candidate passwords. Assume a host with GTX 1050 Ti having the brute-force attack

[54] https://www.elcomsoft.com/

[55] https://openwall.info/wiki/john/parallelization

performance of 250 MH/s. A 1-minute workunit thus represents $250,000,000 \cdot 60 = 15,000,000,000$ passwords. Hence, Fitcrack uses $\frac{15,000,000,000}{17,576} \approx 853,437$ as the `--limit` parameter for hashcat. To demonstrate that this principle works, I made a brief live test with by running the following command:

```
time ./hashcat64.bin -m 17600 -a 3 sha3-512.hash ?l?l?l?l?l?l?l?l \
--limit 853437
```

The `-a 3` parameter corresponds to the brute-force attack mode, while the `time` utility is used to measure real time of running. The experiment was conducted on a system with the above mentioned GPU, Intel(R) Core(TM) i7-8700 CPU, and 16 GB RAM. The operating system was Debian GNU Linux 9. The commands produced the following output (the very beginning is omitted):

```
Session..........: hashcat
Status...........: Exhausted
Hash.Type........: SHA3-512
Hash.Target......: 9ece086e9bac491fac5c1d1046ca11d737b92a2b2ebd93f005d...
Time.Started.....: Sun May 10 15:28:11 2020 (1 min, 0 secs)
Time.Estimated...: Sun May 10 15:29:11 2020 (0 secs)
Guess.Mask.......: ?l?l?l?l?l?l?l?l [8]
Guess.Queue......: 1/1 (100.00%)
Speed.#1.........: 248.1 MH/s (6.94ms) @ Accel:32 Loops:16 Thr:640 Vec:1
Recovered........: 0/1 (0.00%) Digests, 0/1 (0.00%) Salts
Progress.........: 15000008712/15000008712 (100.00%)
Rejected.........: 0/15000008712 (0.00%)
Restore.Point....: 853437/11881376 (7.18%)
Restore.Sub.#1...: Salt:0 Amplifier:17568-17576 Iteration:0-16
Candidates.#1....: wzvxrxba -> xqxmrbpe
Hardware.Mon.#1..: Temp: 69c Fan: 49% Util: 91% Core:1733MHz Mem:3504MHz

Started: Sun May 10 15:28:10 2020
Stopped: Sun May 10 15:29:13 2020

real 1m2.491s
user 0m4.364s
sys 0m1.448s
```

As we can see, the cracking session took exactly 1 minute. The remaining 2.491 seconds represented the overhead for the initialization of hashcat and its OpenCL kernels. Depending on a concrete network, there is additional overhead for network communication, BOINC, and other factors. The scheduling algorithm employed in Fitcrack, however, adapts to such impacts without problems since it measures the overall time between assigning each workunit and receiving its result. All additional delays are thus taken into account.

3.8.4 Hybrid Attacks

Hybrid attacks combine the dictionary attack (see Section 3.8.1) with the brute-force attack (see Section 3.8.3). Hashcat supports two variations of hybrid attacks. The first combines a dictionary on the left side with a mask on the right side. The second hybrid attack works the opposite way, with the mask on the left and dictionary on the right side. Both cases are illustrated in Figure 3.18. For the dictionary-based part, passwords are taken from a password dictionary. For the mask-based part, the passwords are generated using the brute-force technique. The generated candidate passwords are created using string concatenation over the two parts. The resulting keyspace is:

$$p = |D| * \prod_{i=1}^{n_s} |C_i|$$

where D is the dictionary used, n_s is the number of substitute symbols in the mask, and C_i is the character set substituted by i-th symbol of the mask. So that, the complexity equals to $m \times n$, where m represents the size of the dictionary while n is the number of passwords generated by the mask.

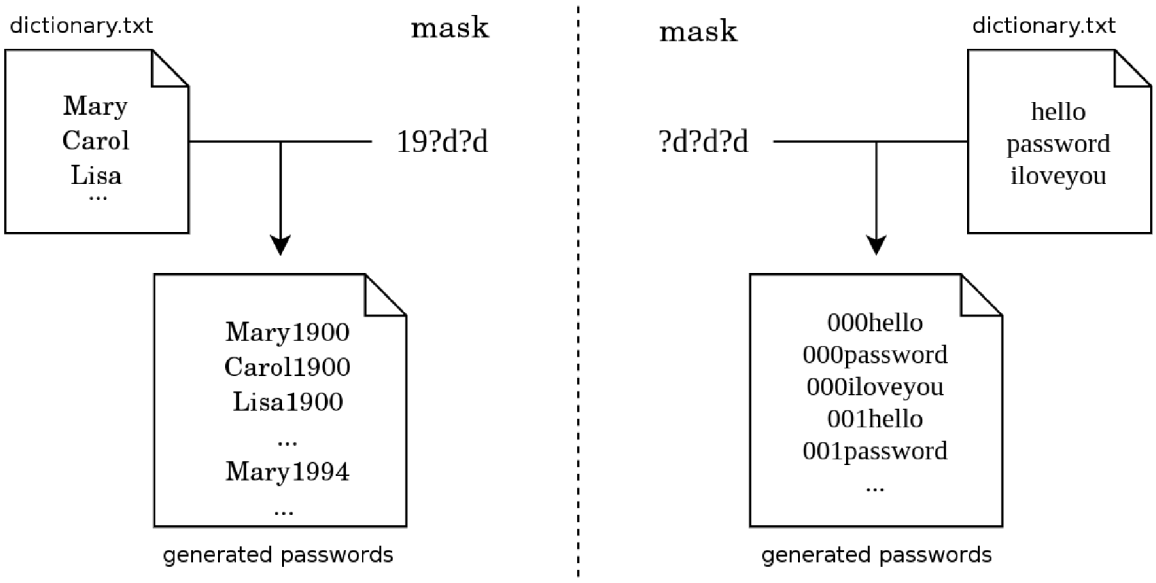


Figure 3.18: The principle of hybrid attacks

SSimilar to the combination attack, hashcat does not allow to control the keyspace of the whole attack. With instructed to verify a single password using `--limit 1`, hashcat checks the combination of one string from the left side and all strings from the right side. How many actual candidate password correspond to a single unit of hashcat's keyspace depends on the variation of the hybrid attack:

- **hybrid wordlist+mask** - a single word from the dictionary concatenated with every possible string created from the mask. The number of actual passwords equals the keyspace of the mask.
- **hybrid mask+wordlist** - a single string generated from the mask concatenated with every word from the dictionary.

Assume the examples from Figure 3.18 and hashcat started with `--skip=0`, `--limit=1`, and `--markov-disable` that disables markov-based guessing (see Section 3.8.3). In the first case, hashcat generates all possible candidate passwords with "Mary" on the left side, i.e., "Mary1900" to "Mary1999". In the second case, hashcat generates the first string from the mask and combines it with all dictionary passwords. Therefore, it generates three passwords: "000hello", "000password", and "000iloveyou".

Distribution Strategies for Hybrid Attacks

Distributing hashcat-based hybrid attacks efficiently and precisely is a true challenge. Similarly to the combination attack, we need to combine every left-hand password with every right-hand one. Nevertheless, hashcat only applies the skip and limit parameters to the left side. If there is a dictionary, the parameters control the number of dictionary passwords. Vice versa, for a mask on the left side, the parameters limit the number of candidate passwords generated from the mask. Sadly, when the mask is on the right, it is always processed entirely. There is no way to generate only part of the mask-created strings. Therefore, the same strategy as with the combination attack cannot be used.

A naive solution is to generate all passwords beforehand and use a classic dictionary attack. This approach allows to precisely control the size of each workunit, but at the cost of efficiency. The overhead for generating and transmitting password guesses would be enormous.

The early hashcat-based versions of Fitcrack used a compromise solution. With the high-performance *maskprocessor*[56] utility, the server created a dictionary of all possible strings from the given masks. Then, the attack was transformed into a combination attack, and the distribution followed the same strategy as proposed in Section 3.8.2. With the two dictionaries, it was possible to control the size of workunits relatively precisely. Figure 3.19 shows an example of the original distribution of hybrid attacks. In the example, we see the server first transforms the mask into a dictionary and then does the fragmenting in the combination-like way. An advantage of the solution was that Fitcrack could the use the left and right password-mangling rules (see Section 3.8.1) in the same way as with the combination attack, although rules are normally not supported in hybrid attacks. The solution worked well with smaller masks. However, with complex high-keyspace masks, the initial overhead and the space requirements were not acceptable.

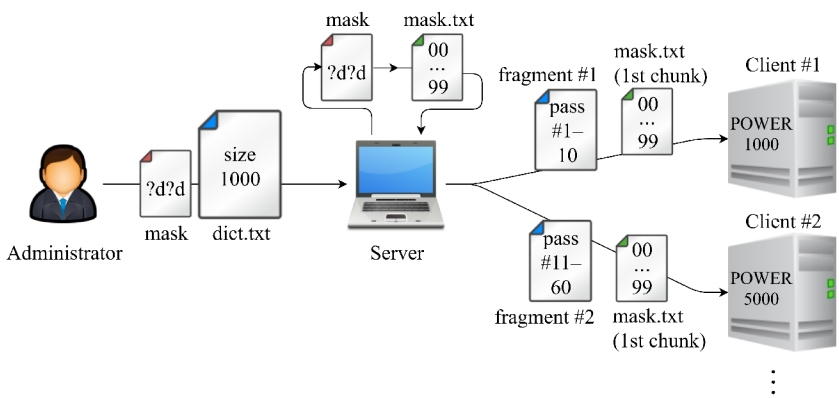


Figure 3.19: The original distribution strategy for hybrid attacks

[56] https://github.com/hashcat/maskprocessor

To allow using more complex masks and eliminate the overhead, starting from version 2.2.0, Fitcrack uses an entirely different strategy. Hashcat runs in the native hybrid attack mode, and there is no need for the maskprocessor utility since no strings are pre-generated anymore. The workunits are created as follows:

- For the **hybrid mask + wordlist** attack, the dictionary on the right side is fragmented in the same manner as in the combination attack. If necessary, the mask is limited using the `--skip` and `--limit` parameters using Algorithm 4. This is entirely safe since, for the hybrid attack mode, hashcat does not use the keyspace optimization described in Section 3.6.4. The limit parameter thus precisely specifies the exact number of strings generated from the mask. An example of the workunit distribution is shown in Figure 3.20.

- For the **hybrid wordlist + mask**, the mask on the right is transformed into multiple masks with lower keyspace using the newly-proposed Algorithm 6. To allow precise control of the keyspace, Fitcrack creates custom character sets on-the-fly using the `GetCharsetSlice()` function defined by Algorithm 5. If necessary, the dictionary on the left is limited using the `--skip` and `--limit` parameters using Algorithm 4. An example of workunit distribution with mask slicing is shown in Figure 3.21.

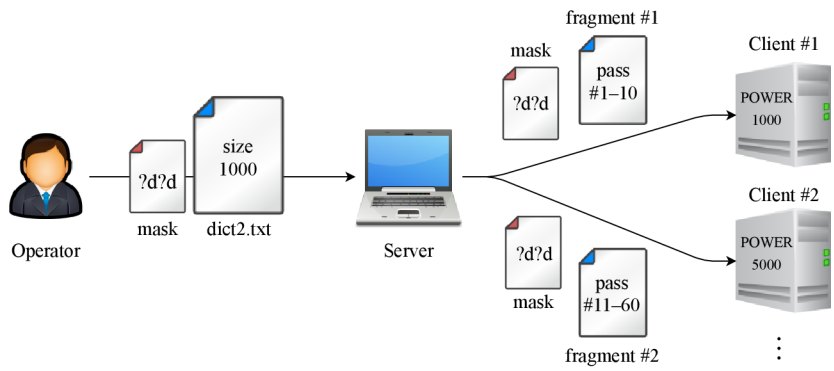


Figure 3.20: An example of the improved hybrid (mask+wordlist) attack distribution

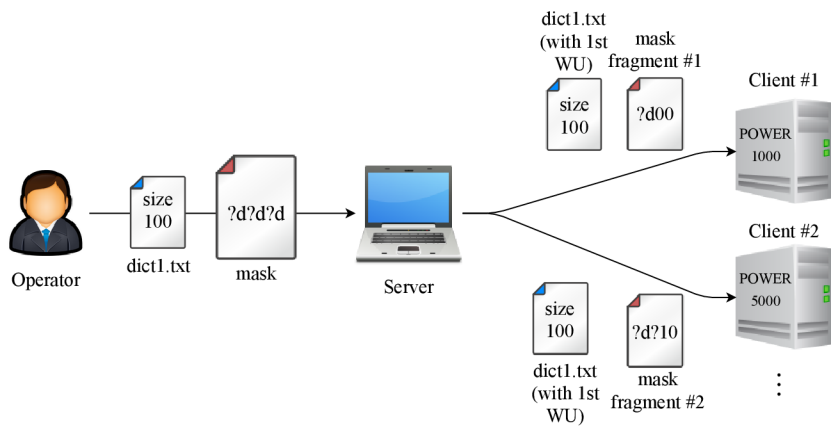


Figure 3.21: An example of the improved hybrid (wordlist+mask) attack distribution

Algorithm 5: The GetCharsetSlice() function for limiting character sets

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

```
if desiredSize > |charset| * 0.75 then
    desiredSize = |charset|
else if desiredsize > |charset|/2 then
    desiredSize = |charset|/2
if desiredSize ≤ 1 then
    return charset[0]
else
    return charset[0:desiredSize]
```

Algorithm 6: Building a password mask with close to the desired keyspace.

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

```
adjustedStartIndex = startIndex
resultKeyspace = 1
maskSlice = []
forall symbol ∈ mask do
    if ¬IsCharset(symbol) then
        continue
    charset = symbol
    charIndex = adjustedStartIndex mod |charset|
    adjustedStartIndex /= |charset|
    if charIndex > 0 then
        charset = charset[charIndex :]                    // Forced split
    if desiredKeyspace ≤ resultKeyspace then   // Desired keyspace reached
        maskSlice += charset[0]                  // Add a single character
        continue
    remainingKeyspace = desiredKeyspace/resultKeyspace
    if charIndex == 0 && remainingKeyspace ≥ |charset| then
        maskSlice += symbol
        resultKeyspace *= |charset|
    else                          // Only a slice of charset should be added
        maskSlice += GetCharsetSlice(remainingKeyspace, charset)
        desiredKeyspace = resultKeyspace              // Do not add any more
return maskSlice
```

The example from Figure 3.20 is relatively simple. The mask produces 100 different strings, and the dictionary has 1000 passwords. The first host needs 1000 passwords. Therefore, it receives a workunit with the mask ?d?d without limiting, and the first ten passwords from the right dictionary: $100 \times 10 = 1000$. The second host needs 5000 passwords, so it gets the same unlimited mask and 50 dictionary passwords: $100 \times 50 = 5000$.

The hybrid wordlist + mask attacks, on the other hand, use the mask slicing. I illustrate the principle using two examples. Consider a hybrid attack with mask `?d?h?l` on the right side. Suppose that we want to send a mask with 20 passwords. The first substitute symbol C_1 is `?d` with 10 different possibilities for the first character: $|C_1| = 10$. The desired keyspace is two times higher, so we leave the first symbol intact and proceed to the second one. The second symbol is $C_2 =$ `?h`, which stands for the character set containing `0123456789abcdef`. Thus $|C_2| = 16$ options. Nevertheless, the current keyspace is 10, and to get 20, we only want to multiply the keyspace by 2. Therefore, we take the first two characters from `?h` and create a custom character set `?1` containing `01`. After that, we reached the desired keyspace of 20. Hence, no more multiplication is needed. Therefore, from the next substitute symbol $C_3 =$ `?l` we take only the first character `a`. The slicing is completed, and we send the mask `?d?1a`.

For the next workunit, suppose that we want a mask with keyspace 150. We already processed 20 passwords from the mask because we used the first substitute symbol `?d` with keyspace 10 with the first two characters of `?h`. From the `?h`, 14 characters are remaining. To avoid unnecessary complexity, the heuristic from Algorithm 5 forces Fitcrack to finish the fragmented character set first. Therefore, the resulting mask is `?d?1a` with the custom character set `?1` containing `23456789abcdef`. The workunit has a keyspace of 140, which is as close to 150 as allowed in the current case.

Assume the third workunit should have the keyspace of 160. Therefore, we leave the first two symbols `?d?h` intact because they produce exactly the desired number. Since the first 160 strings were already generated, we only need to change `a` for `b` as the next symbol from the `?l` character set. In this case, no custom character is necessary, and the resulting mask is `?d?hb`.

The second example follows the situation in Figure 3.21. Both the dictionary and mask have a keyspace of 100. In this case, there are two options: either fragmenting the left dictionary or slicing the mask. The illustration shows the slicing option. The first host needs a workunit of 1000 passwords. Therefore, it receives the entire dictionary of 100 words and the slice `?d00` of the mask that produces ten strings: $100 \times 10 = 1000$. The second host needs 5000 passwords. The sliced mask is `?d?10` where `?1` is the custom character set containing `12345`: $100 \times 50 = 5000$.

3.8.5 PCFG Attack

The use of *probabilistic context-free grammars* (PCFG) for password cracking was originally proposed by Weir et al. [213] The attack is based on the previous knowledge of user passwords whose structure is represented by a grammar. The grammar is a mathematical model that describes the allowed forms of candidate passwords. Rewrite rules have probability values assigned to denote what fragments of symbols will more occur in candidate passwords. The goal is to create more probable passwords first. Such probabilistic attacks offer a time-space trade-off and high efficiency against human-created passwords. The keyspace is usually much lower than with a brute-force attack, but higher than with a simple dictionary attack with the original wordlist. The technique allows creating completely new

passwords that do not exist in the original dictionary. Chapter 4 describes the PCFG-based attacks in detail.

An example of a PCFG attack is displayed in Figure 3.22. The input wordlist called *training dictionary* contains two passwords. By an automated processing of the password, we create a probabilistic grammar. Each rewrite rule has a probability value assigned. One can see that rule "$A4 \rightarrow love$" has a higher probability value than rules "$A4 \rightarrow pass$" and "$A4 \rightarrow word$". This is because "love" occurred twice in the training dictionary, while "pass" and "word" only once. In the cracking session. Using the algorithms described in Chapter 4, we use different sequences of rules to rewrite the start nonterminal S to obtain the strings generated by the grammar - the candidate passwords (password guesses). We can calculate the probability of each password as a product of probabilities of all applied rewrite rules. The algorithms ensure that all possible passwords are eventually created. Moreover, they are generated in the non-increasing probability order, so that more probable passwords are produced first. The keyspace k of a PCFG attack with grammar G can be calculated as:

$$k = \sum_{B \in G} cnt_base(B). \tag{3.6}$$

where the $B = N_1 N_2 \ldots N_n$ is a base structure, i.e., a sentential form created directly from the start nonterminal. The $cnt_base(B)$ calculates the number of possible password guesses for that base structure. The detailed description of keyspace calculation is described in Section 4.7.2.

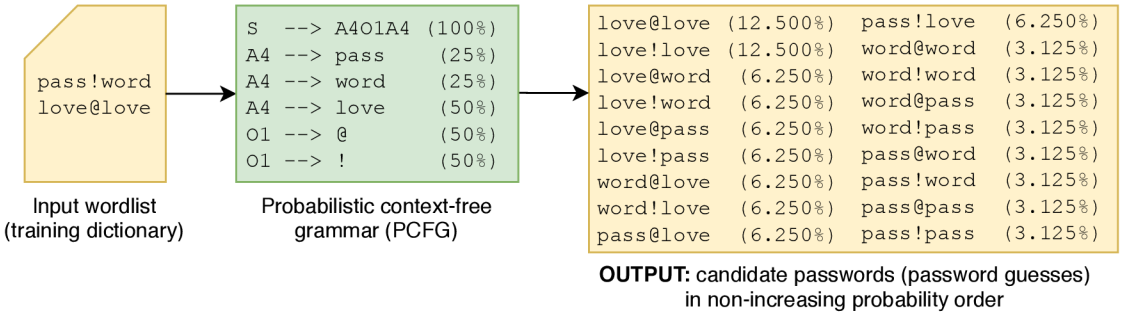


Figure 3.22: An example of PCFG password cracking

Distribution Strategies for PCFG Attacks

Similarly to the other attack modes, we need to deliver the passwords to the cracking nodes somehow. A naive solution is to generate all candidate passwords on the server and use the distribution scheme as with the dictionary attack. However, with the dictionary attack, wordlists of passwords are prepared before the attack even starts. In the PCFG attack, the input is a probabilistic grammar, not a wordlist. As discussed in Section 4.2.5, generating password guesses is computationally complex and has high memory requirements [82]. The process may take a significant amount of time, and often the grammar cannot be processed entirely. Moreover, transferring the passwords in their final form has high requirements on the network bandwidth, as I experimentally prove in Section 4.9.3. The experiments also show that another drawback of the naive solution is limited scalability. Since all the passwords are generated on a single node, the server may easily become a bottleneck of the entire network.

Therefore, Fitcrack uses the distribution of preterminal structures (see Section 4.4.3), i.e., sentential forms representing partially generated passwords. The idea utilizes the fact that each preterminal structure produces passwords with the same probability. The server only generates the preterminal structures (PT), while the terminal structures, i.e., the candidate passwords, are produced by the cracking nodes.

To perform a PCFG attack, Fitcrack needs a probabilistic grammar in the format used by Weir's PCFG Trainer[4]. Fitcrack supports two ways of obtaining the grammar. The WebAdmin either allows the user to upload a ZIP archive containing an already-created grammar, or to select a password dictionary while the WebAdmin will let the PCFG Trainer process it and create a grammar automatically.

To generate preterminal structures and password guesses, Fitcrack uses the enhanced version of PCFG Manager[10] proposed in Chapter 4. For distributed computing, the PCFG Manager can be run either as a standalone server that generates PTs, or as a client that generates the final password guesses. Both sides can communicate via gRPC[13] and Protocol buffers[14], as described in Section 4.8.1. The concept is utilized in Fitcrack as well with slight modifications since the client-server communication in BOINC is based on passing input/output files. With each workunit of a PCFG attack job, two extra files are sent:

- `preterminals` - the file contains one or more preterminal structures that are used for generating password guesses within the workunit.
- `grammar` - the PCFG grammar in a marshalled (serialized) form. The serialization is performed by the `Pcfg` endpoint in WebAdmin backend (see Section 3.7.6) at the time the grammar is created. The file is intentionally marked as *sticky* which implies that BOINC will only send it once - with the first workunit.

When a user launches a PCFG attack, the PCFG Monitor daemon (see Section 3.7.7) starts the PCFG Manager server to listen at a TCP port calculated as:

$$50050 + (jobId \ \% \ 1000)$$

where $JobId$ is the ID of the corresponding job. This allows to run multiple PCFG attack at the same time. Using the `Connect()` call, the Generator (see Section 3.7.1) then connects to the running instance of the PCFG Manager server.

When creating a workunit, the Generator invokes `GetNextItems()` call to obtain one or more preterminal structures. With the call, the Generator also specifies a keyspace value that is necessary to enable the adaptive scheduling (see Section 3.6.5). In response, the PCFG Manager server will then give the Generator a chunk of as many PTs as necessary to generate as least the desired amount of password, and no more. An example is illustrated in Figure 3.23. The exact match can not be guaranteed because different PTs may generate different number of password guesses. The generator creates a `preterminals` file with all obtained PTs. The `preterminals` file together with the `grammar` file represent the input data for the new workunit.

On the client side, the Runner (see Section 3.7.11) launches an instance of PCFG Manager client (see Section 3.7.7) and passes it the `grammar` and `preterminals` file. Using a pipe, the Runner then connects standard output of the PCFG Manager to the standard input of hashcat. The hashcat is started in wordlist attack mode without the specification of a concrete dictionary, so that it reads all the passwords guesses directly from the pipe. For cracking more complex hash algorithms, the PCFG Manager may generate guesses faster than the hashcat manager to verify them. Thus, the Runner sets the pipe as buffered and blocking to make the PCFG Manager wait if necessary.

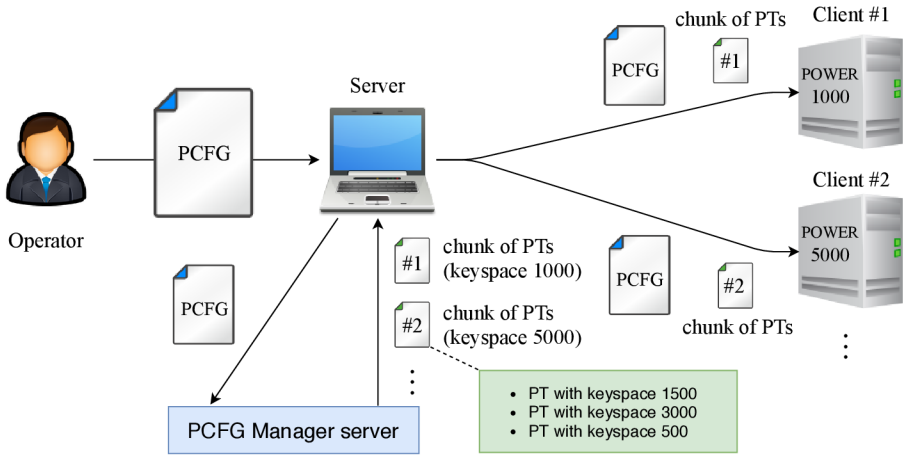


Figure 3.23: Example of PCFG attack distribution

3.8.6 PRINCE Attack

PRINCE (*PRobability INfinite Chained Elements*) is a modern password generation algorithm that can be used for advanced combination attacks. Jens Steube designed this algorithm to use only one vocabulary instead of two different dictionaries and then generate chains of combined words. These chains can contain one to N words derived from the input dictionary and concatenated together. The reference implementation of the algorithm, the *princeprocessor* tool, is available[57] under the MIT license.

Basic Components

The PRINCE algorithm is based on following components: elements, chains, and keyspace.

- **Element** - An element is the smallest entity representing a single unmodified word from the input dictionary. All elements are sorted by their relevance (occurrence frequencies), grouped by the length, and saved into the database of elements. The database contains several tables, one for elements of each length. Table 3.8 shows an example of elements and the table they are stored in.

Word	Table
123456	6
password	8
1	1
qwerty	6
...	...

Table 3.8: Examples of elements

- **Chain** - A chain of length L is a sorted sequence of element lengths whose sum is L. For example, a chain with its length 8 can be (1, 6, 1) or (8). For the length L there are $2^{(L-1)}$ different chains.

[57] https://github.com/hashcat/princeprocessor

Example: chain with length 4 can be created by the following elements:

- 4 letter word,
- 2 letter word + 2 letter word,
- 1 letter word + 3 letter word,
- 1 letter word + 1 letter word + 2 letter word,
- 1 letter word + 2 letter word + 1 letter word,
- 1 letter word + 1 letter word + 1 letter word + 1 letter word,
- ...

- **Keyspace** - A keyspace is the number of all candidate passwords obtained by combining all available elements. The algorithm combines the elements according to the sorted sequence of lengths in the processed chain. For example, if we have X elements of length 2 and Y elements of length 6 in our input dictionary, then the keyspace of chain (6, 2) is $X * Y$. Keyspaces of some chains from the *RockYou* dictionary are shown in Table 3.9.

Chain	Elements	Keyspace
3 + 1	335 * 45	15 075
1 + 3	45 * 335	15 075
4	17889	17889
2 + 2	335 * 335	112 225
2 + 1 + 1	335 * 45 * 45	678 375
1 + 2 + 1	45 * 335 * 45	678 375
1 + 1 + 2	45 * 45 * 335	678 375
1 + 1 + 1 + 1	45 * 45 * 45 * 45	4 100 625

Table 3.9: Keyspace of chains with length 4 (passwords from *rockyyou*)

Algorithm 7 shows how the PRINCE algorithm works. The essential input is the dictionary (`D`) from which the words are taken. The following inputs set the boundaries on generating passwords from word chains and ensure that the guessing process eventually ends. The `EMIN` and `EMAX` parameters define the minimal and maximal number of words in a chain. By setting both to 1, we get a classic dictionary attack without combinations. The other two boundaries are `PASSMIN` and `PASSMAX` that represent the minimal and maximal number of characters in generated passwords. Naturally, the `EMAX` and `PASSMAX` parameters have a dramatic impact on the resulting keyspace. The last parameter is a boolean `CASEPERM` that allows case permutation of the first letter in words. If enabled (when `CASEPERM` is 1), each word from D that begins with a letter is used twice: with a lowercase first letter and with an uppercase first letter. This technique allows generating passwords like "helloHowAreYou" from a dictionary $D = \{are, how, You, Hello\}$, for instance.

If used with the princeprocessor tool, the only required input is D. Other parameters are optional. If not defined, the application uses default values. At the time of writing this thesis, the latest release of princeprocessor is version 0.22. Here, the default `EMIN` and `EMAX` are 1 and 8. For password length settings, the default `PASSMIN` and `PASSMAX` are 1 and 16. Case permutation is disabled by default. Besides, the tool provides some additional options like calculating output password length distribution, eliminating duplicate words, saving state, or storing the output into a pre-defined text file.

Algorithm 7: Pseudocode of PRINCE algorithm

Input: input dictionary (D), minimal number of elements in chain (EMIN), maximal number of elements in chain (EMAX), minimal length of passwords (PASSMIN), maximal length of passwords (PASSMAX), case permutation (CASEPERM)

Output: password candidates

```
1 elements = read_elements(D);
2 chains = [];
3 chain_keyspaces = [];
4 while new_chain = combine_new_chain(elements, EMIN, EMAX, PASSMIN,
    PASSMAX, CASEPERM) do
5     chains.append(new_chain);
6 for i ← 0 to size(chains) by 1 do
7     chain_keyspaces[i] = compute_keyspace(chains[i]);
8 sorted_chains = sort_chains_by_keyspace(chains, chain_keyspaces);
9 print(sorted_chains, stdout);
```

Distribution Strategies for the PRINCE Attacks

The PRINCE is another attack mode of Fitcrack that employs an external password generator. The input of the attack is a password dictionary and a set of options. For workload distribution, there are few possible strategies. Similarly to the naive PCFG attack, it is possible to generate all candidate passwords on the server and send them through the network, resembling a classic dictionary attack. Such a strategy, however, puts a significant load on the server and creates a massive overhead for network communication. Another approach is to divide the password creation by length, e.g., let one node generate passwords of one length and another node passwords of another length, etc. Nevertheless, the length-based distribution would create chunks of significantly different keyspaces. Such a method could not be used for fine-grained workunit tailoring.

Luckily, the princeprocessor utility supports the `--skip` and `--limit` parameters, so it is possible to use a strategy similar to the mask attack. Therefore, Fitcrack runs the princeprocessor utility on the client and supplies hashcat with passwords using a pipe, as illustrated in Figure 3.7. The key idea behind the PRINCE attack distribution is controlling the tool by with the two options. With `--skip=X`, it skips the first X candidate passwords from the start. With `--limit=Y`, it generates exactly Y candidate passwords. No additional optimizations from Section 3.6.4 are used by the princeprocessor tool.

With each workunit, Fitcrack assigns a password range of password indexes to all active hosts according to their performance. This information about the password range is a part of the configuration file of every PRINCE attack workunit. To determine the total keyspace of a job, the WebAdmin backend asks the princeprocessor tool. Based on the selected dictionary and options, it calculates the actual keyspace of the attack.

An example of attack distribution is shown in Figure 3.24. A user creates new PRINCE attack job. The server receives the configuration of the job together with the user-specified dictionary. Then, the Generator on the server side assigns a keyspace range to every active client according to the benchmarked "power" of the client. This range info is a part of the config of every workunit.

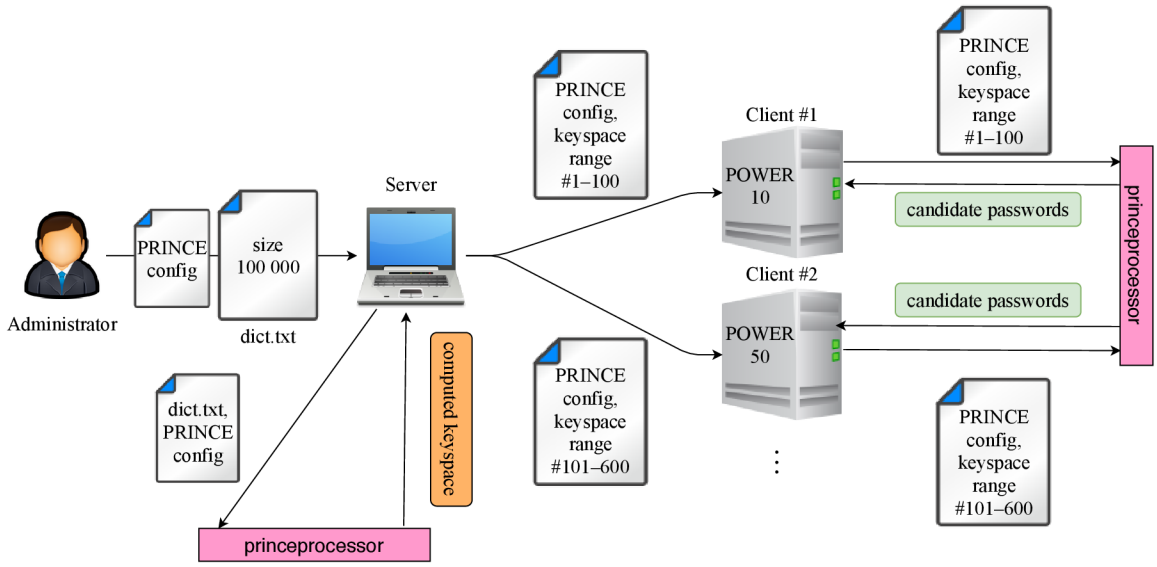


Figure 3.24: Scheme of PRINCE attack distribution

The Runner on the client launches the princeprocessor tool with options --skip and --limit to generate assigned range of candidate passwords using the PRINCE algorithm. Runner internally connects princeprocessor to hashcat using a pipe, so that hashcat can verify the candidate passwords as soon as they are available as the output of princeprocessor.

3.9 Experimental Results

This section aims to verify the usability of the proposed Fitcrack system, related principles, and techniques. Using different scenarios, I illustrate the practical impact of the algorithms integrated to Fitcrack's subsystems. Moreover, I present a series of comparisons with the Hashtopolis tool. At the time of writing this thesis, Hashtopolis is most likely the only other maintained open-source hashcat-based distributed password cracking solution.

The experiments are structured into multiple sections that analyze different aspects of the system. Section 3.9.1 studies the general properties of workload distribution for password cracking tasks. It analyzes both CPU and GPU-based networks and answers what kind of assignments are worth distributing. The large-scale experiments with up to 55 physical computers also examine the efficiency and overhead of distributed attacks. Section 3.9.2 illustrates the practical impacts of the adaptive scheduling algorithm proposed in Section 3.6.5. It discusses the problem of benchmark accuracy and its impacts. The included experiments also compare the old and new method for benchmarking. The rest of the sections study the distribution strategies for different attack modes, described in Section 3.8. The examined metrics involve not only the absolute cracking time, but also the performance, efficiency, scalability, and other aspects. I compare different attack modes in terms of password guessing complexity, the overhead for network transfer, and feasibility for cracking simple and complex hash algorithms.

3.9.1 The Time and Efficiency

Before proceeding to the actual scheduling algorithms and distribution strategies, I want to study the efficiency and other properties of distributed cracking in general. The goal is to analyze if and when the attack is worth distribution, the difference between CPU and GPU nodes, and the scalability of such attacks.

This section describes a series of large-scale experiments on up to 55 physical computers. I performed these early experiments in 2016 using the original non-hashcat version of Fitcrack with custom-made OpenCL cracking kernels [81]. Unfortunately, at that time, I did not have access to that high number of GPU nodes. Therefore, most of the hosts are CPU-based. Nevertheless, it does not matter since the principles are still the same. Later experiments showed me that GPU-based networks have similar behavior; only the performance is generally higher.

Computing Network

The experiments were performed using 2, 4, 8, 16, 37, and 55 CPU hosts and using 1, 2, and 4 GPUs on a single GPU host. Table 3.10 shows the hardware configuration of the nodes. The last line shows the password cracking performance of a single GPU. Each line of the table describes one group of nodes involved in computing. The displayed performance corresponds to the benchmark tests on a given processor. The employment of CPU nodes was from top to bottom, e.g., A 37-node attack used 16 nodes with Intel i5-4460, and 21 with Intel i3-4340, etc. We can also see that the benchmarked performance of the GPU node is much higher. A single AMD Radeon R9 Fury X unit is "strong" as 15 individual Intel i5-4460 processors (15 * 8 M = 120 M).

Nodes	Processor	Performance [p/s]
1–16	Intel i5-4460, 3.2 GHz	~ 8,000,000
17–37	Intel i3-4340, 3,6 GHz	~ 5,000,000
38–55	Intel E8400, 3,0 GHz	~ 2,700,000
GPU	AMD Gigabyte R9 Fury X, 4 GB	~ 120,000,000

Table 3.10: The configuration of working nodes

Processing the Entire Keyspace

All experiments used the classic incremental brute-force attack on an encrypted PDF 1.7 revision 5 file. In the first scenario, I tested the worst-case cracking time, i.e., the time required to generate and verify all candidate passwords. The alphabet contained lowercase Latin letters. For each configuration of the cracking network, I created five jobs. Each with a different maximum password length: 5, 6, 7, 8, and 9. Table 3.11 shows the total keyspace of all jobs. For example, with a maximum length of 5, we need to test $26 + 26^2 + 26^3 + 26^4 + 26^5 = 12,356,630$ different passwords. The complexity grows exponentially with each extra position. I ran these five jobs using the following network configurations: a server + 2, 4, 8, 16, 27, and 55 CPU nodes; and a single node with 1, 2, and 4 GPUs.

Maximum length	Keyspace
5	12,356,630
6	321,272,406
7	8,353,082,582
8	217,180,147,158
9	5,646,683,826,134

Table 3.11: The keyspace of worst-case cracking jobs

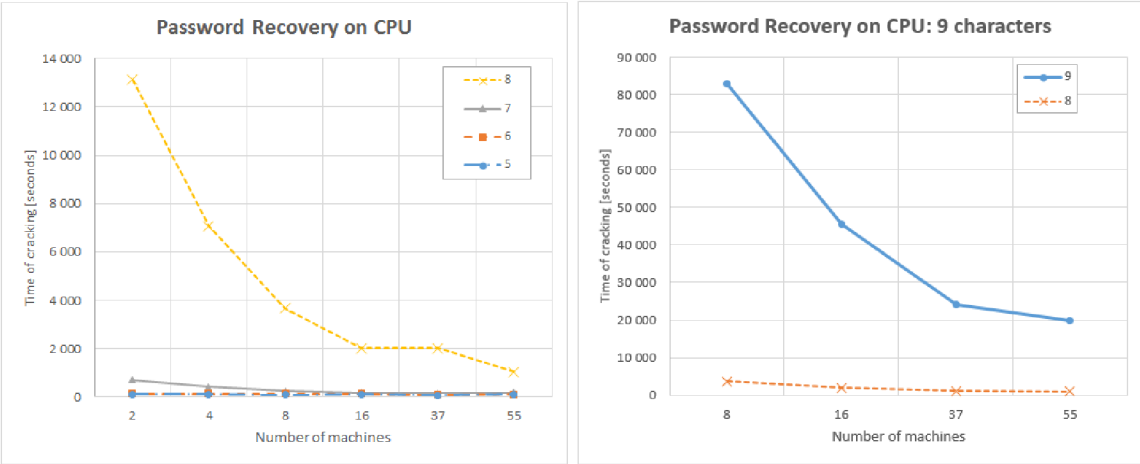
Table 3.12 displays the absolute cracking time in seconds. The most time-exhausting experiment was the maximum length of 9 on 8 CPU hosts, which took over 23 hours. Therefore, I made an exception and have not tried this job on 2 and 4 nodes because it would take more than a day. See also the opposite extreme. The values measured for the easiest job do not even correlate with the available computational power. The assignment is simply too easy to be computed on multiple devices. Cracking with four GPUs was even longer than with one or two since the computer spent time with a pointless initialization of another two cards.

Maximum	Number of nodes								
pass. length	2	4	8	16	27	55	1 GPU	2 GPU	4 GPU
5	100	101	93	104	60	100	3	3	4
6	130	132	132	134	87	99	6	4	4
7	714	415	236	167	186	184	74	39	21
8	13 125	7 051	3 679	2 025	1 177	1045	1 885	961	475
9	–	–	82 962	45 567	24 011	19 779	49 196	25 120	12 379

Table 3.12: Time (in seconds) of distributed CPU and GPU-based password cracking

First, I analyze the distributed CPU-based approach. Figure 3.25(a) shows the cracking time in seconds of jobs of the maximum password length between 5 and 8. For passwords

up to the maximum length of 6, the cracking is so quick that adding new nodes does not add any substantial acceleration. For 7-character passwords, the distribution was still profitable with a 16-host network, but increasing the number of hosts to 27 and 55 did not bring any good. The added overhead for communication with the redundant nodes only extended the overall time. The 8-character job, however, employed 27 hosts without any problem. Figure 3.25(b) shows the same for lengths 8 and 9 with 8+ nodes. We see very similar progress only with a different scale of the Y-axis. For 9-character passwords, the advantage of distributed computing is evident. Cracking on two or four CPU nodes is impossible to perform within a day. Eight nodes could find the password in 82 962 seconds (approx. 23 hours) while 55 nodes in 19 779 seconds (5.5 hours). Overall, more complex jobs scaled well, even with a higher number of hosts. Also, do not forget that nodes 17–37 had a lower performance than the first 16, and 38–55 even lower, as described in Table 3.10. If all nodes had the same performance, I assume, the scalability would be much closer to linear.



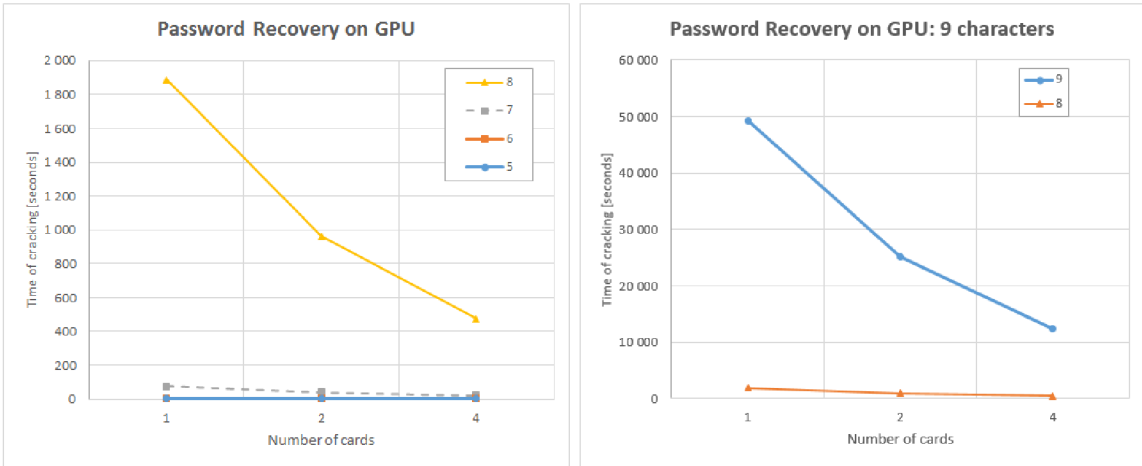
(a) 5 to 8 characters

(b) 9-character password compared with 8-character

Figure 3.25: Worst-case cracking time on CPU nodes

It is interesting to compare these results with the GPU-based version. For jobs up to the length of 8, Figure 3.26(a) shows the cracking time, but this time on GPU. On the X-Axis, we see the number of employed GPU units. The 5 and 6-character passwords were cracked almost instantly, even with a single GPU. Similarly to the distributed solution, multi-GPU cracking was advantageous for passwords of 7 characters and longer. The absolute cracking time of 8-character passwords on one GPU processor corresponds was close to the result obtained with 16 CPU nodes. Figure 3.26(b) displays the results for 8 and 9-character jobs. We see the scalability is almost linear. 9-character passwords can be processed within 49 196 secs (13.6 hours) on 1 GPU and in 12 379 secs (3.4 hours) on 4 GPUs.

While the measured cracking times provide a basic overview of the cracking networks' capabilities, they do not directly document the actual utilization. Therefore, I measured the actual processing times of individual workunits or all nodes to calculate the cracking efficiency. Page et al. define the efficiency of distributed computing as "the percentage of the time that processors actually spend processing rather than communicating or idling" [146]. For example, assume that a job that lasts 100 minutes has an efficiency of 70 %. In this case, 1 hour and 10 minutes are spent by actual cracking, and the remaining 30 minutes are the overhead for communication and synchronization. The efficiency E_{ff} is computed



(a) 5 to 8 characters

(b) 9-character password compared with 8-character

Figure 3.26: Worst-case time of GPU-accelerated cracking

using the following formula:

$$E_{\text{ff}} = \frac{\sum_{x=1}^{N} t_x}{N * T_{fin}}$$

where N is the number of nodes that participated in the computing, t_x is the the real time node x spend by processing, and T_{fin} is the wall clock time of the entire job from start to the end. Naturally, $1 - E_{\text{ff}}$ is the overhead. The efficiency of all previously described jobs is displayed in Figure 3.27. The X-axis represents the maximum password length that defines the complexity of the job. The Y-axis shows the efficiency. Each color line is a different configuration of the cracking network.

The graph shows that the longer the maximum password is, the more efficient cracking we can achieve. The efficiency of cracking short passwords is low since the jobs are too easy and employ more resources than needed. Therefore, the overhead for the initial benchmarking, sending workunit assignments, and reporting results, is much higher in terms of the entire job. In contrast, in a job that lasts multiple hours, several seconds of a benchmark is neglectable. Also, the single-machine GPU approach is more efficient because there is no overhead for intra-node communication. We can see, no matter what cracking network we have, the trend is ascending in all cases. With the increasing complexity of the job, the efficiency gets closer to 100 %. In other words, for every cracking network, we can find a task difficult enough to employ it efficiently. If a job is too easy to utilize all nodes efficiently, we can use only some of them.

Undoubtedly, the experiment showed that distributed cracking is efficient and advantageous if the cracking network is selected adequately to the job's complexity. We also see the high potential of GPUs. In this case, a single powerful GPU node could beat an entire network of CPU nodes. However, this experiment purposedly employed a relatively simple algorithm and had an alphabet limited to lowercase letters. For more complex algorithms or stronger passwords, we would need to use multiple GPU nodes. Down below in this chapter, I also show various experiments with the actual distributed cracking on GPU nodes using different attack modes. Nevertheless, the principles of the relation between the network size, job complexity, cracking time, and efficiency, described in this section, are still the same.

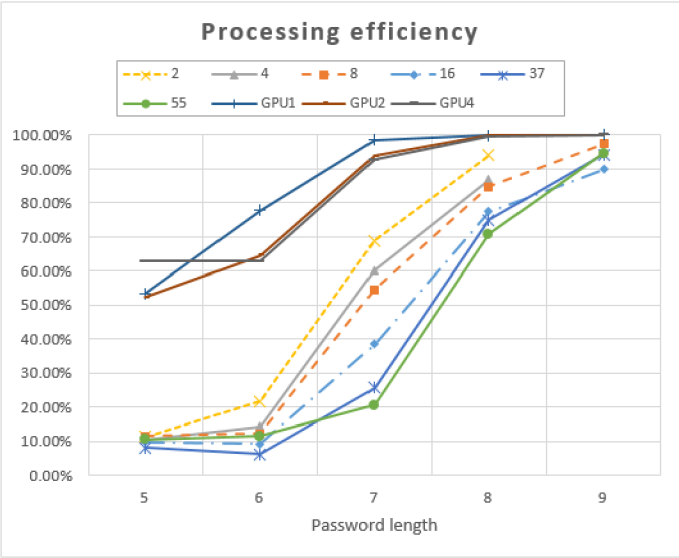


Figure 3.27: Efficiency of the distributed computing

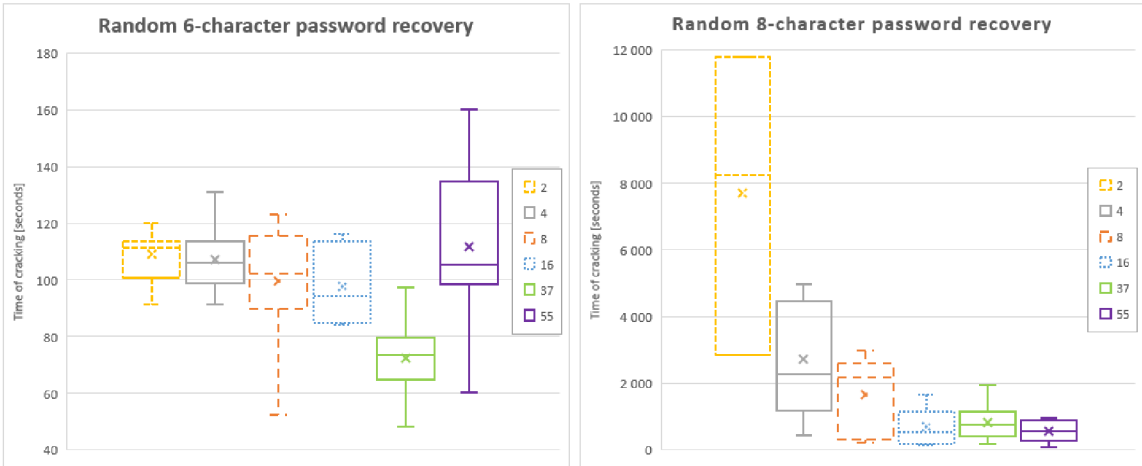
Random Password Cracking

The previous scenario analyzed the worst-case time that is required to process the entire keyspace. In real cases, this is often not necessary since the correct password could be found much earlier. To simulate the real situations more precisely and study the dispersion of different possible cracking sessions, I decided to perform another series of experiments. The next scenario uses the same PDF format with the same 2, 4, 8, 16, 37, and 55 hosts. The difference is that the passwords are random. For each host configuration and password length, I generated ten different random passwords. Then, I measured how long it takes for the system to crack them. Like in the previous experiments, the password generating started from one-character passwords and subsequently continued to longer ones.

Figure 3.28(a) shows the results of cracking random 6-character passwords. All jobs were too simple to be even worth distributed computing. Hence, there is no correlation between the number of hosts and cracking time. Cracking 5-character passwords produced a similar outcome. The actual benefit of distributed computing came with cracking 7 and 8-character passwords. Figure 3.28(b) shows the times for the 8-character ones. Increasing the number of nodes up to 4, 8, and 16 reduced the cracking time dramatically. Additional nodes, however, brought no further speedup. This expected trend is very similar to the previous experiment. The results further validate the principles described above. Above all, the size of the cracking network should match the complexity of the given task.

The Cost Model

An important question of forensic experts is what the cost of the password cracking is? Therefore, this section compares the cost models for cracking networks used in previous experiments. Concretely, CPU nodes 1 to 16 and a host with GPUs from Table 3.10. For each node, I calculated the total price of its hardware. Table 3.13 provides an overview of the cost of working nodes based on 2015 prices. The *unit price* is a price of processing 1,000,000 PDF 1.7 passwords per second on the given hardware. The unit price is constant for distributed computing on nodes with the same configuration. In our case, it is $79. The



(a) Password length up to 6 characters

(b) Password length up to 8 characters

Figure 3.28: Random password cracking

unit price of GPU solutions is lower when inserting additional GPU cards. Please note that the price considers the entire computer, not only the processors. The motherboard, disks, RAM, PSU, and other parts are taken into account as well.

The table shows that the GPU-based solutions are much cheaper than CPU-based cracking, even using a cluster. If the cluster is already available, then the distributed solution can provide an alternative to the high-performance multi-GPU node with high initial costs. Also note, with a change of PSU and RAM, each CPU node from the cluster is upgradeable to a GPU one. The investment of about $1000 could change the power from 8 to 120 million passwords per second.

Node(s)	**Speed (pwd / sec)**	**Price (USD)**	**Unit price USD**
1	8,000,000	639	79
2	16,000,000	1278	79
4	32,000,000	2556	79
8	64,000,000	5 112	79
16	128,000,000	10 224	79
1 GPU	120,353,069	2 337	19
2 GPU	240,706,138	3 151	13
4 GPU	481,412,276	4 440	9

Table 3.13: Relative cost of the solution

Another important factor connected with the cost model is energy consumption, displayed in Table 3.14. The first two columns show the power consumption (in Watts) of working nodes during an idle state and processing (load state). The next two columns represent the real costs of cracking the PDF 1.7 secured by passwords of lengths 8 and 9 with the unit price of $0.12 per kWh. The table shows that the power consumption of the 16-node cluster is similar to a 4-GPU node. However, the prices of energy consumed to perform the jobs are lower on GPU because of faster computing. The jobs can be processed there in a much shorter time.

Node(s)	Idle (W)	Load (W)	Price L8 (USD)	Price L9 (USD)
1	31	58	-	-
2	62	116	0.0493	-
4	124	232	0.0518	-
8	248	464	0.0531	1.2678
16	496	928	0.0561	1.3437
1 GPU	182	370	0.0232	0.6067
2 GPU	182	541	0.0173	0.4530
4 GPU	182	856	0.0135	0.3532

Table 3.14: Power consumption and the cost

The overall power consumption (in watt-hours) is calculated as the electrical power consumed during the effective computing (P_{Load}) and the power consumed during the idle time (P_{Idle}): $P_{Total} = (N*T_{fin}*(E_{\text{ff}}*P_{Load}+(1-E_{\text{ff}})*P_{Idle}))$ / 3600, where N is the number of nodes, T_{fin} is the wall clock time of the entire job, E_{ff} is the efficiency, P_{Load} and P_{Idle} are the power consumption during the password cracking and the idle state, respectively. The total power consumption spent on cracking password of length 5 to 8 on CPU and GPU nodes is shown in Figure 3.29.

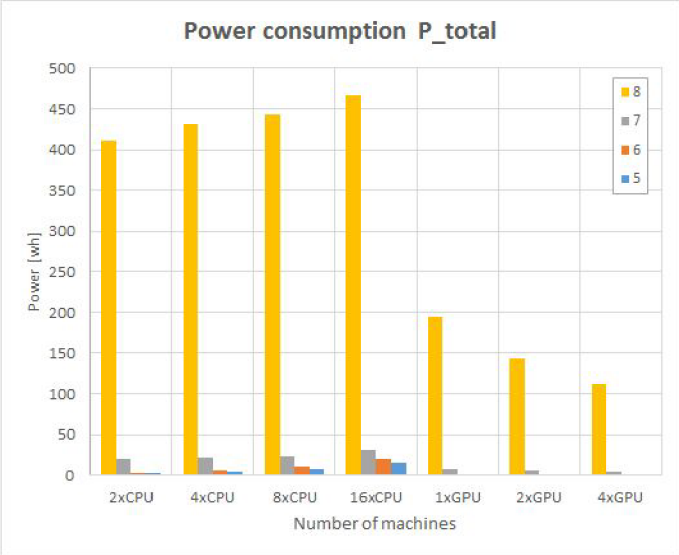


Figure 3.29: Power consumption of the computing

In terms of power consumption, the password cracking is generally more efficient on GPU cards than CPU units. An interesting observation is that when we add new CPU nodes, the total power consumption P_{total} softly rises. On the GPU machine, adding new units decreases the total consumption. The reason is that the job is over much earlier, even though the peak power consumption is higher. Another cause is that in the experiments, GPU units shared the same motherboard and PSUs. Adding a new GPU did not require starting another computer.

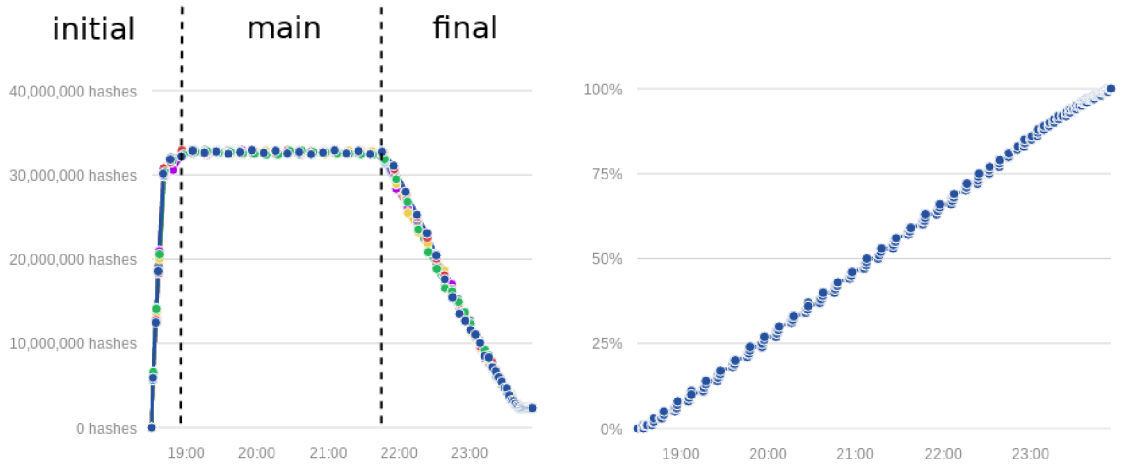
3.9.2 Adaptive Scheduling

This section shows how the proposed *adaptive scheduling algorithm* [81, 84] affects generating workunits. Firstly, it illustrates the impact of the algorithm in different phases of a job. Secondly, we explore how the system reacts to a sudden change in a host's performance. Finally, it compares the original solution for benchmarking with the improved one.

Fitcrack's Scheduling Algorithm

Fitcrack and Hashtopolis tools allow the user to specify the *chunk size* as the desired number of seconds for processing each workunit. In Fitcrack, the setting is called *seconds per workunit*. Based on the benchmark of hosts, performed at the start of each cracking job, both systems try to create fine-tailored workunits to fit the defined chunk size. While Hashtopolis strictly respects the user-entered chunk size and uses the same tailoring mechanism from the very start till the end, Fitcrack utilizes the *ramp-up* and *ramp-down* techniques. The details and motivation are described in Section 3.6.5.

Figure 3.30 illustrates the practical impact of the original scheduling algorithm using an experiment with a brute-force attack on SHA-1 hash. The attack employed 8 hosts with NVIDIA GTX 1050 Ti GPU and a password mask made of 10 lowercase letters (10x `?l`). The seconds per workunit was 10 minutes, i.e., $t_{pmax} = 600$. The system-wide settings were $\alpha = 0.1$ and $t_{pmin} = 60$.



(a) Workunit size in job phases

(b) Job progress

Figure 3.30: Illustration of adaptive workunit scheduling

The chart in Figure 3.30(a) illustrates how workunit size changes over time. It displays 8 concurrent progresses with a different color for each node, however, we can see they precisely overlap. This is an anticipated result since all nodes had the same GPUs and no link outage, or computation error occured. According to the algorithm's specification, two changing variables modify the workunit size: the *elapsed time from start* (t_J), and the *remaining keyspace* ($|P_R|$). The cracking took the interval of 5 hours and 35 minutes, and the job progress chart is divided into three distinguishable parts. The *initial* phase, where $t_J \leq t_{pmax}$, showed the ramp-up and took approximately the first 10 minutes. At that time, the Generator did not create full-size workunits. Next, the *main* phase displays workunits of the same size being assigned in 10-minute intervals, which correlates with the

user-entered chunk size (t_{pmax}). Thanks to the distribution coefficient alpha, no more than 10 % of the remaining keyspace was distributed in any workunit. Therefore, in the *final* phase, we can see the ramp-down. The workunits shrank progressively, as expected. They, however, did not shrink infinitely. Once they reached the t_{pmin}, the ramp-down stopped. Figure 3.30(b) displays the job progress over time. We can see, the trend is, more or less, linear. The behavior of the scheduling algorithm meets its goals. After the initial ramp-up, the workunit size stabilizes. Workunits get smaller again at the end of the job to utilize all hosts by all means.

Adaptability

In an unstable environment, a host's performance may change over time. If another process utilizes the computer, the cracking speed may suddenly drop. In this experiment, I illustrate that the system can react appropriately to such an event. Moreover, I want to show that the ramp-up correctly converges to a stable state for nodes where no unexpected event occurs.

Therefore, I purposely chose hosts of different types and performance. To make the results comparable, I used faster CPUs and a slower GPU. The job was cracking an encrypted PDF 1.7 revision 5 (see Section A.1.1) using a brute-force attack. The reason for brute-force was to eliminate a possible impact of transferring dictionary passwords over the network. The cracking network consisted of four hosts, described in Table 3.15. The table also shows their performance measured by a benchmark.

Name	Processing unit	Performance [p/s]
Node A	Intel(R) Core(TM) i5-4200U @ 1.60 Ghz	~ 2,700,000
Node B	Intel(R) Core(TM) i7-5930K @ 3.50 Ghz	~ 14,800,000
Node C	Intel(R) Core(TM) i5-3570K @ 3.40 Ghz	~ 9,270,000
Node D	AMD Radeon R5 M255	~ 8,800,000

Table 3.15: Processing units and performance of hosts

Figure 3.31 displays the time points where workunits began and ended. The value of the Y-axis is the workunit size. We can see that despite the high potential of GPGPU, cracking on high-end CPU like Core i7-5930K can be faster than on low-end notebook graphics like Radeon R5 M255. I analyzed the workunit assignment at the beginning of the cracking.

At a marked time point, I intentionally added extra load to CPU threads of node B, causing its performance to go down. The sudden drop in cracking speed increased the processing time of the third workunit. We may see that after the initial ramp-up, the size of the workunits assigned to hosts A, C, and D became stable. Fore node B, the situation is different. Since the server received its result later than expected, it recalculated its v_i and made the following workunits smaller to match the desired processing time t_J. Once the load was removed, the algorithm reacted again by increasing the size of the next workunits for host B. In contrast, the keyspace of workunits assigned to nodes A, C, and D was relatively fixed.

The experiment showed that the algorithm behaves as expected. It calculates the workunit size accordingly to the performance of nodes. Moreover, it quickly adapts to a sudden change.

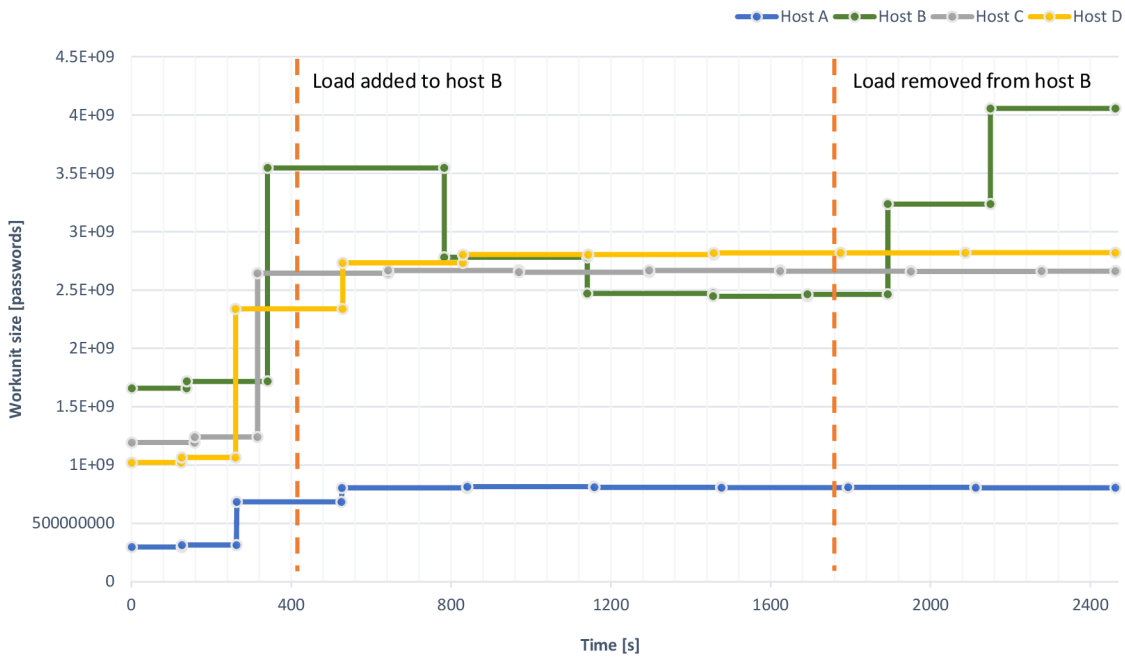


Figure 3.31: Fitcrack's reaction to an additional load

Improved Benchmarking

As discussed in Section 3.6.5, the baseline for workunit scheduling is the initial benchmark that is often inaccurate. To make a clear image, I performed a series of tests using different GPUs (NVIDIA, AMD) and hash algorithms (MD5 [172], SHA-1 [97], SHA-512 [76], and Whirlpool [188]) of a different computing complexity.

Table 3.16 shows the graphic cards used for the experiment. For each GPU, it shows the core clock, number of stream processors, effective memory clock, memory bandwidth, and the total memory capacity. I intentionally chose different classes of cards from both manufacturers to provide a clearer image of how the GPU selection affects the cracking sessions. NVIDIA GTX 1050 Ti and AMD Radeon RX 460 are low-end cards whose main advantage is value. The NVIDIA GTX 1080 Ti, AMD Radeon RX Vega, and AMD Radeon Fury X are high-end cards representing the best what the companies offered at the year of release.

The experiment compares the performance obtained using Hashcat's benchmark option with the actual performance of real cracking. Concretely, with a brute-force attack using `7x?a` mask, and a dictionary attack using a 1.1 GB wordlist. The measured cracking performances are shown in Table 3.17. For a brute-force attack, the actual cracking is significantly slower than the benchmark reports. For the dictionary attack, the measured speed is only a small fraction of the benchmark result. This result makes sense since hashcat needs to load and cache dictionary passwords, making the cracking operations much slower. Empty columns stand for OpenCL "CL_OUT_OF_RESOURCES" error, which occurred due to insufficient memory on the given GPU. It seems hashcat's implementation of Whirlpool cracking kernel has a high space complexity. I encountered this problem when trying to compute Whirlpool on AMD Radeon RX 460 and AMD Radeon R9 Fury X.

GPU	Stream processors		Memory		
	Clock [Mhz]	**Amount** [units]	**Eff. clock** [Mhz]	**Bandw.** [Gb/s]	**Cap.** [MB]
NVIDIA GTX 1050 Ti	1,291	768	7,008	112	4,096
NVIDIA GTX 1080 Ti	1,480	3,584	11,008	484	11,264
AMD Radeon RX 460	1,090	896	7,000	112	2,048
AMD Radeon RX Vega 64	1,247	4,096	1,890	485	8,192
AMD Radeon R9 Fury X	1,050	4,096	1,000	512	4,096

Table 3.16: Comparison of hardware specifications of used GPUs

		Performance [Mh/s]		
GPU	**Algorithm**	**Benchmark**	**Brute-force**	**Dictionary**
NVIDIA GTX 1050 Ti	MD5	6310	2425	21.00
	SHA-1	2022	1540	9.10
	SHA-512	302	47	14.00
	Whirlpool	66	58	16.00
NVIDIA GTX 1080 Ti	MD5	35401.5	11267.3	29.40
	SHA-1	11872.3	7263.8	29.10
	SHA-512	1416.9	192.3	25.50
	Whirlpool	338.9	306.3	26.70
AMD Radeon RX 460	MD5	4186	1277	7.00
	SHA-1	1400	800	7.70
	SHA-512	155	41	4.56
	Whirlpool	-	-	-
AMD Radeon RX Vega 64	MD5	26479	8134	41.46
	SHA-1	9260	5664	45.92
	SHA-512	1212	751	40.49
	Whirlpool	699	332	39.57
AMD Radeon R9 Fury X	MD5	17752	5548	9.40
	SHA-1	17754	3785	9.30
	SHA-512	534	75	7.90
	Whirlpool	527	257	-

Table 3.17: Difference between hashcat's benchmark and real attacks

The observed differences motivated me to change the benchmarking in Fitcrack. The classic benchmark option was too inaccurate. An alternative `--speed-only`, used in Hashtopolis, does not consider salted hashes. Therefore, Fitcrack's Runner subsystem starts an actual cracking session for a short time, as described in Section 3.6.

To compare the old and new benchmarking methods, I conducted another series of experiments that show the practical impact on actual cracking tasks. The goal here was to let Fitcrack create workunits whose processing will take as close to 300 seconds as possible. The experiments use a dictionary attack, a PCFG attack with an external password generator, and variations of a mask brute-force attack. The first one is without the Markov-based (see Section 3.8.3) guessing, the other two use a 3D Markovian model with the default `hashcat.hcstat` file. The experiments employ three different formats and also explore the impact of a cryptographic salt. The 7z format is computationally the most complex. The moderate SHA-512 was supplemented with a cryptographic salt. Finally, MD5 is easiest-to-compute, but the attack was performed on 20 hashes, each with a unique salt. The ramp-up was intentionally disabled to ensure Fitcrack calculates the first workunit's keyspace from the benchmarked performance only. For the purpose of this experiment, the ramp-up was intentionally disabled to let the scheduling system create full-sized workunits from the very beginning.

Table 3.18 shows the results. Both versions display the processing time of the first workunit and its percentual difference from the desired 300 seconds. The last column shows the difference between the old and the new version. All experiments show a noticetable improvement. We can see a dramatic change in the dictionary attack. As discussed, above the actual performance of the dictionary attack in the old version was very inaccurate from the benchmark. The greatest improvement was achieved in the last measurement. Every cryptographic salt means that the hash needs to be recomputed again. The reason is discussed and illustrated in Section 3.1.1. Since we have 20 different salts, the initial workunit could have been up to 20 times longer than the server estimated. Based on the results, I state that the improved benchmarking technique is definitely far more accurate.

Job		**Old version**		**New version**		**Change**
Attack	**Format**	**Time [s]**	**Diff**	**Time [s]**	**Diff**	**Diff**
dictionary	7z	1445	+482%	360.45	+20%	462%
PCFG	7z	355.57	+19%	291.71	−3%	16%
mask	SHA-512+salt	97.796	−67%	316.48	+5%	62%
mask+markov	SHA-512+salt	124.93	−58%	315.64	+5%	53%
mask+markov	20x MD5+salt	3570	+1090%	323.98	+8%	1082%

Table 3.18: Old and improved benchmarking for $t_{pmin} = 300$

3.9.3 Distributed Dictionary Attack

In this section, I analyze the distribution strategies for the dictionary attack. This attack mode is the most extensive to network transfer. There is no mask or grammar, neither any special algorithm for generating guesses. The server sends the actual candidate passwords in their final form, and the hosts use them directly to compute hashes. As described in Section 3.8.1, there are two basic strategies. The first method, used in the Hashtopolis tool, is to send the entire dictionary to all nodes. Then, the nodes receive ranges of password indexes, which serve as dictionary offsets. The other one, used in Fitcrack, is to distribute smaller dictionary fragments. To test both, I compare the two tools on a series of distributed GPU experiments. The computing network consisted of a server and 8 nodes equipped with NVIDIA GTX 1050 Ti, interconnected using a switch and 1 Gb/s Ethernet links.

To obtain comparable results in fair conditions, I measured the total time consisting of the: a) the benchmarking, b) data transfer, and c) the actual cracking. Fitcrack's total cracking time displayed in WebAdmin includes all these phases, while Hashtopolis displays only the last one. Moreover, Hashtopolis does not have a "start button". The cracking in Hashtopolis starts immediately once a *task* (equivalent to a job) has any *agents* (equivalent to hosts) assigned to it, and the agents are *active*. Therefore, I created an extra PHP extension that assigns and activates all agents at once. From that moment, it measures the time until the result of the last chunk is reported to the server [84]. Another metric used is the number of *chunks* generated and assigned to hosts. While Hashtopolis performs the benchmarking separately from the actual chunks, Fitcrack uses special zero-keyspace workunits. Thus, for comparison, I define that a *chunk* in Fitcrack equals to a non-benchmarking workunit.

It makes no sense to use small dictionaries for a distributed attack since a single machine can usually process them immediately. Therefore, I decided to use larger ones. The distributed dictionary attack was done using four dictionaries with sizes from 1.1 GB to 8.3 GB, and the chunk size set to 60 seconds. The correct password was, in all cases, intentionally placed at the end of each dictionary to force Fitcrack process it entirely. The experimental results are shown in Table 3.19, where for each attack, I show the total time, and the number of chunks generated. The hash algorithms used were SHA-1 [97], which is easier-to-calculate, and Whirlpool [188], which is more complex. The experiments compare three versions of Fitcrack. Their implementation details are narrowly described in Section 3.6.5. The naming of the versions is following:

- **Fitcrack-1** — the original version with the old benchmarking,
- **Fitcrack-2** — Fitcrack with the improved benchmarking, and
- **Fitcrack-3** — Fitcrack with the improved benchmarking and pipeline processing.

See that in the first approach, marked as *Fitcrack-1*, the Generator created a single big chunk. Therefore, only one node from the entire network was actually cracking. The reason for such behavior is the inaccuracy of the hashcat's default benchmark mode. The measured performance is much higher than is possibly achievable with a real attack. This phenomenon is discussed and experimentally tested in Section 3.9.2, concretely Table 3.17. Despite there was only a single node working, Fitcrack was still faster except using the smallest dictionary. Hashtopolis wasted a lot of time by sending the entire dictionary to all eight nodes, even though the smallest one could be processed relatively quickly using just a single node.

SHA-1									
Dictionary		**Fitcrack-1**		**Fitcrack-2**		**Fitcrack-3**		**Hashtopolis**	
size	keyspace	time	ch.	time	ch.	time	ch.	time	ch.
1.1 GB	114,076,081	3m 15s	1	3m 18s	2	2m 40s	2	2m 22s	10
2.1 GB	228,152,161	4m 27s	1	3m 20s	4	3m 28s	4	4m 52s	20
4.2 GB	456,304,321	6m 5s	1	4m 34s	8	4m 2s	8	11m 14s	40
8.3 GB	912,608,641	12m 49s	1	10m 1s	13	4m 58s	16	32m 6s	80
Whirlpool									
Dictionary		**Fitcrack-1**		**Fitcrack-2**		**Fitcrack-3**		**Hashtopolis**	
size	keyspace	time	ch.	time	ch.	time	ch.	time	ch.
1.1 GB	114,076,081	3m 15s	1	3m 36s	2	3m 11s	2	2m 39s	26
2.1 GB	228,152,161	4m 36s	1	3m 25s	4	3m 14s	4	5m 56s	52
4.2 GB	456,304,321	8m 0s	1	4m 0s	8	4m 17s	8	12m 13s	105
8.3 GB	912,608,641	17m 31s	1	8m 42s	15	5m 2s	16	46m 47s	208

Table 3.19: Dictionary attack using 8 nodes, chunk size = 60s

With *Fitcrack-2*, the improved benchmarking technique allowed the scheduling algorithm worked with a much more precise value. Fitcrack was thus able to distribute the work appropriately to the performance of hosts. Naturally, more chunks are created as the benchmarked values were not so high as in the previous case. There was also a significant speedup for bigger dictionaries thanks to better utilization of resources.

The pipeline processing introduced in *Fitcrack-3* helped to eliminate the overhead for wordlist transfer. With the 8.3 GB dictionary, the hosts were processing the first chunk and downloading the next one simultaneously. As a result, the total time required to complete the job became much shorter.

To illustrate the influence of the wordlist size on the overall cracking time, I depicted the results in graphs. Figure 3.32 displays the total time based on the dictionary size for the SHA-1 algorithm. Figure 3.33 shows the same for the Whirpool case. The trends of both charts are similar.

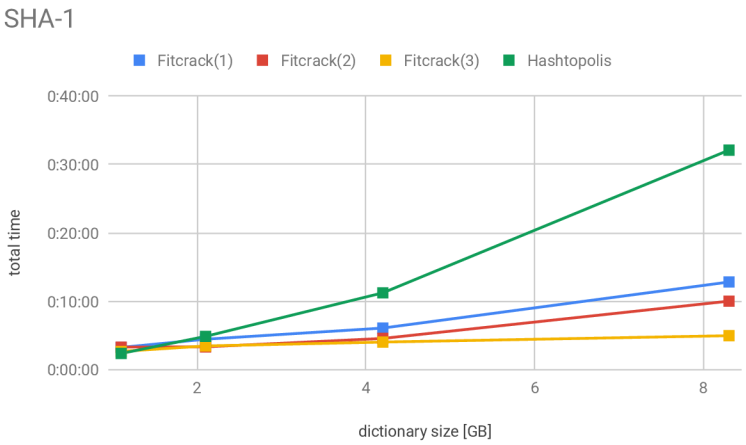


Figure 3.32: Total time of dictionary attack on SHA-1

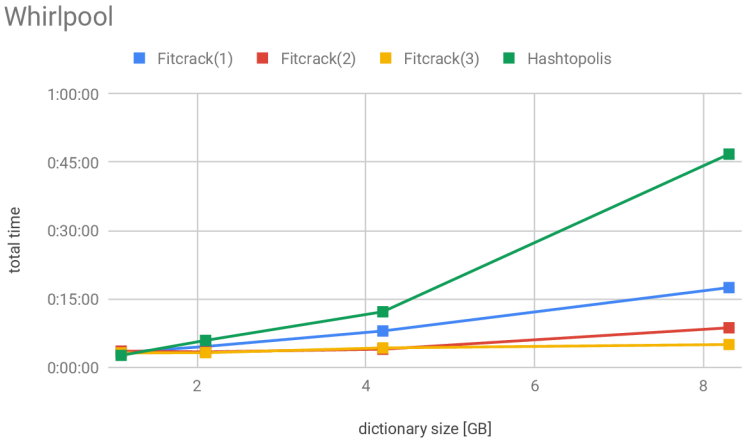


Figure 3.33: Total time of dictionary attack on Whirlpool

The above-described upgrades of Fitcrack's Generator reduced the overhead dramatically. Since HTTP(S) has no broadcast by design, all server-host data transfers are performed using one unicast connection per host. With the increasing dictionary size (d_s), the link to the server soon becomes a bottleneck. In Fitcrack, this is solved by fragmentation which ensures each candidate password is transferred only once. So that the required amount of data to transfer equals to d_s. Hashtopolis, however, sends the entire dictionary to all hosts. For N cracking nodes, we need to transfer $N * d_s$ bytes of useful data plus the overhead of network protocols. In our case, N equals 8. So that, for 4.2 GB dictionary, it is necessary to transmit $8 * 4.2$ GB $= 33.6$ GB of data. For 8.3 GB dictionary, the amount of transferred data equals $8 * 8.3$ GB $= 66.4$ GB, etc. Thus, cracking with the 4.2 GB dictionary took around 4 minutes for Fitcrack, while Hashtopolis required between 11-12 minutes. For the 8.3 GB dictionary, the difference is even more significant – 5 minutes for Fitcrack and 32-47 minutes for Hashtopolis where most of the time is spent by data transfer. The distribution strategy used has a vast impact on scalability since $\lim_{N\to\infty}(d_s) = d_s$, but $\lim_{N\to\infty}(N * d_s) = \infty$ which makes the naive approach practicaly unusable for larger networks and bigger dictionaries.

3.9.4 Distributed Brute-force Attack

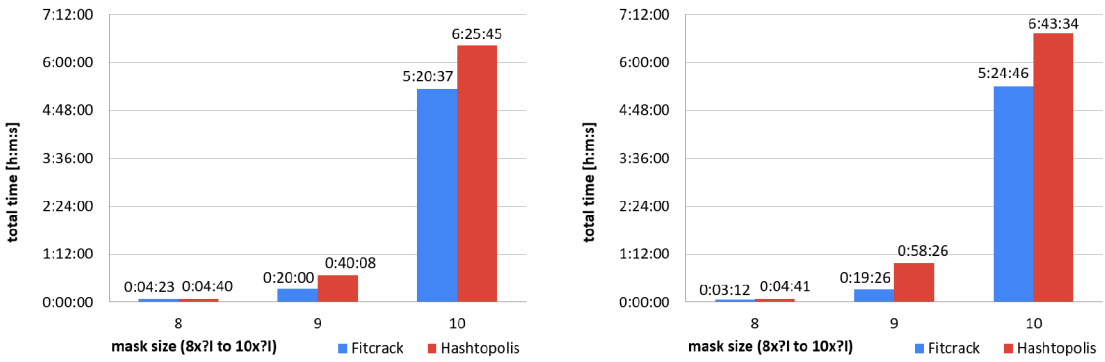
While dictionary attacks may have high requirements for data transfers between nodes, the brute-force attack is not traffic-extensive at all. With each workunit, we only need to send hosts an attack configuration [87], and the indexes of candidate passwords. To analyze how the proposed scheduling strategy behaves in comparison with Hashtopolis, I performed a series of brute-force attacks on SHA-1 [97] using masks from 8 lowercase Latin letters (`?l?l?l?l?l?l?l?l`) to 10 letters (`?l?l?l?l?l?l?l?l?l?l`). The chunk size was set to the following values: 60s, 600s, 1200s, and 1800s. To get comparable results, I let both tools get through the entire keyspace. This was ensured by using two input hashes – one which was crackable using the mask, and another uncrackable one. Table 3.20 shows the total time and the number of generated chunks.

We can see again the behavior of the scheduling algorithm described in Section 3.6.5. Due to the workunit shrinking in the initial and final phases, Fitcrack generated significantly more chunks than Hashtopolis. Obviously, this behavior is not contributive if we use very

Configuration			Fitcrack		Hashtopolis	
Chunk size	**Mask**	**Keyspace**	**Time**	**Ch.**	**Time**	**Ch.**
60s	8x?l	208,827,064,576	4m 38s	5	2m 37s	3
	9x?l	5,429,503,678,976	21m 26s	147	16m 52s	59
	10x?l	141,167,095,653,376	9h 4m 49s	5122	6h 36s 41s	1654
600s	8x?l	208,827,064,576	4m 19s	5	4m 46s	1
	9x?l	5,429,503,678,976	20m 12s	103	20m 7s	6
	10x?l	141,167,095,653,376	5h 24m 25s	480	6h 53m 43s	152
1200s	8x?l	208,827,064,576	4m 23s	5	4m 40s	1
	9x?l	5,429,503,678,976	20m 0s	101	40m 8s	3
	10x?l	141,167,095,653,376	5h 20m 37s	402	6h 25m 45s	76
1800s	8x?l	208,827,064,576	3m 12s	1	4m 41s	1
	9x?l	5,429,503,678,976	19m 26s	100	58m 26s	2
	10x?l	141,167,095,653,376	5h 24m 46s	388	6h 43m 34s	51

Table 3.20: Brute-force attack on SHA-1 using 8 nodes, different chunk sizes

small chunks – for 60s, Hashtopolis was always faster. However, with larger chunk sizes, Fitcrack achieved better distribution. The workunit distribution progress for the attack with 600s chunks and the 10-letter mask is displayed in Figure 3.30. In the main phase, the workunit assignment is standard, however, in the final phase, the count is increased, and the sizes are smaller, ensuring all nodes are utilized in every moment. Especially for chunk sizes of 1200 and 1800 seconds, the cracking times using Fitcrack were much lower. Figure 3.34 displays the comparison. What also helped make Fitcrack faster was the pipeline workunit processing, described in Section 3.6.5, which almost eliminated the communication overhead. The hosts were thus able to start the next workunit immediately after the previous was finished.



(a) chunk size: 1200s

(b) chunk size: 1800s

Figure 3.34: Mask attack on SHA-1 on 8 nodes

While in dictionary attacks, the password inputs of hashcat are processed, cached, and loaded to GPU, in a brute-force attack, such operations are not required. The passwords are generated directly on the GPU, and thus it is possible to achieve much higher cracking speed. The network bandwidth is not limiting since we only transfer a range of password indexes together with additional options. The communication overhead can be, again, reduced by

the pipeline workunit assignment. However, to utilize hardware resources well, it is required to choose the keyspace of workunits wisely. Both Fitcrack and Hashtopolis calculate the keyspace from a cracking performance obtained by the benchmark and a user-defined chunk size. Hashtopolis preserves the similar keyspace to all workunits, which, as I detected, leads to shorter cracking times of less-complex jobs if the chunk size is set to a smaller value. Fitcrack, on the other hand, employs the adaptive scheduling algorithm, which modifies the keyspace of workunits depending on the current progress. If the user-defined chunk size is big enough, the strategy used in Fitcrack helps reduce the total cracking time even if the total number of chunks is higher than in Hashtopolis.

3.9.5 Distributed Combination and PRINCE Attacks

This section covers a series of experiments with the classic combination attack and advanced combination attack called PRINCE. The classic combination attack described in Section 3.8.2 combines passwords from two dictionaries: left and right. Their concatenation is the resulting password. Probability infinite chained elements or PRINCE, on the other hand, uses only a single dictionary. Depending on the desired length of the output, it constructs chains - sequences of word lengths. Using various additional settings, it substitutes positions in these chains with the dictionary passwords, which generates password guesses. The details of PRINCE are described in Section 3.8.6.

Input Hashes

The experiments in this section purposedly use two algorithms with different computing complexity: BCrypt [167] and SHA-1 [97]. The target hashlist comprises the top 100 most used passwords from the LinkedIn leak hashed by BCrypt or SHA-1, respectively. To provide an overview of their complexity, I performed a series of performance measurements using just hashcat, version 5.1.0, which is the same as used with Fitcrack for this series of experiments. The results are shown in Table 3.21. I compare the values gathered from hashcat's benchmarking mode using the `--benchmark` option with actual attacks' performance. For BCrypt, the benchmarking mode does not allow to specify the cost factor. The fixed value is 5, which corresponds to $2^5 = 32$ iterations. In the experiments, I also use the cost factor of 12 as it is the default value in several libraries. The number iterations is thus $2^{12} = 4,096$, which is much more difficult to crack. In the experiments, the SHA-1 is unsalted. For each password, the hash only needs to be computed once, and testing the 100 different hashes is just string comparison. Therefore, testing the entire hashlist is not significantly more difficult than testing a single hash. The BCrypt, on the other hand, uses cryptographic salt, so that the entire algorithm needs to be computed 100 times for each candidate password. I did a short test with a brute-force attack with `?l?l` mask on NVIDIA GTX 1050 Ti. Testing the 676 possible password guesses against a single hash requires about 1 minute and 20 seconds. Testing the same amount of passwords against all 100 hashes takes about 2 hours.

Algorithm	Benchmark	Brute-force	Dictionary	Combination	PRINCE
BCrypt(05)	4,121	3,790	3,829	3,730	3,541
BCrypt(12)	-	32	32	32	32
SHA-1	$2{,}530 \cdot 10^6$	$1{,}580 \cdot 10^6$	$13 \cdot 10^6$	$1{,}285 \cdot 10^6$	$2 \cdot 10^6$

Table 3.21: The performance of hashcat on BCrypt and SHA-1

Moreover, the results discover an interesting phenomenon. While for SHA-1, the cracking performance highly depends on the attack mode, for BCrypt with cost 12, it is the same in all cases. As discussed and experimentally verified in Section 3.9.2, the benchmarking mode provides only the maximal theoretically achievable performance and does not take generating password guesses into account. Since SHA-1 is relatively easy-to-compute, the bottleneck is the password input. Therefore, the brute-force attack is more than 100 times faster than the dictionary attack. The combination attack is also very fast since it only needs to load passwords only once. The slowest attack mode was PRINCE, as it requires an external password generator. BCrypt(12), on the other hand, is so computationally complex that it does not matter how the passwords are created. With all attack modes, the GPU was not able to crack more than 32 hashes per second.

Attacks on the BCrypt Algorithm

The first series of experiments compare the cracking time and the efficiency of dictionary, combination, and PRINCE attacks on the computationally complex BCrypt algorithm with the cost factor of 12. As written above, the hashlist contained 100 unique salted hashes. The keyspace of all jobs was 10,000 password guesses based on combinations of strings from the `adobe100.txt` dictionary from Daniel Meissler's SecLists[58] repository. The PRINCE attack used the original dictionary with 100 passwords, the maximum password length of 30 characters, and the maximum of 2 elements in a chain (see Section 3.8.6). For the combination attack, the `adobe100.txt` dictionary was on both left and right. Therefore, the total number of combinations was $100 \cdot 100 = 10,000$. The dictionary attack used a specially-created wordlist made of all these left-right combinations. All three jobs were performed on 1, 2, and 4 hosts with a single NVIDIA GTX 1050 Ti GPU. The goal was to measure the attack efficiency and the time Fitcrack needs to process the entire keyspace.

	Dictionary		**Combination**		**PRINCE**	
Hosts	**Efficiency**	**Time**	**Efficiency**	**Time**	**Efficiency**	**Time**
1	99 %	13h 1m	99 %	20h 6m	93 %	9h 48m
2	94 %	5h 9m	98 %	10h 53m	93 %	5h 18m
4	91 %	2h 41m	91 %	4h 41m	90 %	2h 43m

Table 3.22: Dictionary, Combination, and PRINCE attacks on BCrypt

Table 3.22 shows the experimental results. Since BCrypt is a very complex algorithm, the hosts were highly utilized most of the time. No host was noticeably being blocked by waiting for new passwords. Therefore, the efficiency of all attacks was above 90 %. The cracking time, also illustrated by the chart in Figure 3.35, scaled pretty well with the number of nodes. We see the trend is close to linear. The only higher-than-expected time is for the dictionary attack on a single node. An interesting observation is the influence of the password order in the dictionary. The experiment used an unsorted wordlist. After sorting the input dictionary, the cracking time was reduced from 13 hours and 1 minute to 9 hours and 42 minutes.

[58] https://github.com/danielmiessler/SecLists/tree/master/Passwords/Leaked-Databases

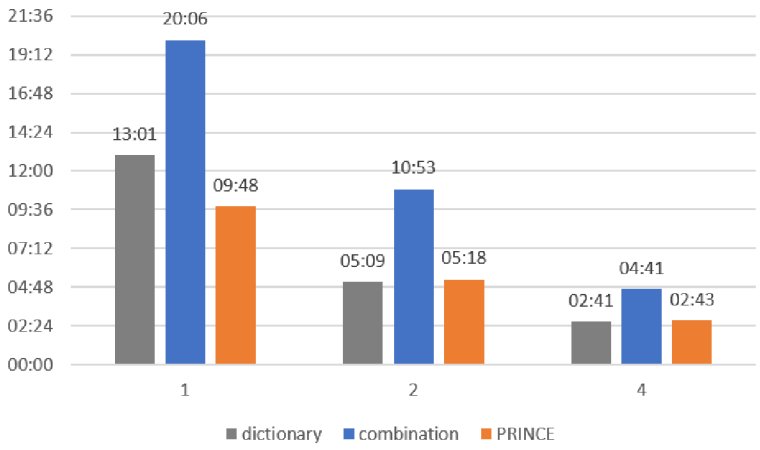


Figure 3.35: Cracking time of different attacks on BCrypt

Attacks on the SHA-1 Algorithm

The following series of experiments aim at the less computationally-complex SHA-1 algorithm. Since we can crack much more SHA-1 hashes than BCrypt hashes per second, the jobs have higher keyspace. The input dictionary for the PRINCE attack was `phpbb.txt` concatenated with `honeynet.txt` from the same repository. The maximum password length was again set to 30 characters with the maximum of 2 elements in a chain. For the combination attack, the same dictionary was on both left and right, as in the previous case. However, since the total keyspace of combined passwords is about $168 \cdot 10^9$, the experiments do not test the dictionary attack because the prepared dictionary would have over 1 TB of size. The jobs were performed on 1, 2, 4, and 8 hosts with NVIDIA GTX 1050 Ti GPU.

Table 3.23 shows the results. The cracking speed was much higher in comparison with BCrypt. Therefore, supplying hosts with enough passwords was more time-critical than the actual cracking. For the combination attack, the password transfer became a bottleneck. And thus, the efficiency is much lower than with BCrypt.

	Combination		**PRINCE**	
Hosts	**Efficiency**	**Time**	**Efficiency**	**Time**
1	53 %	3h 46m	99 %	1d 0h 2m
2	38 %	2h 35m	98 %	11h 37m
4	28 %	2h 31m	96 %	6h 56m
8	22 %	2h 28m	74 %	3h 38m

Table 3.23: Combination and PRINCE attacks on SHA-1

The Scalability of PRINCE

Unlike the classic dictionary and combination attacks, the PRINCE attack's efficiency is not influenced by real-time dictionary transfers. Therefore, PRINCE is highly efficient even with less complex cryptographic algorithms. We can clearly see this in Table 3.23, showing the results of the previous scenario. I guess the only loss of efficiency may occur if we assign to many nodes for a task, as discussed in Section 3.9.1. Another measurement shown in

Table 3.24 further validates my assumptions. The attack on the complex BCrypt algorithm used the `adobe100.txt` dictionary of 100 passwords, resulting in $10 \cdot 10^3$ password guesses. The attack on SHA-1 used the `rockyou.txt` with $14 \cdot 10^6$ passwords, and the total number of password guesses was thus $33 \cdot 10^9$. The maximum password length of both jobs was 24 characters for BCrypt and 30 characters for SHA-1. Both jobs had the minimum and maximum of 2 elements in chain. The desired workunit processing time was 3600 seconds. We see that in most cases, the efficiency was above 90 %. It started to drop (indicated by *) with four hosts on the first job and eight hosts on the second job. We also see the decrease in the total cracking time is much lower here. From this point, adding another nodes makes no extra benefit. A very similar case is seen with the brute-force attack in Figure 3.28(b), where the "scalability cap" for efficient computing was 16 nodes.

		1 Host		**2 Hosts**		**4 Hosts**		**8 Hosts**	
Algo	**Keys.**	**Eff**	**Time**	**Eff**	**Time**	**Eff**	**Time**	**Eff**	**Time**
BCrypt	$10 \cdot 10^3$	99 %	9h 41m	93 %	5h 18m	55 %*	4h 25m	-	-
SHA-1	$33 \cdot 10^9$	99 %	5h 26m	94 %	2h 44m	90 %	1h 27m	74 %*	54m

Table 3.24: The "scalability cap" of PRINCE attacks

While the previous experiment explains there is always a limit on node count, it raises another question: What is the influence of the desired workunit processing time? Theoretically, for a task complex enough, the value should not dramatically impact efficiency, especially with the pipeline processing enabled (see Section 3.6.5). Table 3.25 shows two jobs with the desired workunit times set to 600, 1800, 3600, and 7200 seconds. The first one with BCrypt used the `adobe100.txt` dictionary and the maximum password length of 17 with 1 to 4 elements in a chain. The total keyspace was $81 \cdot 10^3$. The attack on the SHA-1 used the maximum password length of 8 and 1 to 4 elements in a chain. The resulting keyspace was $579 \cdot 10^9$. The jobs were processed on 8 hosts with NVIDIA GTX 1050 Ti GPU. Both assignments took about 9 hours to complete. For all configurations, the efficiency was between 97 to 99 %, which proves my hypothesis, as changing the workunit time configuration had no adverse effect in such a complex task.

Workunit time:		600 s		1800 s		3600 s		7200 s	
Algo.	**Keysp.**	**Eff**	**Time**	**Eff**	**Time**	**Eff**	**Time**	**Eff**	**Time**
BCrypt	$81 \cdot 10^3$	99 %	9h 46m	98 %	9h 44m	97 %	9h 48m	98 %	9h 49m
SHA-1	$579 \cdot 10^9$	99 %	8h 45m	98 %	8h 40m	97 %	8h 47m	98 %	8h 42m

Table 3.25: 8 hosts, different workunit processing times

PRINCE Comparison with Hashtopolis

Finally, Table 3.26 compares the PRINCE's performance with Fitcrack and Hashtopolis. The first three jobs aimed to test a network with the server and only a single host. The last job employed ten nodes with NVIDIA GTX 1050 Ti. The first column shows the algorithm. The following describe the attack's configuration: the used dictionary, the minimal and maximal length of passwords, and the minimal and maximal number of elements in a chain. Next columns display the resulting total keyspace and the number of hosts that participate on the job. Finally, last two columns display the the total processing time using Fitcrack and Hashtopolis, respectively. Apparently, Fitcrack is slightly more efficient. I assume this

is because the Fitcrack has native support and optimizations for the PRINCE attack, while the Hashtopolis uses the general "external password generator" mode.

Algo.	Dictionary	Pass.	Chain	Keysp.	Hosts	Fitcrack	Hashtopolis
SHA-1	rockyou.txt	1–6	1–4	$2 \cdot 10^9$	1	29m 22s	29m 25s
SHA-1	rockyou.txt	1–9	1–2	$24 \cdot 10^9$	1	4h 9m	4h 12m
BCrypt	adobe100.txt	1–24	2–2	$10 \cdot 10^3$	1	5h 56m	6h 36m
SHA-1	rockyou.txt	1–8	1–4	$579 \cdot 10^9$	10	8h 47m	8h 53m

Table 3.26: PRINCE attack using Fitcrack and Hashtopolis

3.9.6 Summary

Distributed computing undoubtedly has its place in the area of password cracking. The performance achievable with a single machine is always limited, even we equip it with multiple GPUs. As the experiments showed, not every cracking task is, however, worth distribution. Yet, if it is, there is always a reasonable number of computing nodes we should employ to complete the job at the desired time. While underestimating the task's complexity results in delays, utilizing more machines than necessary may cause needless overhead and efficiency loss. The final decision is up to the system's operator, who needs to consider the complexity of algorithms, attack mode, network characteristics, and other aspects. The choice, however, does not need to be based on manual calculations. Smart systems should provide a hint to help users create an assignment that is appropriate to their needs. Therefore, the Fitcrack system provides an estimation of the maximum cracking time when a job is created.

The workload distribution can use multiple schemes. Fitcrack uses dynamic chunk distribution with the progressive assignment of keyspace. The system employs the adaptive scheduling algorithm to create fine-tailored workunits that match each hosts' current performance. As experimentally verified, the mechanism allows the computers to process tasks efficiently and handles unexpected events like a sudden change in a node's performance. The most tricky part is to choose an appropriate size for the first workunit. The calculation relies on the accuracy of benchmarking. While the original version was very imprecise, the new benchmarking technique is far more accurate.

The proposed distribution strategies for different attack modes work as intended and allow the system to precisely control the keyspace's distribution between computing nodes. Yet, there are differences since each attack mode has specific properties. The influence of the attack mode on the overall efficiency and scalability depends on the concrete algorithm. For complex hashes like BCrypt, the cryptographic calculations are so complex that attack mode has almost no influence. For easier-to-compute algorithms, the situation is different, and the speed of password creation is the main bottleneck. The faster we get new passwords, the higher the cracking speed.

The brute-force attack provides the best performance since we generate the passwords directly on the GPU. The network utilization is low because there is no need to transfer additional data. The situation is rather different in dictionary-based attacks. The classic dictionary attack is only worth distribution if the input wordlist has enough passwords. But if so, we need to transfer a large amount of data from the server to hosts. Therefore, network bandwidth plays an essential role in overall efficiency. Hashtopolis uses a simple strategy to transfer the entire dictionary to all clients before the cracking even starts. Such a method

is easy-to-implement, but since there is no broadcast in HTTP, it may require a lot of time, and the scalability is terrible. The bigger the dictionary, the higher the overhead. The link to the server becomes the bottleneck, and the more nodes we have, the longer we need to wait. Fitcrack, on the other hand, sends only fragments of the dictionary so that each node receives only the passwords it needs. While it was required to put additional logic to the system, the initial overhead is much lower, and the cracking session in Fitcrack can start sooner. Adding the support fork pipeline processing eliminated the communication overhead almost entirely.

While Fitcrack supports three different dictionary-based attack modes, the PRINCE seems to be the most low-cost one in terms of network utilization. The main advantage is that it does not require a real-time dictionary transfer. The input dictionary sent with the first workunit is usually small, and all the combinations are created on the client-side. Like with the brute-force attack mode, the server controls the distribution process just by defining the index ranges for each workunit. The classic dictionary attack, if performed in a distributed way, is beneficial for complex cryptographic algorithms. For simple algorithms, a single-machine session may be faster if the dictionary is not exceptionally big. In terms of efficiency, the combination attack is in the middle between the dictionary and PRINCE. The amount of data that needs to be transferred to hosts is lower than with the classic dictionary attack but higher than with PRINCE. The combination attack may become efficient for moderate as well as for complex algorithms. In all cases, I suggest that the decision about the employed solution to use should reflect the algorithm's complexity.

Chapter 4

Probabilistic Password Models

While machine-generated passwords are more or less random, if the choice is up to a human being, the situation is different. Years of research on leaked datasets of passwords showed they provide a valuable source of knowledge. Probabilistic methods utilize that knowledge to allow a better targeting of password cracking attacks.

This chapter analyzes existing probabilistic password models and discusses their characteristics. The core of the chapter focuses on probabilistic context-free grammars and their employment in password cracking tasks. In the following sections, I describe the properties of state-of-the-art methods for grammar-based cracking and identify weak spots that complicate their use from the practical point of view. I introduce a series of enhancements that improve existing concepts and allow guessing more passwords for the same time. Moreover, I show how to reduce the total number of password guesses without significantly impacting the success rate. I also propose a methodology for distributed password cracking with probabilistic grammars. Together with my fellow researchers, I created two proof-of-concept tools that demonstrate the discussed principles. Last but not least, I show the benefits of the proposed methods in a series of practical experiments.

4.1 Motivation for Smart Password Guessing

Confidential data and user accounts for various systems and services are protected by passwords. Though a password is usually the only piece that separates a potential attacker from accessing the privileged data, users tend to choose weak passwords which are easy to remember [30]. In reaction, system administrators and software developers introduce mandatory rules for password composition, e.g., "use at least one special character." While password-creation policies force users to create stronger passwords [166, 207], recent leaks of credentials from various websites showed the reality is much more bitter. People widely craft passwords from existing words [63] and often reuse the same password between multiple sites [52]. This fact may be utilized by both malicious attackers and forensic investigators who seek for evidence in password-protected data.

4.1.1 The Downside of Traditional Methods

Traditional ways of password cracking contain a *brute-force* attack where one tries every possible sequence of characters upon a given alphabet, and a *dictionary attack* where one uses a list of existing passwords and tries each of them. The main drawback of the brute-force attack is a big *keyspace* (a number of all possible password candidates), which grows

exponentially with the length of the password, and one does not need to "try everything" to crack the password. The dictionary attack, on the other hand, usually checks a limited number of commonly-used or previously-leaked passwords.

With the complexity of today's algorithms, it is often impossible to crack a hash of a stronger password in an acceptable time using the tradidional methods. For instance, verifying a single password for a document created in MS Office 2013 or newer requires 100,000 iterations of SHA-512 algorithm. Even with the use of the popular *hashcat* tool and a machine with 11 NVIDIA GTX 1080 Ti[1] units, brute-forcing an 8-character alphanumeric password may take over 48 years. Even the most critical pieces of forensic evidence lose value over such a time.

4.1.2 The Potential of Probabilistic Models

Over the years, the use of probability and statistics showed the potential for a rapid improvement of attacks against human-created passwords [136, 213, 114]. Various leaks of credentials from websites and services provide an essential source of knowledge about user password creation habits [35, 212], including the use of existing words [63] or reusing the same credentials between multiple services [52]. Therefore, the ever-present users' effort to simplify work is also their major weakness. People across the world unwittingly follow common password-creation patterns over and over.

One approach is the use of *Markov chains* which consider probabilities that a certain character will follow after another one. The probabilities are learned from an existing password dictionary and then reused for generating password guesses [136]. The method, however, only works with individual characters and does not consider digraphs or trigraphs. To work with larger password fragments, Weir et al. proposed the use of *probabilistic context-free grammars* (PCFG) that can describe the structure of passwords in an existing (training) dictionary. Fragments described by PCFG represent finite sequences of letters, digits, and special characters. Then, by derivation using rewriting rules of the grammar, one can not only generate all passwords from the original dictionary, but produce many new ones that still respect password-creation patterns learned from the dictionary [213].

4.2 Related Work

For a long time, probability and statistics have been applied to measure password strength [166, 207, 104] and generate guesses in password cracking [136, 213, 114, 80]. Major password leaks allowed to make a clearer image of how user create their passwords [35]. Such knowledge has been utilized in multiple password cracking principles and adopted to existing tools.

4.2.1 Early Work

The use of probabilistic methods for computer-based password cracking dates back to 1980. Martin Hellman introduced a time-memory trade-off method for cracking DES cipher [78]. This chosen-plaintext attack used a precomputation of data stored in memory to reduce the time required to find the encryption key. With a method of distinguished points, Ron Rivest reduced the necessary amount of lookup operations [174]. Phillipe Oechslin improved the original concept and invented rainbow tables as a compromise between the brute-force

[1] https://onlinehashcrack.com/tools-benchmark-hashcat-gtx-1080-ti-1070-ti

attack and a simple lookup table. For the cost of space and precomputation, the rainbow table attack reduces the cracking time of non-salted hashes dramatically [142].

4.2.2 Markovian Models

The origin of passwords provides another hint. Whereas a machine-generated password may be more or less random, human beings follow specific patterns we can describe mathematically. *Markov chains* are stochastic models frequently used in natural language processing [169]. Narayanan et al. showed the profit of using zero-order and first-order Markovian models based on the phonetical similarity of passwords to existing words [136]. While in the classic incremental brute-force attack, all characters in the alphabet are equal to each other, the Markovian model assigns probability values to different characters.

Mathematically, we can describe the models using *probabilistic finite automata* [168]. In a zero-order model, future states rely neither on the current nor on the previous states. Password guessing based on a zero-order model means we use more probable characters first, but do not look at the already-generated ones. A first-order model meets the Markov property: the next state depends only on the current state. Higher-order models utilize one or more previous states as well [169, 136].

For Markov-based password cracking, commonly-used is the first-order model. The method uses conditional probability $P(A|B)$ that character A will follow after character B. The probabilities for all characters A, B are stored in a matrix obtained by the analysis of an existing password dictionary [136]. The technique was utilized in *Hashcat* tool which uses Markov chains for brute-force attacks by default. The probability matrix can be generated automatically using *Hcstatgen*[2] utility and is stored in a *.hcstat* file. Recent versions of Hashcat use LZMA compression which is indicated by *.hcstat2* file extension. A modified model based on 3-grams is also employed in the Incremental attack mode of John the Ripper tool [57].

One can also define a threshold value that limits the depth of character lookup. The threshold can be a probability value or just an integer specifying how many most probable passwords to use at each position. Thresholding rapidly decreases the keyspace of the attack because we only use a part of all possible password guesses. In hashcat, the threshold can be specified using the `--markov-threshold` option.

There is also an extended three-dimensional model where the next character relies not only on the current state but also on the position in the password. The technique uses a 3D matrix created by putting multiple 2D matrices together - one for each position. Hashcat uses the per-position 3D Markov by default. A user can switch to classic 2D Markov guessing by specifying the `--markov-classic` option.

For practical cracking, the probability order of characters is more than the actual probability values. We can thus omit the actual values and create a matrix by placing characters from the most probable to the least probable one. Figure 4.1 shows an example of a matrix with threshold set to 3. In case of mask `?l?l?l`, the keyspace would be $26*26*26 = 17576$, since there are 26 lowercase letters in the latin alphabet. However, which threshold set to 3, the keyspace is $3*3*3 = 27$, since on each position, only three characters are used.
The candidate passwords for mask `?l?l?l` and threshold 3 (indicated by the vertical line) are generated in the following order: `bed`, `bec`, `bet`, `bad`, `bat`, `bar`, ... Note that password `bez` is not generated since `z` is on the position 4 in *e*-row, and $4 > 3$. In hashcat, the threshold can be specified using the `--markov-threshold` option.

[2] https://hashcat.net/wiki/doku.php?id=hashcat_utils#hcstatgen

$$
\begin{array}{c}
\varepsilon \\ a \\ b \\ c \\ d \\ e \\ \vdots
\end{array}
\left[\begin{array}{ccc|cccc}
b & n & e & g & a & u & \dots \\
d & t & r & n & d & v & \dots \\
e & a & r & u & o & i & \dots \\
k & i & e & o & u & a & \dots \\
o & m & a & y & r & p & \dots \\
d & c & t & z & d & n & \dots \\
\vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \ddots
\end{array}\right]
$$

Figure 4.1: Example of Markov matrix with threshold set to 3

In 2015, Duermuth et al. presented *Ordered Markov ENumerator* (OMEN), a password guessing algorithm they claim to outperform all previous publicly available Markov-based password guessers. The algorithm discretizes all probabilites into a number of bins, and iterates over the bins in an order of decreasing likelihood. For each bin, it generates all passwords that match the associated probability. Duermuth et al. state that unlike existing algorithms, the "OMEN can output guesses in order of (approximate) decreasing frequency" [57]. While in hashcat, the brute-force attack is always related to a fixed password length defined by the mask, OMEN incorporates the probability of password length as well.

4.2.3 Probabilistic Grammars

Weir et al. introduced password cracking using *probabilistic context-free grammars* (PCFG) [213]. The mathematical model is based on classic context-free grammars [70] with the only difference that each rewrite rule is assigned a probability value [19, 94]. The grammar is created by training on an existing password dictionary. Each password is divided into continuous fragments of letters (L), digits (D), and special characters (S). For fragment of length n, a rewrite rule of the following form is created: $T_n \rightarrow f : p$, where T is a type of the character group (L, D, S), f is the fragment itself, and p is the probability obtained by dividing the number of occurrences of the fragment by the number of all fragments of the same type and length. In addition, we add rules that rewrite the starting symbol (S) to *base structures* which are non-terminal sentential forms describing the structure of the password [213]. For example, password "p@per73" is described by base structure $L_1S_1L_3D_2$ since it consist from a single letter followed by a single special character, three letters, and two digits. Table 4.1 shows rewrite rules of a PCFG generated by training on two passwords: "pass!word" and "love@love". Fragment "love" has a higher probability value than "pass" or "word" because it occurred twice in the training dictionary, whereas the others only once. There is only one rule that rewrites S since both passwords are described by the same base structure. By using PCFG on MySpace dataset (split to training and testing part), Weir et al. were able to crack 28% to 128% more passwords in comparison with the default ruleset from *John the Ripper* (JtR) tool[3] using the same number of guesses.

The proposed approach, however, does not distinguish between lowercase and uppercase letters. Weir extended the original generator by adding capitalization rules like "UULL" or "ULLL" where "U" means uppercase and "L" lowercase. The rules are applied to all letter fragments which increases the number of generated guesses [211]. After adding capitaliza-

[3] https://www.openwall.com/john/

left	→	right	probability
S	→	$L_4 S_1 L_4$	1
L_4	→	pass	0.25
L_4	→	word	0.25
L_4	→	love	0.5
S_1	→	@	0.5
S_1	→	!	0.5

Table 4.1: An example of PCFG rewrite rules

tion, the notation for letter non-terminals were changed from L_n to A_n (as alphabetical or "alpha" characters) since L now stands for lowercase.

While the previous techniques consider only the syntax of passwords, Veras et al. designed a semantics-based approach which divides password fragments into categories by semantic topics like names, numbers, love, sports, etc. With JtR in *stdin* mode feeded by a semantic-based password generator, Veras achieved better success rates than using Weir's approach or the default JtR wordlist [206].

Ma et al. showed how normalization and smoothing can increase the success rate of Markov models. By training and testing on a huge number of datasets, Ma showed that the improved Markov-based guessing could bring better results than PCFGs [114].

Weir's PCFG-based technique encountered extensions as well. Houshmand et al. introduced keyboard patterns represented by additional rewrite rules that helped improve the success rate by up to 22%, proposed the use of Laplace probability smoothing, and created guidelines for choosing appropriate attack dictionaries [80]. After that, Houshmand also introduced targeted grammars that utilize information about a user who created the password [79].

The research presented in this thesis is based on the late 2018's PCFG Cracker (version 3) that was moved to the Github repository called "legacy-pcfg"[4]. The proof-of-concept parallel and distributed tool I describe in this chapter is compatible with the PCFG Trainer from this exact repository. To prove the contribution of the proposed improvements, I decided not to mix PCFG with Markovian models, and thus when one creates a grammar, they need to set the `--coverage 1.0` option of the PCFG Trainer.

After publishing a part of my research related to the PCFG [82], Weir replaced released a new[5] version 4 of the PCFG Cracker. He replaced the original Markov guessing algorithm by OMEN [57], added the *Password scorer* tool that calculates the strength of existing passwords, and *Prince-Ling Wordlist Generator* that creates customized wordlists for use in attacks based on the PRINCE algorithm.

4.2.4 The PCFG Cracker

The current version of Weir's PCFG Cracker consists of two separate tools: PCFG Trainer and PCFG Manager. While *PCFG Trainer* is used to create a grammar from an existing password dictionary, *PCFG Manager* generates new password guesses from the grammar - i.e., gradually applies rewrite rules to the starting symbol and derived sentential forms.

At the time of writing this thesis, both tools include the support for letter capitalization rules [211], keyboard patterns [80], as well as the ability to generate new password segments

[4] https://github.com/lakiw/legacy-pcfg
[5] https://github.com/lakiw/pcfg_cracker

using Markov chains [136] described in Section 4.2.2. In the training phase, a user can set a *coverage* value which defines the portion of guesses to be generated using rewrite rules only while the rest is generated using Markov-based brute-force. A *smoothing* parameter allows the user to apply probability smoothing as described in [80]. Moreover, the tools contain the support for context-sensitive character sequences like "<3" or "#1" that, if present in the training data, form a separate set of rewrite rules. Such replacements can be used to describe special strings like smileys, arrows, and others.

4.2.5 Motivation for Improvement

Despite numerous improvements made by Houshmand [80], users still have to face slow password guessing speed which is currently the bottleneck of the entire process. Besides, the generating of password guesses gets progressively slower as the time goes on and, as I detected, has high memory requirements.

While the creation of a probabilistic grammar is fast and straightforward (see Section 4.4.1), the application of rewriting rules takes a significant amount of processor time, and the number of generated passwords is overwhelming in comparison with the original dictionary. Creating a complete wordlist of possible password candidates using PCFGs trained on leaked datasets may take many hours or even days. For example, using Weir's tool[4] with a PCFG trained on 6.5 kB *elitehacker*[6] dataset (895 passwords) generates a 12 MB dictionary with 1.8 million passwords. However, using 73 kB *faithwriters* dataset (8,347 passwords) generates a 28 GB dictionary with over 3 billion passwords. Even more unpleasant is the time required to generate such datasets. The first 10 and first 100 passwords of *darkweb2017*[7] dataset and *darkweb2017-top100* can be both used for training and generating within 1 minute on Core(TM) i7-7700K CPU. Taking first 1000 passwords requires more than a day to generate guesses on the same processor.

Moreover, the version 3 of the PCFG Manager tools did not provide information about the *keyspace*, i.e., the number of possible password candidates, and thus the user had no clue about how long the guessing took. Thus, I decided to make PCFG-based password cracking suitable for practical use and propose methods how to guess password faster, transform PCFGs to more compact ones, and calculate an exact number of possible password guesses, aka the keyspace. It is, however, necessary to mention, that later, Weir extended the tool with status reports that contain an approximate calculation of keyspace.

In 2019, Weir also released[8] a compiled PCFG password guesser to achieve faster cracking and easier interconnection with existing tools like hashcat and JtR. Making the password guessing faster, however, resolves only a part of the problem. Serious cracking tasks often require to use distributed computing. But how to efficiently deliver the password guesses to different nodes? Weir et al. suggested the possible use of preterminal structures directly in a distributed password cracking trial [213]. To verify the idea, I decided to analyze the possibilities for distributed PCFG guessing, create a concrete design of intra-node communication mechanisms, and experimentally test its usability.

[6] https://wiki.skullsecurity.org/index.php?title=Passwords
[7] https://github.com/danielmiessler/SecLists/tree/master/Passwords
[8] https://github.com/lakiw/compiled-pcfg

4.3 The Scope of Improvements

Motivated by the findings mentioned above, I focus on making PCFG-based password cracking suitable for practical use. Concretely, to allow the user to create a probabilistic grammar, generate a wordlist of password guesses in a short time, and start cracking immediately. Alternatively, to allow generating guesses and calculating cryptographic hashes simultaneously. To achieve this:

- I created a faster "password generator" that produces more guesses in the same amount of time using the same hardware.
- I automated the calculation of keyspace for a given PCFG. Knowing the keyspace helps to estimate the size of an output dictionary and the time required to generate all password guesses.
- I analyzed and showed how modification of an existing grammar could help the password guessing. Concretely, I described how it accelerates the process and allows it to end in the desired maximum amount of time.
- I designed, created, and evaluated a solution for PCFG-based cracking in a distributed environment.

To verify the success of my efforts, I study the following metrics: a) the number of guesses per time unit, b) the total time of password guessing, c) the number of generated passwords, d) the success rate for testing datasets, i.e., how many newly-generated passwords are present in existing password dictionaries.

4.4 Probabilistic Context-free Grammars (PCFG)

As mentioned in Section 4.2, the mathematical model is based on classic formal context-free grammars [70] with the only difference that each rewriting rule is assigned a probability value [19, 94]. A *probabilistic context-free grammar* G is defined as:

$$G = (N, \Sigma, R, S, P), \tag{4.1}$$

where:

- N is a finite set of nonterminal symbols,
- Σ is a finite set of terminal symbols, $N \cap \Sigma = \emptyset$,
- R a finite set of rewriting (or production) rules $A \to \gamma$, where: $A \in N, \gamma \in (N \cup \Sigma)^*$,
- $S \in N$ is the start symbol of the grammar,
- P is a probability function defined as $P : R \to [0, 1]$, i.e. every rewrite rule is assigned a probability value from 0 to 1. Besides, the sum of probabilites of all rewrite rules $A \to \gamma \in R$ with the same left side A is 1: $\forall A \in N, \sum_{A \to \gamma \in R}(P(A \to \gamma)) = 1$.

The grammar is called *probabilistic* because of the probability function P that assigns each rewriting rule a probability value. The grammar is called *context-free* because we can substitute nonterminal A with the right side γ regardless of the context where A is.

4.4.1 Creating Grammars from Dictionaries

In the password cracking method proposed by Weir et al., the grammar is created from an existing ("training") dictionary. In the original design, every password is divided into continuous fragments od letters (L), digits (D), and special characters (S) [213, 211]. In newer versions of the PCFG Cracker tool, the notation changed to alpha (A), digits (D), and others (O). A probabilistic context-free grammar is created from a password dictionary using the following steps:

- Divide each password to A, D, O fragments. Fragments may have different length. Each fragment is unique; i.e., there are no duplicities. If you receive a fragment that was already created from the current or previous password, do not save it for the second time.
- For each fragment of length n, construct a rewrite rule $T_n \to f$, where $T \in \{A, D, O\}$ is the type of the character set, f is the fragment itself.
- Calculate and set probability of $P(T_n \to f)$ of each rule $T_n \to f$ as:

$$P(T_n \to f) = \frac{c_f}{c_{T_n}}, \tag{4.2}$$

  where c_f is the total occurrence count of fragment f in the training dictionary, and c_{T_n} is the total occurence count of all fragments of type T and length n [211].
- From each password in the dictionary, create base structure B as the sequence of nonterminals $T^1_{n^1} \dots T^k_{n^k}$, where k is the number of nonterminals in this structure. A *base structure* is a sequence of nonterminals T_n that describe from which fragments (tydog and lengths) the password consists. For example, the base structure for password "hello!44mike" is "$A_5O_1D_2A_4$" since the password starts with five letters - alpha symbols (A_5), followed by the exclamation mark (O_1), two digits (D_2), and four more alpha symbols (A_4).
- For each base structure B, construct a rewrite rule: $S \to B$. If the rule already exists, do not add it to the grammar.
- Calculate and set probability $P(S \to B)$ as:

$$P(S \to B) = \frac{c_B}{c_{T_B}}, \tag{4.3}$$

  where c_B is the total number of password that match the base structure B, and c_{T_B} is the total number number of all base structures [211].

An example of grammar creation The above shown algorithm is demonstrated on an example. Assume a dictionary containing the following three passwords:

```
pa$$word42
12mike15
oscar42!!xx
```

From this dictionary, we construct grammar G_1 using the following steps:

- At first, we split the password into fragments. Password "pa$$word42" creates four fragments: "pa" (type: alpha, length: 2), "$$" (type: others, length: 2), "word" (type: alpha, length: 4), and "42" (type: digit, length: 2). Similar splitting goes for the other two passwords. In total, we receive 10 unique password fragments.
- For each unique fragment, we construct a rewrite rule. From "pa" we crete rewriting rule "$A_2 \rightarrow$ pa", "$$" creates "$O_2 \rightarrow$ $$", etc.
- For each rewrite rule, we calculate the probability. For rule "$A_2 \rightarrow$ pa", the probability is 0.5 since it occurred once and the only other A_2 fragment is "xx" which also occurred once. The same calculation goes for other rules, e.g. $P(D_2 \rightarrow 42) = 2/4 = 0.5$, etc.
- We create base structures from all three passwords: "pa$$word42" gives "$A_2O_2A_4D_2$", "12mike15" gives "$D_2A_4D_2$", and "oscar42!!xx" gives "$A_5D_2O_2A_2$".
- For each unique base structure, we construct and add a rewrite rule. For our grammar, we get three rules: "$S \rightarrow A_2O_2A_4D_2$" , "$S \rightarrow D_2A_4D_2$", and "$S \rightarrow A_5D_2O_2A_2$".
- We calculate probabilities for rules that rewrite the start nonterminal to base structures. In our case, each of the three base structures occurs only once, so that their probability is $1/3 = 0.33$ (rounded-off to two decimal places).

In the resulting grammar $G_1 = (N, \Sigma, R, S, P)$,

- $N = \{S, A_2, A_4, A_5, D_2, O_2\}$,
- $\Sigma = \{pa, xx, word, mike, oscar, 42, 15, 12, \$\$, !!\}$,
- R and P shows Table 4.2.

R			P
A	$\rightarrow$	γ	$P(A \rightarrow \gamma)$
S	$\rightarrow$	$A_2O_2A_4D_2$	0.33
S	$\rightarrow$	$D_2A_4D_2$	0.33
S	$\rightarrow$	$A_5D_2O_2A_2$	0.33
A_2	$\rightarrow$	pa	0.5
A_2	$\rightarrow$	xx	0.5
A_4	$\rightarrow$	word	0.5
A_4	$\rightarrow$	mike	0.5
A_5	$\rightarrow$	oscar	1.0
D_2	$\rightarrow$	42	0.5
D_2	$\rightarrow$	15	0.25
D_2	$\rightarrow$	12	0.25
O_2	$\rightarrow$	$$	0.5
O_2	$\rightarrow$	!!	0.5

Table 4.2: The rewrite rules and the probability function of an example PCFG created from three passwords: pa$$word42, 12mike15, and oscar42!!xx

4.4.2 Letter Capitalization

The method of creating a PCFG described in Section 4.4.1 does not distinguish between lowercase and uppercase letters. By analyzing existing leaked passwords, I detected users most often createpasswords just from lowercase letters. If an uppercase letter is used, it is usually at the beginning of the password or of a word contained within it [83], e.g. passwords like "Golf-Mike" or "HelloKitty!" from RockYou[9] dataset. We can utilize the knowledge by mangling the capitalization of letters in existing fragments.

Weir extended the original concept by adding capitalization masks [211]. When creating rewrite rules for alpha fragments $A_n \rightarrow f$, we convert all letters in fragment f to lowercase. For fragment length n, we define one or more capitalization masks. A *capitalization mask* M_n for A_n is a string of length n made of characters U and L. Their position indicates which letters should be uppercase and which lowercase. If U is on position p in the mask, the p-th character in the fragment is an uppercase letter. If L is on position p in the mask, the p-th character in the fragment is a lowercase letter. For every capitalization mask M_n, we the probability $P(M_n)$ of its use as:

$$P(M_n) = \frac{c_{M_n}}{c_{A_n}}, \tag{4.4}$$

where c_{M_n} is the occurence count of all alpha fragments where the capitalization of letters match mask M_n, and c_{A_n} is the total occurence count of all alpha fragments of length n. The sum of $P(M_n)$ for all M_n of length n equals 1. Table 4.3 shows an example of capitalization masks and their probabilities for A_5. For each mask, the table also displays a shortened version that is sometimes used in the literature [211].

capitalization mask	shortened version	probability
LLLLL	L_5	0.928421
UUUUU	U_5	0.041223
ULLLL	U_1L_4	0.021047
UULLL	U_2L_3	0.006215
ULULU	$U_1L_1U_1L_1U_1$	0.003094

Table 4.3: An example of capitalization rules for letter fragments of length 5

It is essential to distinguish between the implementation of the methodology in existing tools and the mathematical basis behind it. In the PCFG Cracker tool, data structures that represent alpha nonterminals are first "rewritten" to sequences of letters, and the capitalization from existing masks is applied after. From the mathematical point of view, we want to preserve the property of a context-free grammar [70]. Thus, we need to consider every letter fragment a nonterminal. For every mask M_n we construct a rewrite rule with probability $P(M_n)$ that rewrites the nonterminal to a sequence of letters with capitalization given by mask M_n. As a result, the total number of rewrite rules increases, we get the ability to create currently previously non-existing sequences of letters while still respecting users' habits for choosing small and big letters.

[9] http://downloads.skullsecurity.org/passwords/rockyou.txt.bz2

4.4.3 Sequential Password Guessing

To gather access to password-protected content, we can generate password guesses directly from an existing PCFG. For this purpose, a *candidate password* (or password guess) is a string generated by the grammar. The string may serve as an input of a cracking tool, or as a part of a new password dictionary for later use. For grammar G, the set of all candidate passwords is the language generated by the grammar [70]:

$$L(G) = \{w \mid S \Rightarrow^* w \wedge w \in \Sigma^*\}. \tag{4.5}$$

Therefore, every password $p \in L(G)$ is derived from the start nonterminal S using a sequence of derivation steps:

$$S \Rightarrow ... \Rightarrow p \tag{4.6}$$

with an application of a finite sequence of rewrite rules $r_1, r_2, \ldots, r_n \in R$. Let $R_p = \{r_1, r_2, \ldots, r_n\} \subseteq R$ be the set of rewrite rules used for generating password p. Then the probability $P(p)$ of password p can be calculated as:

$$P(p) = \prod_{r \in R_p} P(r). \tag{4.7}$$

The *probability of a password* is thus a product of probabilities of all rewrite rules used for its creation.

For PCFGs created from large training dictionaries, it may be computationally impossible to generate and verify all possible candidate passwords in an acceptable time [82]. Therefore, we need to limit the guessing to n most probable passwords. The goal is to find an algorithm that can generate them.

A naive solution is to generate all possible password guesses together with their probabilities, sort them by probability, and select n most probable ones. Though such a solution is possible, it faces the same problem it is trying to resolve.

Preterminal structures

When we perform a series of derivation steps from the start nonterminal, sooner or later, we get a sentential form where all possible derivation steps lead to passwords of the same probability. If this sentential form contains nonterminals, each is rewritable by the rules of an equal probability.

Definition 4 (Preterminal structure) *A preterminal structure is a sentential form for which all further possible derivation steps produce strings of the same probability.*

For each preterminal structure, the probability of all derivable passwords is the same. Its value can be calculated using the P function. Such a fact can be utilized to divide the guessing process into two separates steps. First, we can generate the preterminal structures. After that, we can create password guesses from the structures.

The easiest way is to generate all preterminal structures together with their probabilities, sort them in a probability order, and take a number of the most probable ones to generate password guesses. This approach, however, requires performing many computing steps and

processing a large amount of input data before we can even create the first password. Using this method, it is also not possible to generate preterminal structures and password guesses consequently.

Another approach is to use the technique proposed by Narayanan et al., where we only generate preterminal structures with probability values above a pre-defined limit. A similar principle is in by the John the Ripper tool in "Markov" attack mode. This alternative approach, however, does not guarantee that the password guesses are printed out in decreasing probability order. It only ensures that their probability lies above a defined limit [136].

Alternatively, it is possible to use the *Depth-first search* algorithm. However, it is necessary to process all nodes of the derivation tree eventually [194]. Taking into consideration the sizes of derivation trees of PCFGs created from real password datasets, the number of required *backtracking* operations would be enormous. Therefore, the method is not feasible for practical use with PCFG-based cracking [211].

4.4.4 Probability Groups

Creating a grammar from an existing password dictionary using the algorithm from Section 4.4.1 produces a number of rewrite rules represented by set R. Except for rules that rewrite the start nonterminal to base structures, we can determine the type $T \in \{A, D, O\}$ of nonterminal T_n on the left side of the rule $r = T_n \rightarrow \gamma$. The type T defines if the rule is to create fragments of letters (A), digits (D), or other (O) characters. The n represents the length of the created fragments. By analyzing PCFGs created from bigger dictionaries, we can detect the number of rules r with the same left side T_n have equal probability $P(r)$.

Definition 5 (Group of preterminal rules) *Let group of preterminal rules $R_{T_n} \in R$ be a set of rewrite rules $T_n \rightarrow \gamma$ with equal left side T_n and $\gamma \in \Sigma^*$, i.e., the right side consists of terminal symbols only: $R_{T_n} = \{T_n \rightarrow \gamma \mid T_n \rightarrow \gamma \in R \wedge \gamma \in \Sigma^*\}$.*

Definition 6 (Probability group) *Let probability group $R^p_{T_n} \subseteq R_{T_n}$ be a group of preterminal rules where all have same probility p: $R^p_{T_n} = \{r \mid r \in R_{T_n} \wedge P(r) = p\}$.*

Table 4.4 shows an example of rewrite rules for nonterminal A_{14}. One can see, the rules create groups with an equal probability. For simplicity, I do not assume rules for letter capitalization in this example. The existence of probability groups in grammars is the core assumption for the *Next* algorithm described in the following section.

4.4.5 The Next Function

Weir et al. implemented the *Next* function that creates preterminal structures in decreasing probability order. The implementation uses a priority queue where the probability defines the priority. The queue serves to store the preterminal structures temporarily when they are processed. Its implementation ensures that the structures are ordered by their probability automatically. The *pop()* method thus takes a stored preterminal structure with the highest probability value.

An element of the priority queue stores not only the preterminal structure, but also its probability, the base structure of its origin, the size of the base structure in nonterminals A, D, O, and a pivot value. The *pivot* ensures that we do not generate any preterminal structure more than once and that each password guess belongs to a single derivation tree.

R			P	
T_n	$\rightarrow$	γ	$P(T_n \rightarrow \gamma)$	group
A_{14}	$\rightarrow$	siempreteamare	0.002379693	$R_{A_{14}}^{0.002379693}$
A_{14}	$\rightarrow$	backstreetboy	0.002379693	
A_{14}	$\rightarrow$	elamordemivida	0.002379693	
A_{14}	$\rightarrow$	threedaysgrace	0.001322052	$R_{A_{14}}^{0.001322052}$
A_{14}	$\rightarrow$	paralelepipedo	0.001322052	
A_{14}	$\rightarrow$	glasgowrangers	0.001322052	
A_{14}	$\rightarrow$	loveisintheair	0.001322052	
A_{14}	$\rightarrow$	showmethemoney	0.001322052	
A_{14}	$\rightarrow$	lordoftherings	0.001057641	$R_{A_{14}}^{0.001057641}$
A_{14}	$\rightarrow$	iloveyousomuch	0.001057641	
A_{14}	$\rightarrow$	jessemccartney	0.001057641	
A_{14}	$\rightarrow$	ilovechocolate	0.001057641	
...	...	...	...	...

Table 4.4: An example of probability groups of rewrite rules

The input of the algorithm is a probabilistic context-free grammar. In the input representation of the grammar, the rewrite rules for A, D, O nonterminals are sorted in probability order.

As a starting point for creating preterminal structures, the algorithm uses the bases structures. Then, it rewrites their nonterminals using the rewrite rules from the most probable to the least probable ones. If multiple rules with the same probability are applicable, the algorithm "substitutes" the entire probability group. Concretely, it denotes that in the password guessing phase, the given nonterminal will be gradually modified by all rules from the given probability group (see Section 4.4.4). For demonstration purposes, I will illustrate this by displaying terminals that can be substituted, e.g., instead of $4A_3\$\$$, we write $4\{cat, dog\}\$\$$. This notation defines that in the password guessing phase, A_3 will be replaced by "cat" and "dog".

The functionality of the Next function is illustrated by algorithm 8. The PT identifier denotes a preterminal structure. The algorithm gradually inserts the PT-containing elements to the priority queue, pops existing ones out, and creates new from them. Creating new preterminals structures from existing ones is achieved by applying different, less probable, rewrite rules to the original base structure. The algorithm uses three support functions:

- *calculate_probability(PT)* calculates the probability of passwords that can be created from the PT;
- *generate_passwords(PT)* represents the "second phase": generates all possible password guesses from the PT;
- *decrement(PT, i)* creates a new preterminal structure from PT by applying a next (less probable) rewrite rule at position i, or denotes the use of a next (less probable) probability group at position i.

The functionality of the Next algorithm is illustrated in an example. Assume the PCFG with rewrite rules from Table 4.5. In the initial phase, for each base structure, the algorithm

Algorithm 8: The algorithm of the Next function [211]

```
   Data: queue, PCFG
1  // For each base structure, get PT with the highest probability
2  foreach base in PCFG do
3      element.structure = the highest probability PT from base
4      element.pivot = 0
5      element.num_strings = the size of base
6      element.p = calculate_probability(element.structure)
7      push(queue, element)
8  element = pop(queue)
9  while element != NULL do
10     generate_passwords(element) // all passwords from PT
11     for i = element.pivot; i < element.num_strings; i++ do
12         // Use next rule at position i
13         new.structure = decrement(element.structure, i)
14         if new.structure != NULL then
15             new.p = calculate_probability(new.structure)
16             new.pivot = i
17             new.num_strings = element.num_strings
18             push(queue, new)
19     element = pop(queue)
```

creates the most probable PT and pushes it to the priority queue together with its size and probability. Initially, the pivot value is set to 0. After the initial phase, the number of elements in the priority queue equals the number of base structures in the grammar.

The example grammar contains two base structures, and thus after the initial phase, the priority queue contains two elements, as illustrated in Table 4.6. Since the rewrite rules $A_3 \rightarrow cat$ and $A_3 \rightarrow dog$ have the same probability, the entire probability group $\{cat, dog\}$ is used as a replacement. The rewriting to final passwords will be performed later by the *generate_passwords()* function.

At first, we mark individual positions of nonterminals in each base structure with indexes from left to right, starting from 0. For $A_3D_1O_1$, the nonterminal A_3 has index 0, D_1 has index 1, and O_1 has index 2.

Then, we progressively pop elements from the queue and create new ones from them by applying yet un-used (groups of) rewrite rules to the corresponding base structure. However, we only replace nonterminals whose indexes are higher than the current pivot value.

Assume an example of the priority queue from Table 4.6. Callint the *pop()* operation withdraws the element with the higest probability: $4\{cat, dog\}\$\$$. This element generates passwords "4cat$$" and "4dog$$". Then, two new elements with pivot values 0 and 2 are created and pushed into the queue. The contents of the queue after this operation is illustrated by Table 4.7. Note, two different rewrite rules were applied to the base structures for nonterminals D_1 and O_2. Since rules $D_1 \rightarrow 5$ and $D_1 \rightarrow 6$ have the same probability, the entire probability group was used again. The process continues by following the algorithm 8, as long as we can generate new elements and the correct password is not found.

R			P
A	$\rightarrow$	γ	$P(A \rightarrow \gamma)$
S	$\rightarrow$	$D_1A_3O_2D_1$	0.75
S	$\rightarrow$	$A_3D_1O_1$	0.25
A_3	$\rightarrow$	cat	0.5
A_3	$\rightarrow$	dog	0.5
D_1	$\rightarrow$	4	0.6
D_1	$\rightarrow$	5	0.2
D_1	$\rightarrow$	6	0.2
O_1	$\rightarrow$	!	0.65
O_1	$\rightarrow$	%	0.3
O_1	$\rightarrow$	#	0.05
O_2	$\rightarrow$	$$	0.7
O_2	$\rightarrow$	**	0.3

Table 4.5: An example of PCFG rewrite rules for illustration of the Next function

base structure	**preterminal structure**	**probability**	**pivot**
$D_1A_3O_2$	$4\{cat, dog\}\$\$$	0.1575	0
$A_3D_1O_1$	$\{cat, dog\}4!$	0.04875	0

Table 4.6: The contents of the priority queue contents at the beginning of computation

base structure	**preterminal structure**	**probability**	**pivot**
$D_1A_3O_2$	$4\{cat, dog\} * *$	0.1575	2
$D_1A_3O_2$	$\{5, 6\}\{cat, dog\}\$\$$	0.1575	0
$A_3D_1O_1$	$\{cat, dog\}4!$	0.04875	0

Table 4.7: The contents of the priority queue after processing the first element

Weir et al. provide proof of the correctness of the Next algorithm [213]. The proof gives the following conslusions:

- All possible preterminal structures that can be created from a PCFG are eventually created.
- No preterminal structure is created more than once.
- Preterminal structures at the output are in non-increasing probability order.

For each PCFG, the algorithm of the Next function produces a single unique derivation tree. And thus, for each string generated by the grammar, there is only one subset of the set of rewrite rules that produces it. This is ensured by the pivot value.

4.4.6 The Deadbeat Dad Algorithm

Despite the Next algorithm being feasible for creating preterminal structures, generating from more complex PCFGs has enormously high memory requirements. Such a problem is solved by the *Deadbeat dad* algorithm that uses the same priority queue but requires a significantly lower amount of memory [211].

The reason why the original Next function has high memory requirements is illustrated in an example. For simplicity, assume a preterminal structure made of three nonterminals. At each position, there are three possible replacements: 1, 2, and 3. Figure 4.2 displays all the possibilities generated by the Next function. Each node represents a single preterminal structure. For simplicity, the probability values are neglected. After the initial phase, node (1,1,1) is pushed to the priority queue as the first entry. After its popped out, nodes (2,1,1), (1,2,1), and (1,1,2) are generated from it and pushed back to the queue. Every time, the algorithm pops the node with the highest probability. In the Next function, the pivot value ensures that every child node has only one parent. However, the problem is that the size of the priority queue grows enormously.

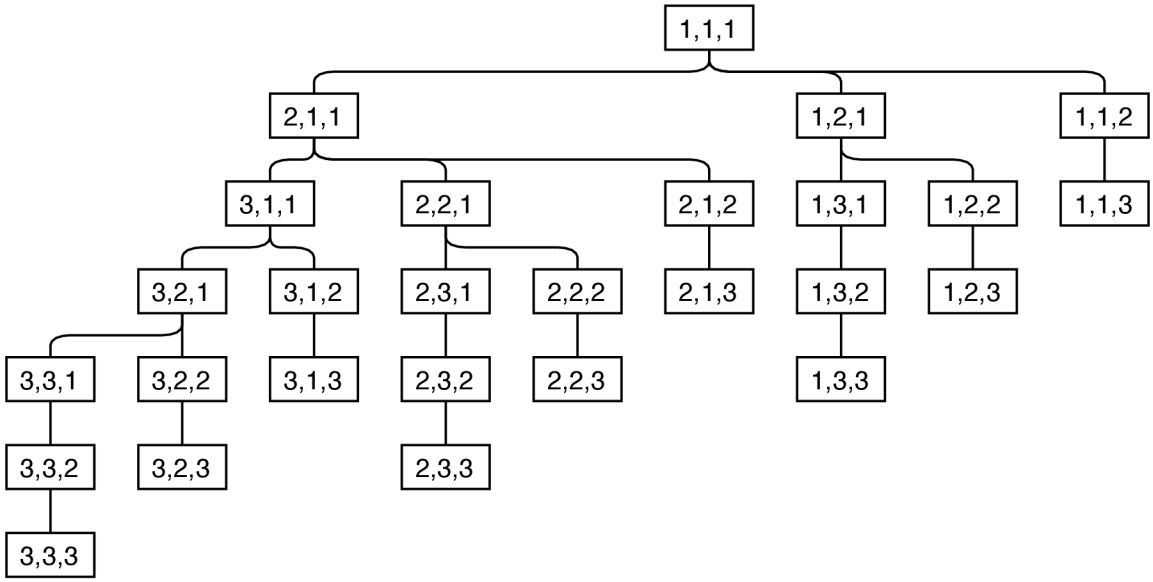


Figure 4.2: Illustration of the nodes generated by the Next function

As illustrated in Figure 4.3, when the Next function pops node 1 from the priority queue, it generates nodes 2 and 3 from it and pushes both to the priority queue. Next, node 2 is popped because it has a higher probability than node 3. Then, the function creates node 4 and pushes it into the priority queue, because node 4 is node 2's child, as illustrated in Figure 4.2. See that node 3 has a higher probability than node 4. That means that node 4 needlessly occupies the queue and is never popped out before node 3 is popped. And thus, it would make sense to have node 3, instead of node 2, responsible for creating node 4.

The idea behind the Deadbeat dad algorithm proposed by Matt Weir is to postpone pushing nodes to the priority queue as long as possible. The design of the algorithm is motivated by two facts:

- No child nodes are popped before their parents are popped.
- The parent nodes are popped in probability order.

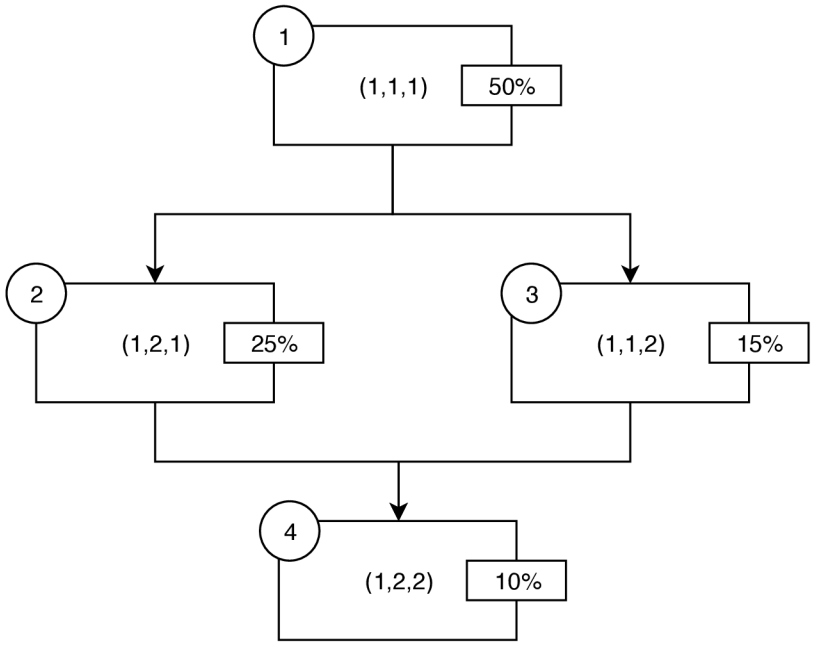


Figure 4.3: Multiple parents for node no. 4

The Deadbeat dad algorithm uses the same priority queue as the Next function but ensures that every child node is inserted by the parent node with the lowest probability. How it works is illustrated in Figure 4.4. Node (1,2,2) has three possible children: (2,2,2), (1,3,2), and (1,2,3). For each of them, the algorithm generates all possible parents and checks if any of them have a lower probability then the currently popped parent. In this case, the probability of the currently popped parent (1,2,2) is 10%. Both node (2,2,2) and node (1,2,3) have other parents whose probability is lower. Therefore, the other parents will take care of them in the future, and the only child that is inserted into the priority queue is (1,3,2).

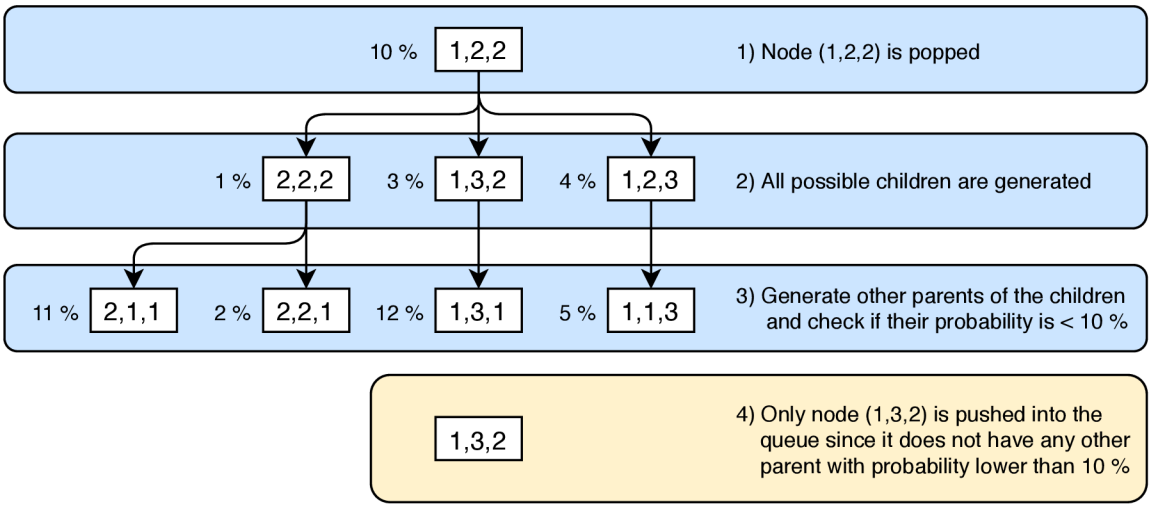


Figure 4.4: A demonstration of the Deadbeat Dad algorithm

Algorithm 9 shows the pseudo-code of the Deadbeat dad algorithm. The code uses an additional function called `lowest_probability_parent()` whose functionality is illustrated by Algorithm 10. The function returns true if, for the given node, the given parent is the one with the lowest probability. In the Python PCFG Cracker, it is implemented inside the `dd_is_my_parent()` function, to which I refer in Section 4.7.1. The `get_parent_probability()` function returns the probability of the i-th other parent.

In the example from Figure 8, the popped node is (1,2,2) with a probability of 10 %. For all three children, the Deadbeat dad algorithm calls the `lowest_probability_parent()` function. For nodes (2,2,2) and (1,1,3), it returns false since they have other parents with probabilities 2 % and 5 %. For node (1,3,2), it returns true, because the popped node (1,2,2) is its parent with the lowest probability. In contrast to the Next function, the elements inside the probability queue do not have the pivot values. The pivot i serves only as a temporary variable that is used locally in the algorithm.

Algorithm 9: The Deadbeat dad algorithm [211]

```
   Data: queue, PCFG
1  // For each base structure, get PT with the highest probability
2  foreach base in PCFG do
3      element.structure = the highest probability PT from base
4      element.num_strings = the size of base
5      element.p = calculate_probability(element.structure)
6      push(queue, element)
7  element = pop(queue)
8  while element != NULL do
9      generate_passwords(element) // all passwords from PT
10     for i = 0; i < element.num_strings; i++ do
11         // Use next rule at position i
12         child.structure = decrement(element.structure, i)
13         child.num_strings = element.num_strings
14         if child.structure != NULL
15         and lowest_probability_parent(child, element, i) then
16             child.p = calculate_probability(new.structure)
17             push(queue, child)
18     element = pop(queue)
```

Figure 4.5 displays the size of the priority queue with the increasing number of password guesses. The grammar used is trained on the MySpace dataset. In the experiment, the grammar used letter capitalization, and probability smoothing was applied. We can see that the Deadbeat dad algorithm reduces the necessary amount of space dramatically in contrast to the original Next function. The saved space is, however, redeemed by higher computational complexity. As I detected, the biggest obstacle represents the high number of iterations of the `lowest_probability_parent()` function. The number increases with the number of probability groups. In this chapter, I propose a solution that reduces the number of required computing operations by filtering out long base structures. The idea is described in Section 4.7.1.

Algorithm 10: The pseudo-code of the `lowest_probability_parent()` function used by the Deadbeat dad algorithm [211]

```
Input  : child, element, parent_pivot
Data:  PCFG
Output: boolean
for i = 0; i < child.num_strings; i++ do
    if i != parent_pivot then
        // not the parent node // find the probability of the other parent
         other_parent_probability = get_parent_probability(child, i)
        if other_parent_probability < element.probability then
            // some other parent will take care of the child
            return false
        else if other_parent_probability == element.probability then
            // the location of the pivot is used
            if i > parent_pivot then
                return false
return true
```

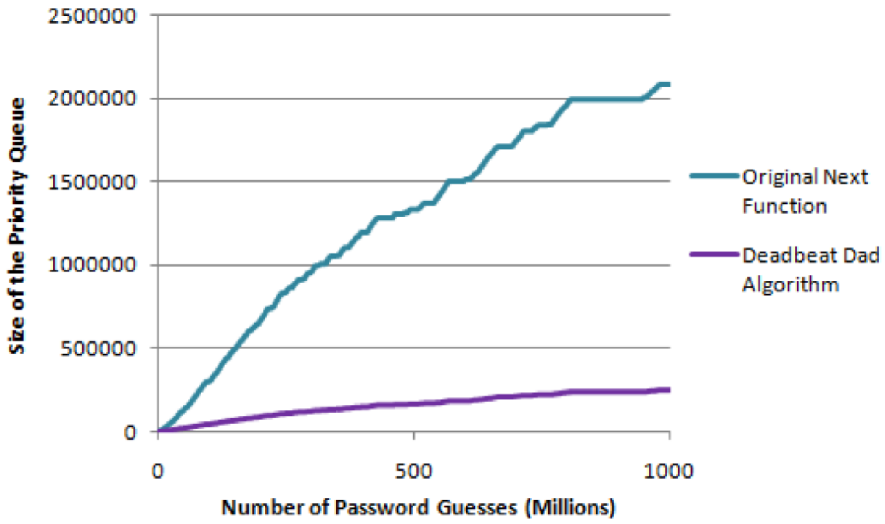


Figure 4.5: Size of the priority queue with the Next function and the Deadbeat dad algorithm (Source: Matt Weir's dissertation [211])

4.5 Key Observations

By analyzing the existing methods and the behavior of Weir's proof-of-concept PCFG Cracker on various leaked password datasets, I observed the following:

- The Python implementation of PCFG Manager uses a priority queue and three processes: one that fills the queue with pre-terminal structures [213], one that creates terminal structures (password guesses), and one for storage backup. No other parallelization is supported. Thus, the processor cores are not utilized well.

- Processing long base structures like $A_1D_1A_2D_2A_3D_3A_4D_4A_5D_5$ is computationally complex and wastes a lot of time even if their probabilities are insignificant.
- Rewrite rules for alpha characters (A), digits (D), and other symbols (O) have all similar probability, while rewrite rules for base structures differ more between each other.
- For capitalization of letter fragments, a grammar usually contains few (1 to 4) rules with higher probabilities while the rest have probability below 0.1 and only little impact on success rate.
- The PCFG Manager is a utility that generates and prints out the password guesses. While it is possible to perform a live cracking by redirecting the output directly to a password cracking tool like hashcat, the existing solution is by design a single-machine one. Without additional implementation, the only way of employing multiple nodes is to generate a password wordlist offline using Weir's tool, split it to smaller ones and use each to perform a dictionary attack on a particular computing node. Such a solution, however, requires a lot of user effort. Moreover, the efficiency of the attack would be low since it prevents generating and verifying passwords at the same time. Performing a live distributed cracking session instead would require:
  - an external software that transfers the generated password guesses to other computing nodes, or
  - a modification of the existing solution.

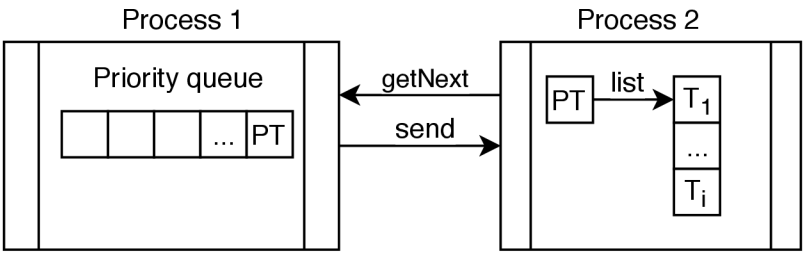
4.6 Parallel PCFG Cracking

As mentioned in Section 4.5, the downside of the PCFG Manager proposed by Weir et al. [213, 211] is a relatively low password guessing performance. As I detected, the reasons are both in the design and implementation. Concretely, lacking parallelization and missing option for compilation into the native code. Therefore, I propose an enhanced design that supports parallel and distributed (see Section 4.8) computing to improve the use of available resources. In the following paragraphs, I describe the changes step-by-step and directly propose an alternative proof-concept implemantation[10].

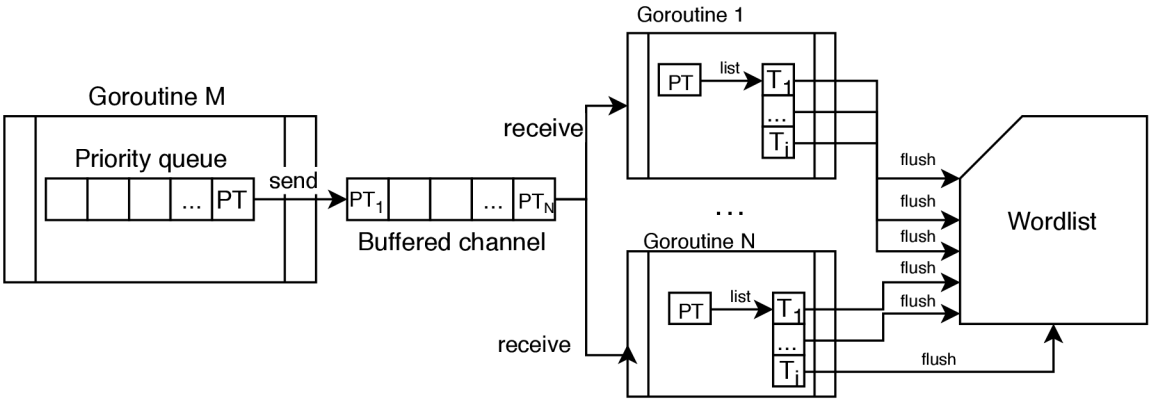
First of all, I wanted to eliminate the infulence of using an interpreted language. Therefore, the first step was a simple transcription of the original Python sources to Go[11] programming language that was chosen because of its speed, simplicity, and compilation to machine language. Early experiments showed that the Go-based alternative using the same algorithms was about four times faster than the original solution. However, there was still enough space for optimization.

Within all operations performed by the PCFG Manager, generating password guesses from preterminal structures [213, 211] was the most computationally complex part. Since there is no mutual dependence between the preterminals, I decided to modify the program and parallelize this part of the process. The new design uses a single *goroutine* (a lightweight thread) for filling the priority queue [213] with preterminal structures, and one to n goroutines for generating terminals in parallel. The n can be set by a user to reflect the processor's capabilities. Moreover, the new tool provides a parameter which allows the

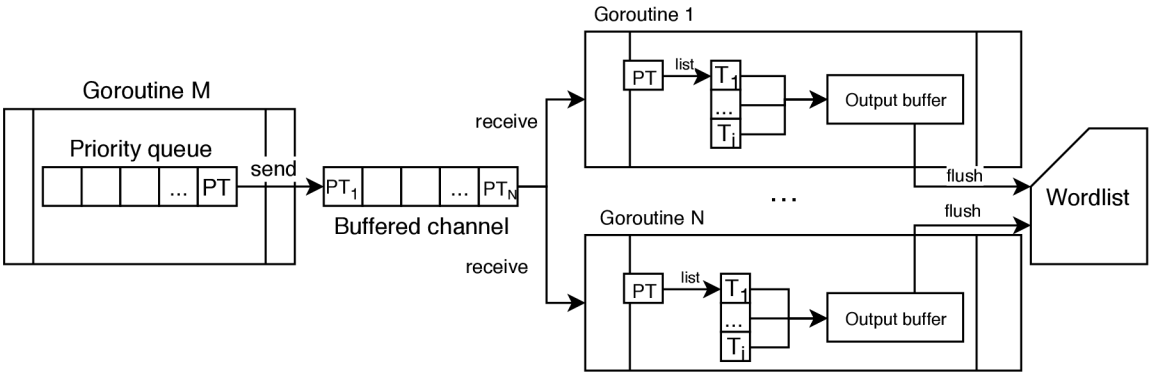
[10] https://github.com/nesfit/pcfg-manager
[11] https://golang.org/



(a) Python PCFG Manager



(b) Go PCFG Manager



(c) Go PCFG Manager with buffered output

Figure 4.6: The architecture of PCFG Manager in Python and Go
(PT - preterminal structure, T - terminal)

user to limit the number of generated password guesses. I illustrate both approaches by simplified schematics that display goroutines and data transfer operations. While Figure 4.6(a) shows the original design of Weir's PCFG Manager, the parallel version is depicted in Figure 4.6(b).

For synchronization and mutual communication, goroutines use a mechanism called *channels* that act as FIFO queues. A goroutine can send values to a channel or receive values from it. By default, channels are not buffered and both send and receive operations are blocking. The proposed solution uses a buffered channel of size n where the sender is blocked only if the channel contains n values in the queue. Each value represents a preterminal

structure. The main goroutine (M) implements the Deadbeat dad algorithm [211] filling the priority queue with preterminals. Every time a preterminal is created, it is sent to the buffered channel if there is enough space. Every time the channel is full, the main goroutine is suspended automatically by the send operation. There is no need to generate more preterminals at the time they cannot be processed. In contrast to the original version, the proposed design allows to process multiple preterminals and generate passwords in parallel if $n > 1$. In that case, the only apparent drawback is the possible slight change of the password order at the output. This behavior could be resolved by adding a supplementary synchronization mechanism at the output, however, at the cost of performance loss. For practical use, I do not consider this as a large obstacle since for millions of password, the changes are insignificant because the order of larger password blocks is preserved. Moreover, if the user does not set the guess limit explicitly, or if the limit is set in the PCFG Mower (see Section 4.7) instead of PCFG Manager, the output dictionary contains the same passwords, and the success rate would be intact.

Additional profiling, revealed that even though the parallelization accelerated generating terminal structures, the new bottleneck was at the output, where simple I/O text operations slowed down the entire process. Eventually, this obstacle was removed by adding extra output buffers to goroutines that generate terminal structures. The buffers store the terminal structures and are flushed to output after the entire preterminal is processed. The final design is illustrated in Figure 4.6(c) and the experimental results in Section 4.9.1.

4.7 Grammar Filtering

Grammars created from real datasets of passwords often have a high number of rewriting rules. Performing a PCFG-based attack in an acceptable time requires to limit the number of password guesses. If such a limit is specified, many of the rules are never used. And as I have experimentally detected, the presence of some even complicate the computation process and make password guessing slower.

Therefore, in this section, I propose methods of grammar filtering that remove selected rules from the grammar to: a) make guessing faster, b) obtain a compact grammar that can be processed entirely without defining a "hard limit" for password guesses.

4.7.1 Long Base Structures

For every PCFG, possible sentential forms create a tree structure where the starting symbol represents the root node, and terminal structures are leaves. Every edge stands for the application of a rewriting rule that transforms a parent node to a child node. In terms of probabilistic password cracking, terminal structures are password candidates, and base structures (e.g., $A_4D_2O_1$) are located on the second level of the tree.

In PCFG Manager, every base structure is processed by *Deadbeat dad* algorithm [211]. The goal of this algorithm is to create new children from the current node and ensure that these child nodes are inserted into the priority queue in the correct order. Deadbeat dad replaced the original *Next* function [213] and significantly reduced the size of the priority queue at the expense of computing operations [211].

I analyzed the algorithm and observed that the most expensive task is to find every possible parent of every node which is being inserted into the priority queue. In Weir's PCFG Manager, the task is resolved by a function called `dd_is_my_parent` that runs in iterations whose count is potentially increased by every non-terminal present in the processed

base structure. The deciding factor is the number of different probabilities assigned to the rewriting rules applicable to the non-terminal. This behavior bears on the use of *probability groups* described in Section 4.4.3. If all usable rules have the same probability value, the number of iterations is not increased. The more different probabilities are present, the more rapidly the iteration count grows, if the non-terminal is added to the base structure.

Table 4.8 shows the number of `dd_is_my_parent` iterations under different settings. For D_3 non-terminal, all rules have the same probability, and thus D_3 has no impact on the iteration count. For A_1, rewriting rules have 26 to 29 different probability values (A_1^p). As a capitalization rule for A_1, only „L“ is used. One can see, the number of iterations grows almost exponentially each time A_1 is added to the base structure.

base structure	$A_1^p = 26$	$A_1^p = 27$	$A_1^p = 28$	$A_1^p = 29$
A_1	103	107	111	115
A_1D_3	103	107	111	115
$A_1D_3A_1$	15,811	17,067	18,371	19,723
$A_1D_3A_1D_3$	15,811	17,067	18,371	19,723
$A_1D_3A_1D_3A_1$	1,506,286	1,688,528	1,884,906	2,095,948
$A_1D_3A_1D_3A_1D_3$	1,506,286	1,688,528	1,884,906	2,095,948
$A_1D_3A_1D_3A_1D_3A_1$	120,939,106	140,790,314	162,990,446	187,717,930

Table 4.8: The number of iterations of `dd_is_my_parent` function

In PCFGs trained on leaked password datasets, the variedness between rule probabilities is usually high, especially for shorter character fragments. For long base structures, the `dd_is_my_parent` function may iterate millions of times which significantly slows the password guessing process. Such structures usually have low probability values since they are in most cases created from randomly generated strings, not created by users. As I assume and experimentaly prove in Section 4.9.2, removing such structures from the grammar speeds up password generation several times and does not noticeably decrease success rate at cracking sessions. Experiments with

4.7.2 Calculating the Number of Password Guesses

The exact calculation of possible password guesses from a PCFG is a currently missing feature that is, however, essential for tools presented in this chapter. Let $size(N)$ be the number of terminal structures that can be created by applying rewrite rules on non-terminal N. For base structure $B = N_1N_2 \dots N_n$, the number of possible password candidates can be calculated as:

$$cnt_base(B) = \prod_{i=1}^{n} size(N_n). \tag{4.8}$$

For grammar G, the total number of possible password candidates is the sum of $cnt_base(B)$ for all base structures $B \in G$:

$$cnt_total(G) = \sum_{B \in G} cnt_base(B). \tag{4.9}$$

The file and directory structure of Weir's PCFG considers a single rewriting rule per line. All rewriting rules have non-zero probability, and thus, all are used. Therefore, $size(N)$ for

non-terminal $N = T_n$ (see Section 4.4) is, in most cases, the number of lines in `n.txt` file located in a directory for fragments of type T. For example, $size(D_3)$ equals the number of lines in `Digits/3.txt` file. Since letter capitalization rules have been introduced, it is necessary to take them into consideration. Thus, $size(A_n)$ is the number of lines in `Alpha/n.txt` file multiplied by the number of lines in `Capitalization/n.txt` file.

The calculation shown above is usable for classical PCFG-based approach only, i.e., with the `--coverage` parameter of PCFG Trainer set to 1. Otherwise, Weir's PCFG Manager would create additional character fragments using brute-force and Markov chains. Combining PCFGs with these attack methods is out of the scope of this research.

4.7.3 Rule Filtering

To increase speed even more, I experimented with various modifications of already-trained grammars. I noticed that removing rules which rewrite the starting symbol into long base structures brings a significant speedup without higher impact on a success rate. The motivation for such filtering was discussed in Section 4.7.1. I automated the process by creating a simple script that automatically filters out all base structures longer than a user-defined maximum.

At this point, I was able to generate much more passwords per time unit. However, without a manually-defined limit for password guesses, the total amount of time required for generating was still extensive. From a practical perspective, any limit to guess count means that there is always a part of the grammar that is never used and unnecessarily wastes memory during the guess generation. Such a consideration led me to speculate about reducing the size of the grammar instead of limiting guesses in PCFG Manager.

I came with an idea to remove the least significant rewriting rules from the grammar. I am aware of the fact that any removal of rules from already-created PCFG without adjusting probability values results in a mathematically incorrect grammar where the total probability of rules that rewrite some non-terminals may be lower than one – and thus, such a grammar is not purely probabilistic. The correctness of the grammar could be fixed by additional correction of the probability function $P(r)$ for all rules $r = A \rightarrow \gamma$ where any other rule with the same left side A was removed by the filtering. The correction can make the formula $\forall A \in N, \sum_{A \rightarrow \gamma \in R}(P(A \rightarrow \gamma)) = 1$ valid again.

From a practical standpoint, such a correction is not necessary since the implementation of the PCFG Manager can produce password guesses even if some rules are missing. The goal of the filtering is to make the output dictionary more compact and to ensure that generating passwords will end in an acceptable time. Besides, having a reduced grammar that can be processed entirely, ensures that even the parallel run of PCFG Manager generates the same passwords every time. Nevertheless, the strongest motivation for grammar filtering is a potentially massive saving of processor time. Putting a limit before the guessing even starts prevents the Deadbeat dad algorithm [211] from performing many useless derivation steps on trees that never form terminal passwords due to a low probability.

As denoted above, rules for alpha characters, digits, and special symbols usually have similar probabilities, thus removing them leads to a considerable loss of information which decreases the success rate. Rulesets for base structures and capitalization, on the other hand, contain many insignificant rewriting rules that can be removed safely.
To verify my assumptions, I created a draft of a simple *PCFG reduction algorithm* that is implemented in the PCFG Mower. Algorithm 11 shows its pseudocode. The goal of the algorithm is not to provide a universal solution, but to validate or disprove that system-

atic PCFG filtering brings a possible benefit to password cracking. Besides the original grammar, it takes the following input parameters: *limit* defining the maximum number of password guesses to be generated, and probability values b_s, c_s. While b_s allows to set how rapidly should the algorithm remove base structures, c_s sets the same for capitalization rules. The output of the algorithm represents a PCFG which generates the maximum of *limit* password guesses.

Algorithm 11: PCFG reduction algorithm

```
Input  : original grammar, limit, b_s, c_s
Output: reduced grammar
reduce = true, i = 0, grammar = original grammar
repeat
    i++
    count = password_count()
    if count ≤ limit then
        reduce = false
    if reduce then
        Remove as many base structures from the grammar as required to reduce
          their total probability by b_s.
        Remove all capitalization rules from the grammar that have probability
          lower than i × c_s.
until not reduce;
return grammar
```

It is essential to state that the proposed PCFG reduction algorithm represents a naive solution created for experimental purposes. The algorithm removes the least significant rewrite rules. Still, it is based on heuristics and does not guarantee that the filtering only prevents generating the least significant password guesses since the probability of every password guess depends on the probabilities of all applied rules.

To experimentally evaluate the technique's benefits, I propose a proof-of-concept tool called the *PCFG Mower*[12] which can:

- Calculate the total number of possible password guesses from a PCFG and inform the user about achievable keyspace. Moreover, if the user knows an average speed of password guessing, it is possible to estimate the total time required for generating all password candidates.
- Filter a PCFG by performing an automatic removal of rewriting rules based on a set of options entered by the user.

If the results show that such filtering brings advantages and can serve as an alternative to a password guess count limit, it is possible to use a more systematic way. I assume advanced filtering solutions can calculate with the preterminal or terminal probability for a more precise choice of what rules to remove.

[12] https://github.com/nesfit/pcfg_mower

4.8 Distributed PCFG Cracking

For distributed cracking, I assume a network consisting of a server and a set of clients, as illustrated in Figure 4.7. The server is responsible for handling client requests and assigning work. Clients represent the cracking stations equipped with one or more OpenCL-compatible devices like GPU, hardware coprocessors, etc. In the proposal, I talk about a client-server architecture since the clients are actively asking for work, whereas the server is offering a "work assignment service."

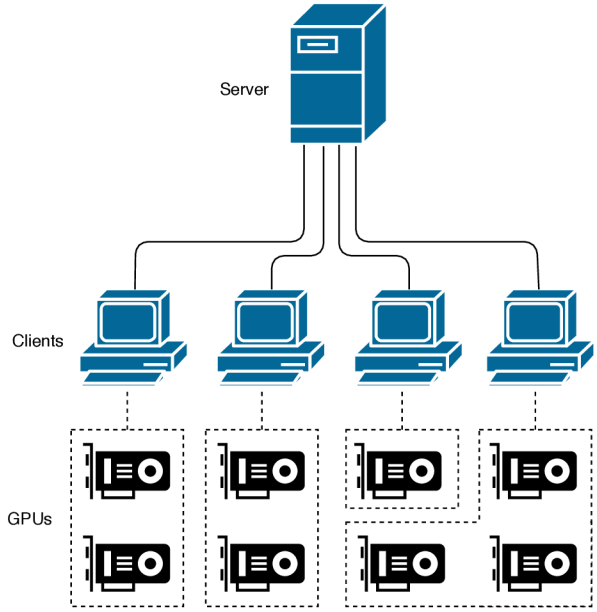


Figure 4.7: An example of a cracking network

In PCFG-based attacks, a probabilistic context-free grammar represents the source of all password guesses, also referred to as candidate passwords. Each guess represents a string generated by the grammar, also known as a *terminal structure* [213, 211]. In a distributed environment, we need to deliver the passwords to the cracking nodes somehow. A naive solution is to generate all password candidates on the server and distribute them to clients. However, such a method has high requirements on the network bandwidth, and as I detected in my previous research, also high memory requirements to the server [82]. Another drawback of the naive solution is limited scalability. Since all the passwords are generated on a single node, the server may easily become a bottleneck of the entire network.

And thus, I propose a new distributed solution that is inspired by the parallel one described in Section 4.6. In the following paragraphs, I describe the design and communication protocol of the *distributed PCFG Manager*. To verify the usability of the concept, the proof-of-concept parallel PCFG Manager[10] was extended with the support for distributed computing.

The general idea is to divide the password generation across the computing nodes. The server only generates the preterminal structures (PT), while the terminal structures (T) are produced by the cracking nodes. The work is assigned progressively in smaller pieces called chunks. Each *chunk* produced by the server contains one or more preterminal structures, from which the clients generate the password guesses. To every created chunk, the server

assigns a unique identifier called the *sequence number*. The *keyspace*, i.e., the number of possible passwords, of each chunk is calculated adaptively to fit the computational capabilities of a node that will be processing it. Besides that, the design allows direct cracking with hashcat tool. I chose hashcat as a cracking engine for the same reasons as I did for the Fitcrack distributed password cracking system [84], mainly because of its speed and range of supported hash formats. The proposed tool supports two different modes of operation:

- **Generating mode** - the PCFG Manager generates all possible password guesses and prints them to the standard output. A user can choose to save them into a password dictionary for later use or to pass them to another process on the client-side.
- **Cracking mode** - With each chunk, the PCFG Manager runs hashcat in stdin mode with the specified hashlist and hash algoritm. By using a pipe, it feeds it with all password guesses generatable from the chunk. Once hashcat processes all possible guesses, the PCFG Manager returns a result of the cracking process, specifying which hashes were cracked within the chunk and what passwords were found.

4.8.1 Communication Protocol

The proposed solution uses remote procedure calls with the gRPC[13] framework. For describing the structure of transferred data and automated serialization of payload, it uses the Protocol buffers[14] technology.

The server listens on a specified port and handles requests from client nodes. The behavior is similar to the function of Gouroutine M from the parallel version (see Section 4.6) - it generates PT and tailors workunits for client nodes. Each workunit, called chunk, contains one or more PTs. As shown in listing 4.1, the server provides clients an API consisting of four methods. Listing 4.2 shows an overview of input/output messages that are transferred with the calls of API methods.

```
1 service PCFG {
2     rpc Connect (Empty) returns (ConnectResponse) {}
3     rpc Disconnect(Empty) returns (Empty);
4     rpc GetNextItems(Empty) returns (Items) {}
5     rpc SendResult(CrackingResponse) returns (ResultResponse);
6 }
```

Listing 4.1: Server API

When a client node starts, it connects to the server using the `Connect()` method. The server responds with the `ConnectResponse` message containing a PCFG in a compact serialized form. If the desire is to perform an attack on a concrete list of hashes using hashcat, the `ConnectResponse` message also contains a *hashlist* (the list of hashes intended to crack) and a number defining the *hash mode*[15], i.e., cryptographic algorithms used.

```
1 message ConnectResponse {
2     Grammar grammar = 1;
3     repeated string hashList = 2;
```

[13] https://grpc.io/
[14] https://developers.google.com/protocol-buffers
[15] https://hashcat.net/wiki/doku.php?id=example_hashes

```
4     string hashcatMode = 3;
5 }
6
7 message Items {
8   repeated TreeItem preTerminals = 1;
9 }
10
11 message ResultResponse {
12   bool end = 1;
13 }
14
15 message CrackingResponse {
16   map<string, string> hashes = 1;
17 }
```

Listing 4.2: Messages transferred between the server and clients

Once connected, the client asks for a new chunk of preterminal structures using the `GetNextItems()` method. In response, the server assigns the client a chunk of 1 to N preterminal structures, represented by the `Items` message. After the client generates and processes all possible passwords from the chunk, using the `SendResult()` call, it submits the result in the `CrackingResponse` message. In cracking mode, the message contains a map (an associative array) of cracked hashes together with corresponding plaintext passwords. If no hash is cracked within the chunk or if the PCFG Manager runs in generating mode, the map is empty. With the `ResultResponse` message, the server then informs the client, if the cracking is over or if the client should ask for a new chunk by calling the `GetNextItems()` method.

The last message is `Disconnect()` that clients use to indicate the end of their participation, so that the server can react adequately. For instance, if a client had a chunk assigned, but disconnected without calling the `SendResult()` method, the server may reassign the chunk to a different client. The flow of messages between the server and a client is illustrated in Figure 4.8.

4.8.2 Server

The server represents the controlling point of the computation network. As displayed in Figure 4.9, i maintains the following essential data structures:

- **Priority queue** - the queue is used internally by the *Deadbeat dad* algorithm [211] for generating pre-terminal structures. In the figure, the priority queue is part of the "Generator" block.

- **Buffered channel** - the channel represents a memory buffer for storing already generated PTs from which the server creates chunks of work.

- **Client information** - for each connected client, the server maintains its IP address, current performance, the total number of password guesses performed by the client, and information about the last chunk that the client completed: its keyspace and timestamps describing when the processing started and ended. If the client has a chunk assigned, the server also stores its sequence number, PTs, and the keyspace of the chunk.

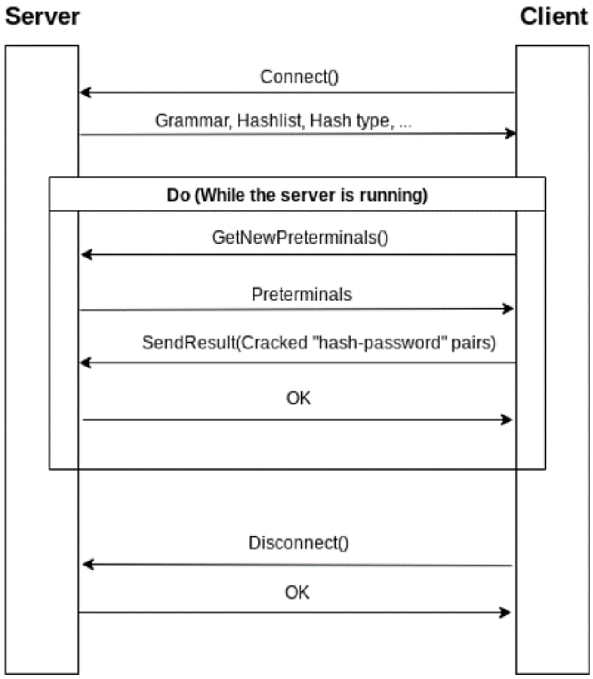


Figure 4.8: The proposed communication protocol

- **List of incomplete chunks** - the structure is essential for a failure-recovery mechanism I added to the server. If any client with a chunk assigned disconnects before reporting its result, the chunk is added to the list to be reassigned to a different client.
- **List of non-cracked hashes** (cracking mode only) - the list contains all input hashes that have not been cracked yet.
- **List of cracked hashes** (cracking mode only) - The list contains all hashes that have already been cracked, together with corresponding passwords.

Once started, the server loads an input grammar in the format used by Weir's PCFG Trainer[16]. Next, it checks the desired mode of operation and other configuration options – the complete description is available via the tool's help. In cracking mode, the server loads all input hashes to the list of non-cracked hashes. In generating mode, all hashlists remain empty. The server then allocates memory for the buffered channel, where the channel size can be specified by the user.

As soon as all necessary data structures get initialized, the server starts to generate PTs using the *Deadbeat dad* algorithm, and with each PT, it calculates and stores its keyspace. Generated PTs are sent to the buffered channel. The process continues as long as there is free space in the channel, and the grammar allows new PTs to be created. If the buffer infills, generating new PTs is suspended until the positions in the channel get free again.

When a client connects, the server adds a new record to the client information structure. In the `ConnectResponse` message, the client receives the grammar that should be used for generating passwords guesses. In cracking mode, the server also sends the hashlist and hash mode identifying the algorithm that should be used, as illustrated in Figure 4.8.

[16] https://github.com/lakiw/legacy-pcfg/blob/master/python_pcfg_cracker_version3/pcfg_trainer.py

Upon receiving the `GetNextItems()` call, the server prepares a new assignment for the client. The flow if information is displayed in Figure 4.9. The the server pops one or more PTs from the buffered channel and sends them to the client as a new chunk. Besides, the server updates the client information structure to denote what chunk is currently assigned to the client. The number of PTs taken depends on their keyspace. Like in Fitcrack [81, 84], the system schedules work adaptively to the performance of each client. In the distributed PCFG Manager, the performance of a client (p_c) in passwords per second is calculated from the keyspace (k_{last}) and computing time (Δt_{last}) of the last assigned chunk. The keyspace of a new chunk (k_{new}) assigned to the client depends on the client's performance and the `chunk_duration` parameter that the user can specify:

$$p_c = \frac{k_{last}}{\Delta t_{last}}, \tag{4.10}$$

$$k_{new} = p_c * chunk_duration. \tag{4.11}$$

The server removes as many PTs from the channel as needed to make the total keyspace of the new chunk at least equal to k_{new}. If the client has not solved any chunk yet, we have no clue to find p_c. Therefore, for the very first chunk, the k_{new} is set to a pre-defined constant.

An exception occurs if a client with a chunk assigned disconnects before reporting its result. In such a case, the server saves the assignment to the list of incomplete chunks. If the list is not empty, chunks in it have an absolute priority over the newly created ones. And thus, upon the following `GetNextItems()` call from any client, the server uses the previously stored chunk.

Once a client submits a result via the `SendResult()` call, the server updates the information about the last completed chunk inside the client information structure. For each cracked hash, the server removes it from the list of non-cracked hashes and adds it to the list of cracked hashes together with the resulting password. If all hashes are cracked, the server prints each of them with the correct password and ends. In generating mode, the server continues as long as new password guesses are possible. The same happens in the cracking mode in case there is a non-cracked hash.

Finally, if a client calls the `Disconnect()` method, the server removes its record from the client information structure. As described above, if the client had a chunk assigned, the server will save it for later use.

4.8.3 Client

After calling the `Connect()` method, a client receives the `ConnectResponse` message containing a grammar. In cracking mode, the message also include a hashlist and a hash mode. Then it calls the `GetNextItems()` method to obtain a chunk assigned by the server. Like in the parallel version (see Section 4.6), the client then subsequently takes one PT after another and uses the generating goroutines to create passwords from them. In the generating mode, all password guesses are printed to the standard output.

For the cracking mode, it is necessary to have a compiled executable of hashcat on the client node. The user can define the path using the program parameters. The PCFG Manager then starts hashcat in the *dictionary attack* mode with the hashlist and hash mode parameters based on the information obtained from the `ConnectResponse` message. Since no dictionary is specified, hashcat automatically reads passwords from the standard input. And thus, the client creates a *pipe* with the end connected to the hashcat's input. All password guesses are sent to the pipe. From each password, hashcat calculates a cryptographic

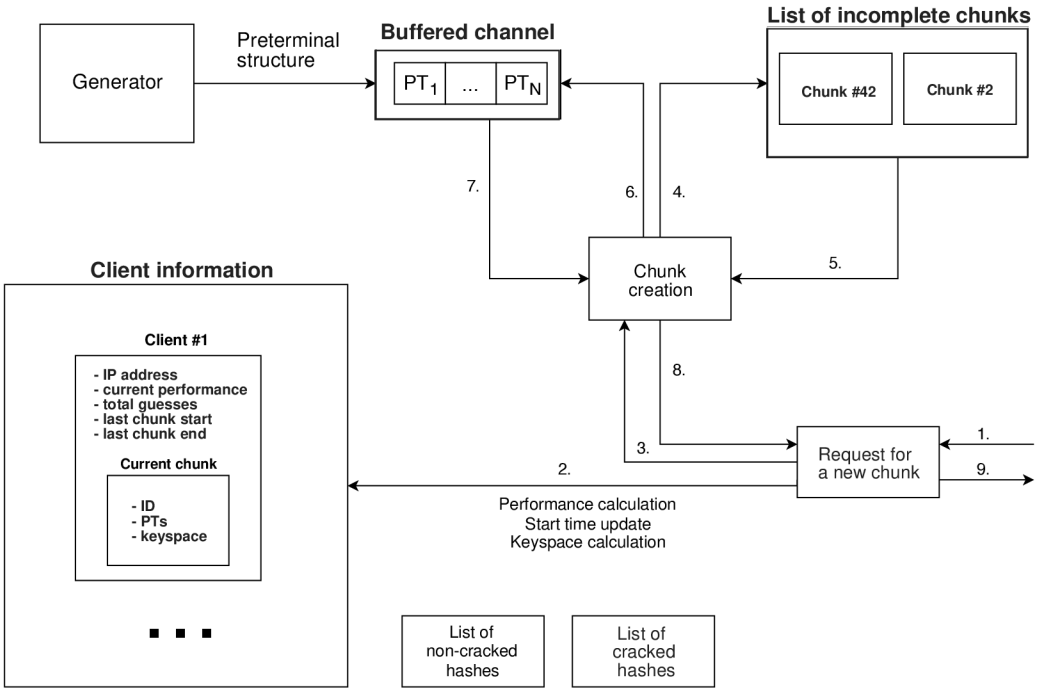


Figure 4.9: The architecture of the server and information flow when asked for a new chunk

hash and compares it to the hashes in the hashlist. After generating all password guesses within the chunk, the client closes the pipe and waits for hashcat to end and reads its return value. On success, the client loads the cracked hashes.

In the end, the client eventually informs the server about the chunk completion using the `SendResult()` call. If any hashes are cracked successfully, the client adds them to the `CrackingResponse` message that is sent with the call. The architecture of the PCFG Manager client is displayed in Figure 4.10.

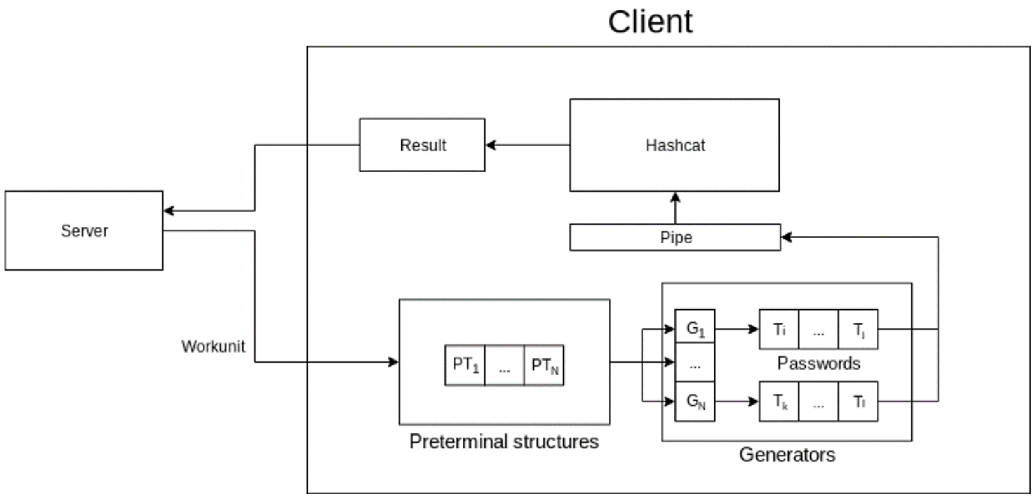


Figure 4.10: The architecture of the client side

4.9 Experimental Results

In this section, I demonstrate the practical benefits of the proposed enhancements to PCFG-based Password cracking. Section 4.9.1 compares the parallel version of the PCFG Manager to the original solution proposed by Weir et al. [213, 211]. Section 4.9.2 shows the advantages of grammar filtering. Finally, Section 4.9.3 experimentally proves the benefits of the proposed solution for distributed PCFG password cracking

4.9.1 Parallel PCFG Cracking

In this section, I measure the performance of the proposed parallel PCFG Manager. To evaluate its benefits, I compare the results with the original solution.

Experimental Setup

For experimental purposes, I work with both original and modified datasets from real password leaks. The wordlist used for experiments are obtained from SkullSecurity[6] pages and SecLists[7] repository. All employed datasets are enlisted in table 4.9. For shorter notation, I assign each a unique identifier (ID). The last row (def) represents the default PCFG from Weir's PCFG Cracker[4], which is said to be trained on a random sample of million passwords from RockYou dataset.

The table shows the number of passwords in the dataset (pw count), its size, and the average password length (avg). The other columns illustrate how a PCFG trained on the dataset looks like. Moreover, I show the number of rewriting rules for alpha characters (A), digits (D), other characters (O) as well as the number of rewriting rules for base structures (base) and capitalization (cap).

The first experiment measures the acceleration achieved using the new parallel PCFG Manager in contrast with the original one from Weir et al. [213]. Table 4.10 shows experimental results of generating password guesses using PCFG trained on *Darkweb2017-10000* dataset (dw), *RockYou-75* dataset (ru75), and the default PCFG (def) used in Weir's cracker. All three experiments were performed using a computer with Intel(R) Core(TM) i7-4700HQ CPU with 8 GB RAM. I also decided to study the influence of disk I/O speed, so that I measured everything using HDD and then using SSD. In all cases, I measured how many password guesses it it possible to generate within 3 minutes.

dataset					PCFG				
ID	name	pw count	size	avg	A	D	O	base	cap
dw	Darkweb2017-10000.txt	10,000	82.6 kB	7	5,244	947	30	323	83
r65	rockyou-65.txt	30,290	344.5 kB	7	17,845	4,213	35	256	39
r75	rockyou-75.txt	59,187	478.9 kB	7	30,670	10,601	51	351	51
ms	myspace.txt	37,126	354.2 kB	8	22,587	4,273	133	1,574	179
tl	tuscl.txt	38,820	324.7 kB	7	26,806	6,518	71	1,290	242
pr	probab-v2-top12000.txt	12,645	100.2 kB	6	11,117	534	1	125	23
def	Random million passwords from RockYou				330,343	145,510	906	84,307	950

Table 4.9: Password datasets used for experiments

The Results of Parallel Cracking

The proposed solution was from 8 to 40 times faster than the original one. Using *Darkweb* dataset (see Figure 4.11) resulted in lowest acceleration since it contains long and complex base structures. With the default PCFG (see Figure 4.12) and *Rockyou-75* dataset (see Figure 4.13), It was possible to generate much more password guesses, and the difference between HDD and SSD is more noticeable.

training	manager	HDD	SSD
dw	Python	3,022,923	2,948,532
	Go	24,592,908	24,609,579
	acceleration	8.14 x	8.35 x
def	Python	29,613,726	32,402,490
	Go	405,819,926	485,244,534
	acceleration	13.70 x	14.98 x
r75	Python	18,418,684	20,843,491
	Go	490,635,443	842,695,475
	acceleration	26.64 x	40.43 x

Table 4.10: No. of guesses and acceleration of the parallel PCFG Manager

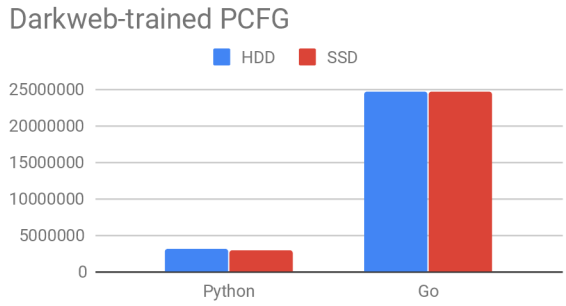


Figure 4.11: No. of guesses within 3 minutes using Darkweb-trained PCFG

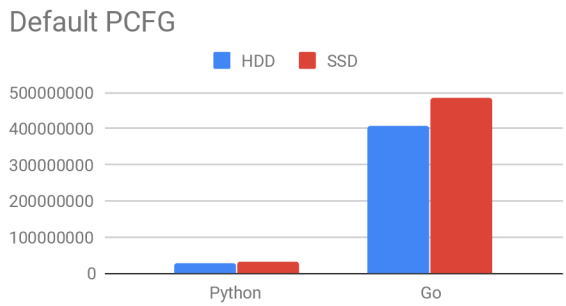


Figure 4.12: No. of guesses within 3 minutes using Default PCFG

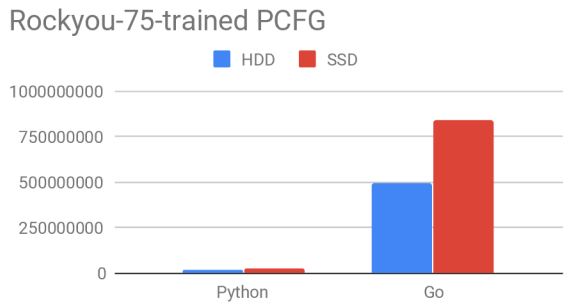


Figure 4.13: No. of guesses within 3 minutes using Rockyou-75-trained PCFG

Evaluation

The parallelization unambiguously showed the potential to make cracking faster. The results prove that the concept presented in Section 4.6 is usable and works well. Dividing the generation of password guesses from preterminal structures into multiple goroutines was a step forward. The experiment showed the chosen method brings a massive acceleration on a multi-core CPU since all cores can be utilized. What also helped to achieve higher performance was compilation into machine language instead of using an interpreter. Finally, the obtainable speedup highly depends on concrete grammar. The more simple base structures the grammar contains, the higher acceleration can be achieved with parallel computing.

4.9.2 Grammar Filtering

The second set of experiments aim to examine the effects of grammar filtering. Table 4.11 shows the results of training, modification, generating password guesses, and checking success rate using multiple datasets.

Experimental Setup

The experiments were performed using Intel(R) Core(TM) i7-7700K CPU with 32 GB RAM and an SSD. Since generating password guesses using non-modified PCFGs would take hours and days, a time limit of 10 minutes was set to all measurements - every time the PCFG manager exceeded the 10-minute interval, it was stopped.

The first column (tr) shows which dataset was used for training to create the PCFG. For all training datasets, the first line represents generating password guesses using the original grammar - i.e., without any modification. The *longbase* modification stands for the grammar where base structures longer than 10 characters (5 non-terminals) were removed. Other measurements use a grammar with already-removed long base structures and then filtered by the *PCFG Mower*. The *mow-n* modification means that *longbase* is performed first and then the *limit* of the PCFG reduction algorithm is n passwords. The experiment evaluate the following *limit* values: 1,000,000,000 (1000M), 500,000,000 (500M), and 20,000,000 (20M) passwords. In all cases, the b_s and c_s constants were set to 0.001 to achieve fine-grained filtering. Since the algorithm removes selected rules, the table illustrates the changes done to the grammars in each step. For every modification, it displays the preserved number of rewriting rules for base structures (base) and capitalization (cap).

Next columns inform about password guessing. We display the amount of time required to generate the output dictionary (time), (or $10m^*$ if it reached the time 10-minute limit), the size of the output dictionary (out size) and the number of its passwords in millions (mop). The rest displays the success rate of password guessing on testing datasets - i.e., the percentage telling how many generated password guesses were included in different testing datasets. The last column displays the *average success rate impact* (ASRI) which is calculated as:

$$ASRI = \frac{\sum_{i=1}^{n}(SR_i^{mod} - SR_i^{orig})}{n}$$

where SR_i^{orig} is the success rate on testing dataset number i before the modification of the PCFG, and SR_i^{mod} is the success rate on testing dataset number i after the modification of the PCFG, and n is the total number of testing datasets. In the experiments, $n = 4$. I use ASRI to analyze the influence of the modifications. Positive ASRI means that the success rate was improved while negative stands for decrease.

Removing Long Base Structures

As we can see from results, removing long base structures resulted in a massive increase of password guessing speed which enabled to generate much more passwords within 10 minutes. The highest acceleration was achieved on *dw* and *r65* since they contain very complex passwords that create enormously long base structures. After the modification, it was possible to generate over 14 times more password guesses. In contrast, training on *ms* and *tl* creates more simple grammars, and thus the speedup was not as rapid. Removing long base structures showed almost no impact on the success rate which confirms my assumption that their importance is negligible. From 16 testings, only 8 led to decrease by a maximum of 0.06 %. To my surprise, the ASRI was mostly positive since in 6 cases, removing long base structures improved the success rate by up to 0.7 % thanks to more passwords generated within the same time.

Rule Filtering

Next measurements analyzed grammars filtered by PCFG Mower to verify if the removal of low-probability rewriting rules brings any benefit. In all cases, the *mow* modification allowed the PCFG Manager to process the entire grammar in less than 4 minutes, showing that it can provide a suitable alternative to a "hard" limit for password guessing. More compact PCFGs produced smaller dictionaries. With more compact PCFGs, the generated dictionaries were smaller as well. Again, we achieved the best results with *dw* and *r65* datasets, where we were able to reduce the size from 12 GB (longbase) to 112 MB dictionary, and from 25 GB to 130 MB with a loss of success rate below 4 % in all cases. For *ms* and *tl*, filtering the grammar spared time and space as well, however, the *mow-20M* limit was too strict to provide satisfactory results. For *dw*, the *mow-1000M* and *mow-500M* modifications produced the same results since the grammar remained the same. The *dw*-trained grammar contains a high number of base structures with similar probabilities. Thus, a lot of them was removed by *mow-1000M* modification, and no further filtering was necessary.

Evaluation

Long base structures originate at the time of grammar creation. Their presence is caused by the existence of complex passwords in the training dictionary. In the leaked datasets

	grammar			password guesses			success rate				
tr	modification	base	cap	time	out size	mop	pr	ms	dw	r65	ASRI
dw	original	323	83	10m*	731 MB	78	45.03 %	26.83 %	98.27 %	41.39 %	
	longbase	288	83	10m*	12 GB	1,110	45.01 %	26.91 %	98.35 %	41.40 %	+0.04 %
	mow-1000M	106	40	25s	3.3 GB	373	44.54 %	24.47 %	96.42 %	38.36 %	-1.93 %
	mow-500M	106	40	25s	3.3 GB	373	44.54 %	24.47 %	96.42 %	38.36 %	-1.93 %
	mow-20M	86	32	2s	77 MB	9	44.18 %	24.12 %	95.65 %	38.00 %	-2.39 %
r65	original	256	39	10m*	1.5 GB	151	72.34 %	37.63 %	88.25 %	99.84 %	
	longbase	223	39	10m*	25 GB	2,210	72.30 %	37.63 %	88.14 %	99.81 %	-0.05 %
	mow-1000M	161	36	3m 31s	11 GB	980	72.17 %	37.17 %	87.73 %	99.61 %	-0.35 %
	mow-500M	123	31	1m 31s	4.5 GB	409	72.01 %	36.62 %	87.23 %	99.35 %	-0.71 %
	mow-20M	79	20	3.5s	130 MB	13.8	70.98 %	34.26 %	85.80 %	97.16 %	-2.47 %
ms	original	1574	179	10m*	5.7 GB	616	47.47 %	93.68 %	69.14 %	46.42 %	
	longbase	1430	179	10m*	9.5 GB	1,030	47.45 %	94.38 %	69.07 %	46.42 %	+0.15 %
	mow-1000M	110	25	3m	9.2 GB	941	46.37 %	82.40 %	66.74 %	43.04 %	-4.54 %
	mow-500M	78	20	1m	3.1 GB	334	45.13 %	79.67 %	64.71 %	42.62 %	-6.15 %
	mow-20M	21	20	2s	126 MB	15	33.25 %	61.17 %	54.28 %	35.58 %	-18.11 %
tl	original	1290	242	10m*	4.5 GB	520	55.27 %	36.87 %	69.85 %	43.86 %	
	longbase	1158	242	10m*	7.6 GB	870	55.23 %	37.15 %	69.79 %	43.87 %	+0.05 %
	mow-1000M	91	20	2m 43s	7.5 GB	884	54.06 %	30.94 %	66.08 %	40.37 %	-3.60 %
	mow-500M	48	19	1m 8s	1.8 GB	200	53.77 %	29.05 %	64.19 %	39.39 %	-4.86 %
	mow-20M	24	18	2s	133 MB	17	52.08 %	22.27 %	55.61 %	35.64 %	-10.07 %

Table 4.11: Success rates of original and modified PCFGs (* - reached the time limit)

used, such passwords occur only barely. Moreover, it is unlikely to have multiple complex passwords that belong to the same base structure. Usually, every long base structure only refers to a single password or a few passwords. Therefore, the rules for these structures have low probability values and are not actually used if the user sets a limit on password guessing. Their presence in a grammar, thus only complicates the process since the number of necessary computing operations within the Deadbeat dad algorithm [213, 211] is much higher. The experiments showed that removing long base structures simplifies generating preterminal structures dramatically, and thus bring a rapid speedup of password guessing. Without long base structures, it is possible to generate many more passwords for the same time interval.

With larger grammars, generating every possible password guess is not possible in an acceptable time. And thus, the guessing needs to be limited somehow. A limit on guess count or generating time resolves the issue but wastes space and processor time for loading rewrite rules that are never actually used. Rule filtering, on the other hand, removes these unnecessary rules and produces a grammar that can be processed entirely in minutes or even seconds without the need to set any limitation. Despite the PCFG filtering algorithm being heuristical and very simple, the experiments show that the filtering can serve as an alternative to classic guess or time limit. If used in practice, however, I suggest it would be appropriate to use a more systematic way of filtering that takes the probability of preterminal structures into account as well. It would also be advisable to perform additional correction of the probability function to renew the correctness of the grammar, as discussed in Section 4.7.3.

4.9.3 Distributed PCFG Cracking

I conduct a number of experiments in order to prove several points. First, I want to show the proposed solution results in a higher cracking performance and lower network usage. I also demonstrate that while the naive terminal distribution quickly reaches the speed limit by

Dictionary Statistics			PCFG Statistics			
name	**pw-cnt**	**avg-len**	**pw-cnt**	**base-cnt**	**avg-base-len**	**max-base-len**
myspace	37,145	8.59	6E+1874	1,788	4.50	600
cain	306,707	9.27	3.17E+15	167	2.59	8
john	3,108	6.06	1.32E+09	72	2.14	8
phpbb	184,390	7.54	2.84E+37	3,131	4.11	16
singles	12,235	7.74	6.67E+11	227	3.07	8
dw17[18]	10,000	7.26	2.92E+15	106	2.40	12

Table 4.12: Grammars used in the distributed cracking experiments

filling the network bandwidth, the new solution scales well across multiple nodes. I discuss the differences among different grammars and the impact of scrambling the chunks during the computation. In the experiments, I use up to 16 computing nodes for the cracking tasks and one server node distributing the chunks. All nodes have the following configuration:

- CentOS 7.7 operating system,
- NVIDIA GeForce GTX1050 Ti GPU,
- Intel(R) Core(TM) i5-3570K CPU,
- 8GB RAM.

The nodes are in a local area network connected with links of 10, 100, and 1000 Mbps bandwidth. During the experiments, I incrementally change the network speed to observe the changes. Furthermore, To analyze the influence of task size on overall results, the number of generated passwords is limited to 1, 10, and 100 million.

With this setup, I perform a number of cracking tasks on different hash types and grammars. As the hash cracking speed has a significant impact on results, I chose bcrypt with five iterations, a computationally difficult hash type, and SHA3-512, an easier, yet modern hash algorithm. Table 4.12 displays all chosen grammars with description. The columns cover statistics of the source dictionary: password count (pw-cnt) and average password length (avg-len), as well as statistics of the generated grammar: the number of possible passwords guesses (pw-cnt), the number of base structures (base-cnt), their average length (avg-base-len) and the maximum length of base structures (max-base-len), in nonterminals. One can also notice the enormous number of generated passwords, especially with the myspace grammar. Such a high number is caused only by few base structures with many nonterminals. I discussed the complexity added by long base structure in Section 4.7.1. If not stated otherwise, the grammars are generated from password lists found on SkullSecurity wiki page[17]. For each combination of described parameters, I run two experiments – first, with the naive terminal distribution (terminal), and second using our solution with the preterminal distribution (preterminals).

Computation Speedup and Scaling

The primary goal is to show that the proposed solution provides faster cracking with PCFG-based password guessing. In Figure 4.14, one can see the average cracking speed of SHA3-512 hash with *myspace* grammar, with different task sizes and network bandwidths.

[17] https://wiki.skullsecurity.org/Passwords

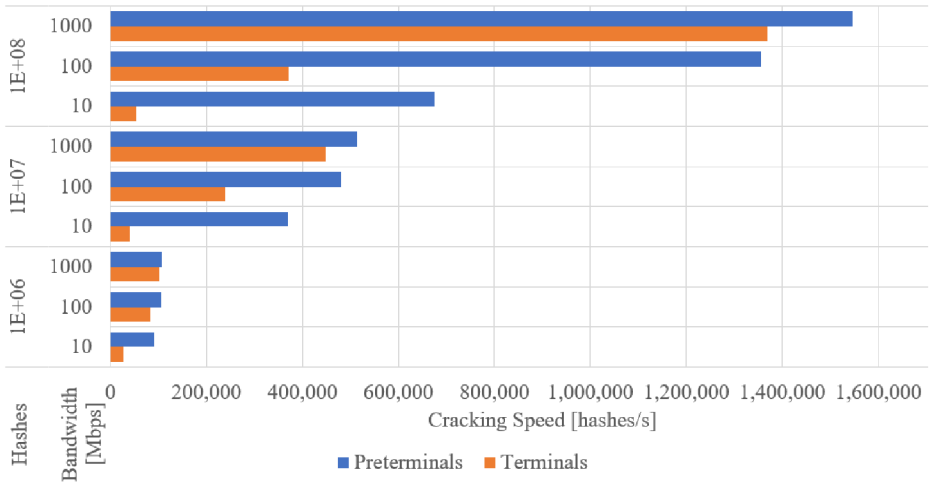


Figure 4.14: Average cracking speed with different bandwidths and password count (SHA3-512 / *myspace* grammar / 4 nodes)

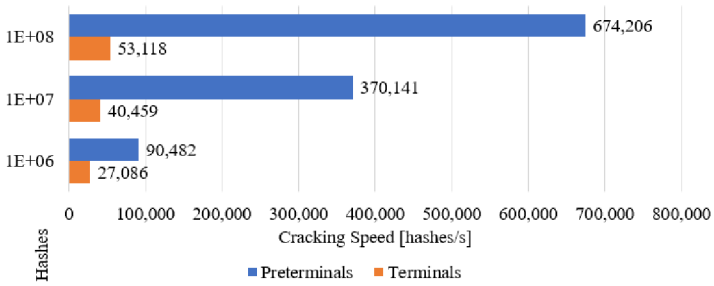


Figure 4.15: Detail of 10Mbps network bandwidth experiment (SHA3-512 / *myspace* grammar / 4 nodes)

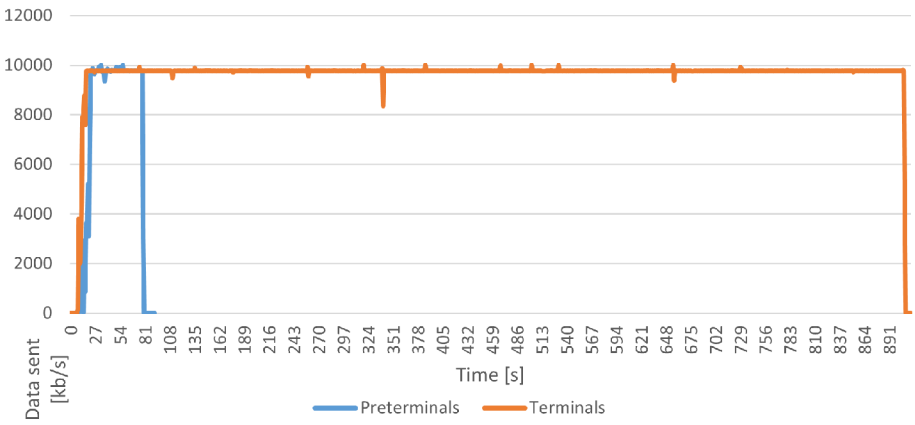


Figure 4.16: Comparison of network activity (SHA3-512 / *myspace* grammar / 100 million hashes / 10 Mbps network bandwidth / 16 nodes)

Apart from the proposed solution being generally faster, there is a significant difference in speeds with the lower network bandwidths. This is well seen in the detailed graph in Figure 4.15 which shows the cracking speed of SHA3-512 in a 10Mbps network with different task sizes. The impact of the network bandwidth limit is expected as the naive terminal distribution requires a significant amount of data in the form of a dictionary to be transmitted. In the naive solution, network links become the main bottleneck that prevents achieving higher cracking performance. In the newly proposed solution, the preterminal distribution reduces data transfers dramatically, which removes the obstacle and allow for achieving higher cracking speeds.

Figure 4.16 illustrates the network activity using both solutions. The graph compares the data transferred from the server to clients in the SHA3-512 cracking task on a 10Mbps network. Both runs use almost all of the bandwidth for the entire experiment. Nevertheless, we may see that cracking with the naive terminal distribution took much longer and required the transfer of more than a 14 times larger amount of data.

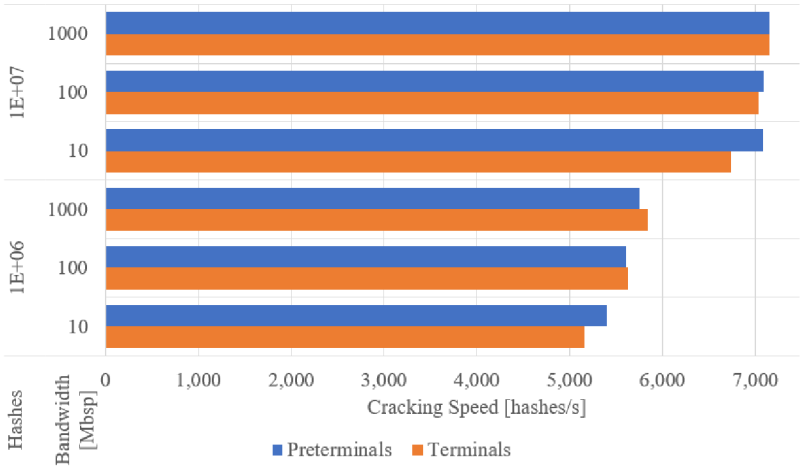


Figure 4.17: Average cracking speed with different bandwidths and password count (bcrypt / *myspace* grammar)

The difference between the two solutions disappears if we crack very complex hash algorithms. Figure 4.17 shows the results of cracking bcrypt hashes with *myspace* grammar. The average cracking speeds are multiple times lower than with SHA3. In this case, the two solutions do not differ because the transferred chunks have much lower keyspace since clients can not verify as many hashes as was possible for SHA3. Most of the experiment time is used by hashcat itself, cracking the hashes.

In the previous graphs, one could also notice the cracking speed increases with more hashes. This happens since smaller tasks cannot fully leverage the whole distributed network as the smallest task took only several seconds to crack.

Figure 4.18 shows that the average cracking speed is influenced by the number of connected cracking nodes. While for the smallest task, there is almost no difference with the increasing node count, for the largest task, the speed rises even between 8 and 16 nodes. As this task only takes several minutes, we expect larger tasks would visualize this even better. One can also observe the naive solution using terminal distribution does not scale well. Even though we notice a slight speedup up to 4 nodes, the speed is capped after that even in the largest task because of the network bottleneck mentioned above.

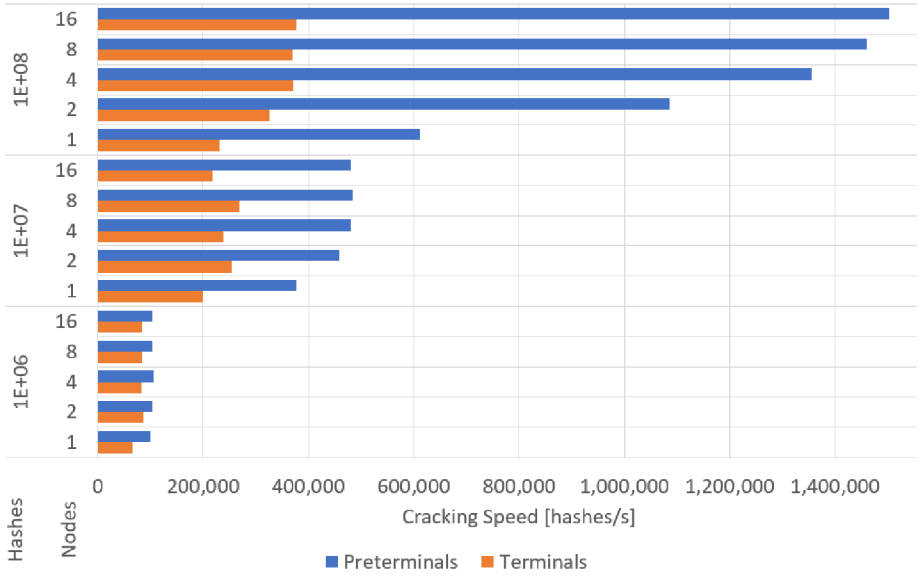


Figure 4.18: Scaling across multiple cracking nodes (SHA3-512 / *myspace* grammar / 100 Mbps network bandwidth)

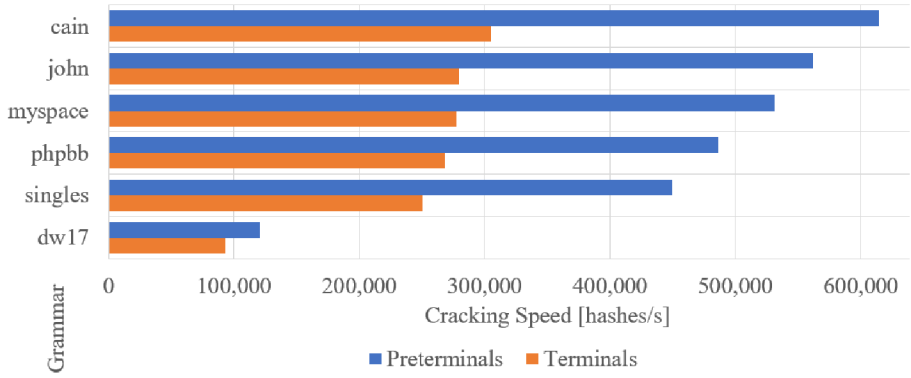


Figure 4.19: Differences in cracking speed among grammars

Grammar Differences

Next, I measure how the choice of a grammar influence the cracking speed. In Figure 4.19, we can see the differences are significant. While the cracking speed of the naive solution is capped by the network bandwidth, results from the proposed solution show generating passwords using some grammars is slower than with others – a phenomenon that is connected with the base structures lengths [82].

Generating passwords from the *Darkweb2017* (dw17) grammar is also very memory demanding because of the long base structures at the beginning of the grammar, and 8GB RAM is not enough for the largest cracking task using the naive solution. With the proposed preterminal-based solution, we encounter no such problem.

Workunit Scrambling

The Deadbeat dad algorithm [211] ran on the server ensures the preterminal structures are generated in a probability order. The same holds for passwords, if generated sequentially. However, in a distributed environment, the order of outgoing chunks and incoming results may scramble due to the non-deterministic behavior of the network. Such a property could be removed by adding an extra intra-node synchronization to the proposed protocol. However, I do not consider that necessary if the goal is to verify all generated passwords in the shortest possible time.

Moreover, the scrambling does not affect the result as a whole. The goal of generating and verifying n most probable passwords is fulfilled. For example, for an assignment of generating and verifying 1 million most probable passwords from a PCFG, our solution generates and verifies 1 million most probable passwords, despite the incoming results might be received in a different order.

Though the scrambling has no impact on the final results, I study the extent of chunk scrambling in our setup. I observe the average difference between the expected and real order of chunks arriving at the server, calling it a *scramble factor* S_f. In the following equation, n is the number of chunks, and r_k is the index where k-th chunk was received:

$$S_f = \frac{1}{n} \sum_{k=1}^{n} |(k - r_k)|. \tag{4.12}$$

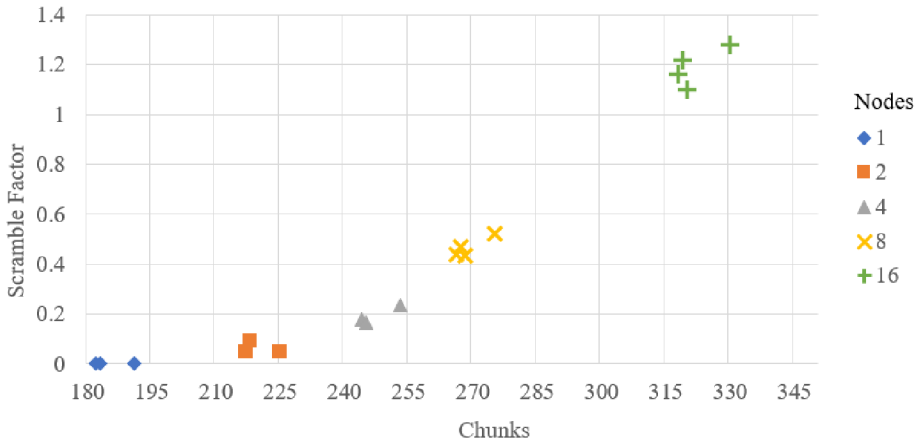


Figure 4.20: Average scramble factor, bcrypt *myspace* grammar capped at 10 milion hashes

I identified two key features affecting the scramble factor – the number of the computing nodes in the system and the number of chunks distributed. In Figure 4.20, one can see that the scramble factor is increasing with the increasing number of chunks and computing nodes. This particular graph represents experiments with bcrypt algorithm, *myspace* grammar, capped at 10 million hashes. Other experiments resulted in a similar pattern. We conclude that the scrambling is relatively low in comparison with the number of chunks and nodes.

Evaluation

The design of the proposed method and the communication protocol was experimentally verified using the proof-of-concept tool with a native hashcat support. I showed that dis-

tributing preterminal structures instead of the final passwords reduces the bandwith requirements dramatically, and in many cases, brings a significant speedup. This fact confirms the hypothesis of Weir et al., who suggested preterminal distribution may be helpful in a distributed password cracking trial [213]. Moreover, I showed how different parameters, such as the hash algorithm, network bandwidth, or the choice of concrete grammar, affect the cracking process.

4.9.4 Summary

The proposed modification of the existing method allowed parallel generating of candidate passwords. As the experiments show, the new parallel solution provides a massive speedup in the password guessing performance. In contrast to the original tool, the new concept can efficiently utilize all available processors. The actual acceleration, however, depends on the concrete grammar. In general, simple base structures parallelize better than complex ones. Therefore, the more simple passwords the training dictionary contains, the higher acceleration we can possibly achieve. Using a faster SSD is potentially beneficial since it allows potentially faster generating of password guesses. Nevertheless, a training dictionary with many complex passwords that create very long base structures complicates the effort. Experiments showed processing these structures creates a bottleneck, in which case there is no advantage of using SSD over HDD.

The second series of experiments analyzed the consequences of grammar filtering. These experimental techniques have two implications: a) the increase in performance, b) the decrease of the total guess count. If configured carefully, the success rate can remain similar to the original. If anyone decides to follow the same way, I suggest starting with removing long base structures. The password guessing performance is much higher when the algorithm does not have to process complex sentential forms. With larger grammars, generating every possible password guess is not possible in an acceptable time. Further filtering of rules is a working alternative to a „hard guess limit.“ It is, however, arguable what rules are „unnecessary.“ While my experiments with the trivial algorithm indicate a possible way, I suppose practical use would require more profound research to find more systematic techniques. For a more advanced approach, I suggest taking preterminal structures into account as well. It would also be advisable to perform additional correction of the probability function to preserve the correctness of the grammar.

Finally, the distributed solution showed to produce excellent results. Breaking the password guessing algorithm into multiple parts computed by different nodes produced many more password guesses per second. Moreover, live cracking with hashcat tool use the available resources efficiently. While the GPU computes cryptographic algorithms, the CPU takes care of generating new candidate passwords, and the TCP/IP stack communicates with the server. The experiments showed that the proposed preterminal distribution saves a lot of network bandwidth, achieves much higher cracking performance, and better scalability over the naive solution.

Chapter 5

Conclusion

While password cracking calculates with relatively little data, it is one of the most computationally complex tasks ever. Deep inside, the security of systems and encrypted media relies upon cryptographic hash functions and encryption algorithms. The more password guesses we make, the more effort we need to put forth for their verification. Another important aspect is the complexity of the verification procedure. Passwords hashed by the obsolete MD5 algorithm or older Microsoft Office documents encrypted by the RC4 cipher are very easy-to-crack. Today's graphical processors can break even longer passwords within a day. Dealing with bcrypt hashes, VeraCrypt-secured disks, or newer Microsoft Office documents using the Agile Encryption is an entirely different story. With serious tasks, cracking sessions may take even years. Luckily, there are ways to accelerate the process, and in the thesis, I described some of them.

5.1 Achievements in Distributed Password Cracking

One direction is verifying multiple password guesses simultaneously. Parallel processing with GPGPU brought a revolution to the password cracking area. Buying a collection of high-end graphics cards is the number one option for a long-term password cracking solution. Yet, there is always a limit to the performance we can achieve with a single computer. And where the single-machine approach fails comes the distributed computing.

Therefore, I studied existing solutions for distributed processing and possible techniques to employ multiple nodes in a single cracking task. Based on my observations, I proposed the design of a general-purpose distributed password cracking system called Fitcrack. The solution uses the BOINC framework that is robust, secure, proven in even world-scale networks. It provides a high level of automation, like the native support for downloading and updating client files. As the cracking engine, Fitcrack uses the open-source hashcat tool because of its ultimate performance, a large scale of supported formats, and various community-driven enhancements. For work assignment, I proposed and experimentally tested a method based on the dynamic distribution of the keyspace and adaptive scheduling algorithm. I also presented the pipeline processing of workunits that reduces the overhead of attacks. Finally, I provided a deep insight into different attack modes. Those include the default ones supported by hashcat, plus the PRINCE and the PCFG attack. For each attack mode, I proposed possible strategies for distributed processing and picked the one that fits my needs the most. The goal was to efficiently divide the job's keyspace into chunks of the desired granularity and allow their fine-grained control over time with minimal overhead.

The experiments showed the proposed strategies work as intended and that the system meets the pre-defined requirements. Moreover, the results bring valuable findings to the password cracking area. I mapped the influence of the hash algorithms and attack modes on the overall efficiency of the task. I studied the scalability of the attacks and the practical implications of different attack configurations. In general, the more complex the cryptographic algorithm, the less matter the choice of attack mode. For instance, the brute-force and dictionary attacks have the same achievable performance on the complex BCrypt with a high number of iterations. However, for MD5 or SHA-1, the brute-force is much more efficient since it generates passwords directly on GPU. Logically, the more we can pre-load to the GPU, the less we need to provide on-the-fly. Therefore, the combination attack is more efficient than the classic dictionary attack, where we need to feed the GPU with new passwords continually.

Moreover, I compared my solution with the Hashtopolis tool. The Hashtopolis is, by design, more low-level and closer to hashcat. The system is, however, not easy to use. I suggest Hashtopolis could be an excellent choice for advanced users who have previous experience with hashcat. It offers a lot of flexibility and allows the user to craft attack commands directly. The Fitcrack is much more user-friendly and usable without any knowledge of hashcat. It provides a high level of abstraction and a lot of automation. For each attack mode, Fitcrack employs a unique strategy that is optimized and fine-tailored for this exact use-case. If appropriately configured, the Fitcrack can utilize the resources more efficiently. For example, the dictionary attack with bigger dictionaries is far more efficient in Fitcrack. Hashtopolis, on the other hand, treats all attack modes the same way and distribute the range of password indexes. While this may not be optimal in all cases, it is much easier to add a completely new attack mode to Hashtopolis.

5.2 Achievements in Probabilistic Methods

Another way to accelerate the cracking process is by reducing the number of password guesses. Instead of all possibilities, we may only check those candidate passwords that are more likely to be correct. Probabilistic methods undisputedly have the potential to help with cracking human-created passwords. In the thesis, I mapped the history from the early time-memory trade-off attack by Martin Hellman, through Markovian models employed in hashcat and John the Ripper tools, to probabilistic context-free grammars (PCFG). In my research, I decided to focus on grammar-based cracking, initially proposed by Matt Weir. The technique benefits from the knowledge obtained by an automated analysis of existing passwords. The analysis creates a mathematical model of users' password creation habits represented by a grammar. Rewrite rules from the grammar serve directly for generating password guesses. This "smart" method generates more probable candidate passwords first and allows for better targeting of attacks. Despite numerous improvements, including the Deadbeat dad algorithm, state-of-the-art solutions were hardly usable for real attacks due to the low performance and missing solution for a parallel or distributed cracking.

The analysis of existing solutions revealed multiple weak spots that created a bottleneck for achieving higher performance. Based on my observations, I proposed a series of improvements. A modification of the existing method allowed parallel processing and therefore utilizing all available processor cores efficiently. An experimental technique of grammar filtering enabled to transform a grammar into a more compact form that is easy-to-process. Finally, I introduced methods that allow distributed grammar-based cracking on multiple nodes. The solution uses the distribution of preterminal structures and allows

running direct cracking sessions with hashcat or other tools. The new ideas were transformed into a proof-of-concept implementation. A series of experiments showed that the proposed improvements introduced a massive speedup in password guessing performance. While the state-of-the-art tools were strictly single-machine, the new distributed solution allows employing a larger network of multiple GPU-equipped nodes. The newly-created tools also serve as modules for Fitcrack's PCFG attack.

5.3 Overall Summary

In the end, I assess parallelization and distribution are the only ways of increasing raw cracking performance today. Due to the laws of physics, further increases in processor frequency are hardly expectable. Quantum computers may one day bring a revolution to the password cracking area, but their practical use is beyond the reach of technology in the foreseeable future. An alternative is employing ASIC chips like in the 1990s EFF DES cracker or the Antminer for bitcoin mining with SHA-256. Such solutions are, however, strictly single-purpose. In contrast, building a large grid or cluster of GPU nodes is relatively easy, and only a matter of funding. With the hardware, it is only about a proper software solution for the distributed cracking, and in my thesis, I showed a series of ways to go. Besides, there is no need to "brute-force everything" when cracking user passwords. Publicly-available datasets of leaked credentials provide an excellent source of knowledge. Password guessing with Markovian chains, PRINCE, and PCFG, are only a few examples of utilizing such knowledge.

5.4 Future Work

I hope the presented work will inspire other researchers and developers in the future. The area of password cracking offers a much larger space for enhancements and new ideas than can fit into a single thesis. The proposed distributed cracking system uses strategies and attacks that are fixed and only applied to individual jobs. It should be possible to create a cracking system that learns over time and uses the findings from previously completed tasks to improve attacks. Moreover, Fitcrack may incorporate a subsystem for rainbow table attack, at least for most common algorithms like MD5, SHA1, or NTLM. Before an actual job starts, the system would perform a fast lookup first. In case of a match, it can crack the password almost immediately. In case a company utilizes multiple Fitcrack servers, one might create an interface that allows for mutual synchronization, e.g., sharing the cache of cracked passwords, dictionaries, Markovian statistics, grammars, and others. The discussed PCFG-based cracking is a powerful method. However, there is still space for improvements. A significant drawback of the technique is that a series of letters is always handled as a single fragment, i.e., represented by one nonterminal. Users often create passwords from multiple words that follow each other without any separator, e.g., "Ilikeapples" that would be handled as a whole. With a knowledge of the individual words, a possible improvement may distinguish between the three parts and allow for generating passwords like "Ilikebananas", "Ilikecars", and others. Another approach may break down the words into syllables and perform substitutions of that level. Last but not least, current probabilistic methods are extremely powerful but mostly focus on the syntax of the password. Semantical-based approaches are extremely rare and could be a subject of future research.

Bibliography

[1] AccessData Corporation: Distributed network attack. [Online; Archived page from: 2000-08-15].
Retrieved from: https://web.archive.org/web/20000815082113/http://www.accessdata.com/dna/index.html

[2] AccessData Corporation: Description of Product: Password Recovery Utilities. October 1996. [Online; Archived page from: 1996-10-23].
Retrieved from:
https://web.archive.org/web/19961023160223/http://www.accessdata.com:80/

[3] AccessData Corporation: Password Recovery Demo. October 1996. [Online; Archived page from: 1996-10-23].
Retrieved from: https://web.archive.org/web/19961023160255/http://www.accessdata.com/datademo.zip

[4] AccessData Group, Inc.: Terms and Conditions. [Online; Accessed: 2020-11-25].
Retrieved from: https://accessdata.com/legal/terms

[5] AccessData Group, Inc.: AccessData Password Recovery Toolkit and Distributed Network Attack 8.2.1: User Guide. 2017.
Retrieved from:
https://ad-pdf.s3.amazonaws.com/dna/8.2.1/PRTK_DNA%20User%20Guide.pdf

[6] Acritum software, win.rar GmbH and RARLAB: RAR file format, version 3.93 – Technical information. [Online; Accessed: 2017-01-03].
Retrieved from: http://acritum.com/winrar/rar-format

[7] Adobe Systems Incorporated: Adobe Supplement to the ISO 32000, BaseVersion: 1.7, ExtensionLevel: 3. June 2008.

[8] Adobe Systems Incorporated: Document management - Portable document format - Part 1: PDF 1.7. Standard ISO 32000-1:2008. International Organization for Standardization. Geneva, Switzerland. July 2008.

[9] Agostini, E.; Bernaschi, M.: BitCracker: BitLocker meets GPUs. *ArXiv preprint:1901.01337, Cornell University.* 2019.

[10] Ah Kioon, M. C.; Wang, Z. S.; Deb Das, S.: Security analysis of MD5 algorithm in password storage. *Applied Mechanics and Materials.* vol. 347. 2013: pp. 2706–2711. ISSN 1662-7482.

[11] Al Fahdi, M.; Clarke, N. L.; Furnell, S. M.: Challenges to digital forensics: A survey of researchers practitioners attitudes and opinions. In *Proceedings of the 12th Information Security for South Africa (ISSA) Conference*. Johannesburg, South Africa. August 2013. ISBN 978-1-4799-2678-7. pp. 1–8. doi:10.1109/ISSA.2013.6641058.

[12] Alexander Peslyak: Parallel and distributed processing with John the Ripper. [Online; Accessed: 2020-11-25].
Retrieved from: https://openwall.info/wiki/john/parallelization

[13] Almasi, G.; Gottlieb, A.: *Highly parallel computing*. Benjamin/Cummings Series in computer science and engineering. January 1988. ISBN 978-0805301779.

[14] An, X.; Jia, H.; Zhang, Y.: Optimized Password Recovery for Encrypted RAR on GPUs. In *Proceedings of the 17th IEEE International Conference on High Performance Computing and Communications*. New York, NY, USA. April 2015. ISBN 978-1-4799-8937-9. pp. 591–598. doi:10.1109/HPCC-CSS-ICESS.2015.270.

[15] Anderson, D. P.: BOINC: A system for public-resource computing and storage. In *Proceedings of the 5th IEEE/ACM International Workshop on Grid Computing*. Pittsburgh, PA, USA. November 2004. ISBN 0-7695-2256-4. pp. 4–10. doi:10.1109/GRID.2004.14.

[16] Anderson, D. P.; Christensen, C.; Allen, B.: Designing a runtime system for volunteer computing. In *Proceedings of the 1st ACM/IEEE conference on Supercomputing*. Tampa, FL, USA. November 2006. ISBN 978-0-7695-2700-0. pp. 126–136. doi:10.1109/SC.2006.24.

[17] Apostal, D.; Foerster, K.; Chatterjee, A.; et al.: Password recovery using MPI and CUDA. In *Proceedings of the 19th International Conference on High Performance Computing*. Pune, India. December 2012. ISBN 978-1-4673-2372-7. pp. 1–9. doi:10.1109/HiPC.2012.6507505.

[18] Aron, L.; Hanáček, P.: Introduction to Android 5 Security. In *Proceedings of the 41st International Conference on Current Trends in Theory and Practice of Computer Science (SOFSEM) 2015*. Pec pod Snezkou, CZ. 2015. ISBN 978-80-87136-20-1. pp. 103–112.
Retrieved from: http://www.fit.vutbr.cz/research/view_pub.php.en?id=10806

[19] Baker, J. K.: Trainable grammars for speech recognition. *The Journal of the Acoustical Society of America*. vol. 65, no. S1. 1979: pp. S132–S132. ISSN 0001-4966.

[20] Bakker, M.; Van Der Jagt, R.: GPU-based password cracking. *University of Amsterdam, System and Network Engineering, Amsterdam, Research*. 2010. Master's thesis.
Retrieved from:
https://homepages.staff.os3.nl/~delaat/rp/2009-2010/p34/report.pdf

[21] Barak, A.; Shiloh, A.: The Virtual OpenCL (VCL) Cluster Platform. In *Proceedings of 4th Intel European Research and Innovation Conference (ERIC)*. Leixlip, Ireland. 2011. pp. 1–5.

[22] Barker, E.; Roginsky, A.: Transitioning the use of cryptographic algorithms and key lengths. Technical Report SP 800-131A Rev. 2. National Institute of Standards and Technology. 2018.

[23] Bédrune, J.-B.; Sigwald, J.: iPhone data protection in depth. Presentation at the 9th annual Hack in The Box Security Conference, Kuala Lumpur, Malaysia. 2011.

[24] Belenko, A.: Overcoming iOS data protection to re-enable iPhone forensics (Elcomsoft Co. Ltd.). Presentation at the 15th Black Hat USA Briefings, Last Vegas, NV, USA. 2011.

[25] Belenko, A.; Sklyarov, D.: Evolution of iOS Data Protection and iPhone Forensics:from iPhone OS to iOS 5. Presentation at Black Hat Abu Dhabi. 2011.

[26] Bengtsson, J.: Parallel Password Cracker: A Feasibility Study of Using Linux Clustering Technique in Computer Forensics. In *Proceedings of the 2nd International Workshop on Digital Forensics and Incident Analysis (WDFIA)*. Samos, Greece. August 2007. ISBN 978-0-7695-2941-7. pp. 75–82. doi:10.1109/WDFIA.2007.10. Retrieved from: doi.ieeecomputersociety.org/10.1109/WDFIA.2007.10

[27] Bertoni, G.; Daemen, J.; Peeters, M.; et al.: Keccak. In *Advances in Cryptology - EUROCRYPT 2013*, edited by T. Johansson; P. Q. Nguyen. Berlin, Heidelberg: Springer Berlin Heidelberg. 2013. ISBN 978-3-642-38348-9. pp. 313–314.

[28] Biham, E.; Anderson, R. J.; Knudsen, L. R.: Serpent: A New Block Cipher Proposal. In *Proceedings of the 5th International Workshop on Fast Software Encryption*. FSE '98. London, UK, UK: Springer-Verlag. 1998. ISBN 978-3-540-64265-7. pp. 222–238. Retrieved from: http://dl.acm.org/citation.cfm?id=647933.740889

[29] Biham, E.; Kocher, P. C.: A known plaintext attack on the PKZIP stream cipher. In *Proceedings of the 2nd International Workshop on Fast Software Encryption (FSE)*, edited by B. Preneel. Leuven, Belgium: Springer Berlin Heidelberg. 1994. ISBN 978-3-540-47809-6. pp. 144–153.

[30] Bishop, M.; Klein, D. V.: Improving system security via proactive password checking. *Computers & Security*. vol. 14, no. 3. 1995: pp. 233–249. ISSN 0167-4048.

[31] Björn König: CLara - OpenCL across the net. [Online; Accessed: 2020-11-28]. Retrieved from: https://sourceforge.net/projects/clara/

[32] Bläsing, T.; Batyuk, L.; Schmidt, A. D.; et al.: An Android Application Sandbox system for suspicious software detection. In *Proceedings of the 5th International Conference on Malicious and Unwanted Software*. Nancy, Lorraine, France. October 2010. ISBN 978-1-4244-9353-1. pp. 55–62. doi:10.1109/MALWARE.2010.5665792.

[33] Blumenthal, U.; Maino, F.; McCloghrie, K.: The Advanced Encryption Standard (AES) Cipher Algorithm in the SNMP User-based Security Model. Request for Comments (RFC) 3826. Internet Engineering Task Force (IETF). June 2004. Retrieved from: http://www.ietf.org/rfc/rfc3826.txt

[34] Bone, B.: GPU Acceleration In PRTK/DNA. April 2015. [Online; Accessed: 2020-11-16].
Retrieved from: https://support.accessdata.com/hc/en-us/articles/204785138-GPU-acceleration-in-PRTK-DNA

[35] Bonneau, J.: The Science of Guessing: Analyzing an Anonymized Corpus of 70 Million Passwords. In *Proceedings of the 33rd IEEE Symposium on Security and Privacy*. San Francisco, CA, USA. May 2012. ISBN 978-1-4673-1244-8. pp. 538–552. doi:10.1109/SP.2012.49.

[36] Brauer, M.; Durusau, P.; Oppermann, L.: *Open Document Format for Office Applications (OpenDocument v1.1)*. OASIS Open. February 2007. OASIS Standard.

[37] Callas, J.; Donnerhacke, L.; Finney, H.; et al.: OpenPGP Message Format. Request for Comments (RFC) 4880. Internet Engineering Task Force (IETF). November 2007. updated by RFC 5581.
Retrieved from: http://www.ietf.org/rfc/rfc4880.txt

[38] Casey, E.: Confronting Encryption in Computer Investigations. In *Proceedings of the 2nd annual Digital Forensic Research Workshop (DFRWS)*. Syracuse, New York, USA. August 2002. pp. 1–27.

[39] Casey, E.: *Handbook of digital forensics and investigation*. Elsevier Academic Press. 2010. ISBN 978-0-12-374267-4.

[40] Casey, E.; Fellows, G.; Geiger, M.; et al.: The growing impact of full disk encryption on digital forensics. *Digital Investigation*. vol. 8, no. 2. 2011: pp. 129–134. ISSN 1742-2876.

[41] Chatterjee, B. B.: New but not improved: a critical examination of revisions to the Regulation of Investigatory Powers Act 2000 encryption provisions. *International Journal of Law and Information Technology*. vol. 19, no. 3. October 2011: pp. 264–284. ISSN 0967-0769. doi:10.1093/ijlit/ear008.
Retrieved from:
https://academic.oup.com/ijlit/article-pdf/19/3/264/2047140/ear008.pdf

[42] Chen, Z.; Han, F.; Cao, J.; et al.: Cloud computing-based forensic analysis for collaborative network security management system. *Tsinghua Science and Technology*. vol. 18, no. 1. February 2013: pp. 40–50. ISSN 1007-0214. doi:10.1109/TST.2013.6449406.

[43] Cho, C.; Chin, S.; Chung, K. S.: Cyber forensic for hadoop based cloud system. *International Journal of Security and its Applications*. vol. 6, no. 3. 2012: pp. 83–90. ISSN 2207-9629.

[44] Choudary, O.; Grobert, F.; Metz, J.: Security Analysis and Decryption of FileVault 2. In *Proceedings of the 9th IFIP WG 11.9 International Conference on Digital Forensics*. Orlando, FL, USA: Springer Berlin Heidelberg. January 2013. ISBN 978-3-642-41148-9. pp. 349–363. doi:10.1007/978-3-642-41148-9_23.

[45] Corel Corporation: WinZIP - AES Encryption Information: Encryption Specification AE-1 an AE-2. January 2009. [Online; Accessed: 2021-01-08].
Retrieved from: https://www.winzip.com/win/en/aes_info.html

[46] The Criminal Procedure Act No. 141/1961 Coll. Trestní řád, zák. č. 141/1961 Sb. Retrieved from:
http://aplikace.mvcr.cz/sbirka-zakonu/ViewFile.aspx?type=c&id=1101

[47] Crumpacker, J. R.: *Distributed password cracking*. Master's Thesis. Naval Postgraduate School, Monterey, California, USA. 2009.

[48] crypt(3) - Linux manual page. April 2018. [Online; Accessed: 2020-11-16]. Retrieved from: https://www.man7.org/linux/man-pages/man3/crypt.3.html

[49] Daemen, J.; Rijmen, V.: AES proposal: Rijndael. October 1999.

[50] Daemen, J.; Rijmen, V.: Announcing the Advanced Encryption Standard (AES). National Institute of Standards and Technology (NIST). November 2001. doi:10.6028/NIST.FIPS.197.
Retrieved from:
http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf

[51] Danalis, A.; Marin, G.; McCurdy, C.; et al.: The Scalable Heterogeneous Computing (SHOC) benchmark suite. In *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units*. Pittsburgh, Pennsylvania, USA. March 2010. ISBN 978-1-60558-935-0. pp. 63–74. doi:10.1145/1735688.1735702.

[52] Das, A.; Bonneau, J.; Caesar, M.; et al.: The Tangled Web of Password Reuse. In *Proceedings of the 21st Network and Distributed System Security (NDSS) Symposium*, vol. 14. San Diego, California. 2014. ISBN 978-1-7138-2071-0. pp. 23–26.

[53] Deutsch, P.: DEFLATE Compressed Data Format Specification version 1.3. Request for Comments (RFC) 1951. Internet Engineering Task Force (IETF). May 1996. Retrieved from: http://www.ietf.org/rfc/rfc1951.txt

[54] Dobbertin, H.; Bosselaers, A.; Preneel, B.: RIPEMD-160: A Strengthened Version of RIPEMD. In *Proceedings of the Third International Workshop on Fast Software Encryption*. London, UK, UK: Springer-Verlag. 1996. ISBN 3-540-60865-6. pp. 71–82.
Retrieved from: http://dl.acm.org/citation.cfm?id=647931.740583

[55] Dougherty, C. R.: Vulnerability Note VU# 836068: MD5 vulnerable to collision attacks. CERT/CC Vulnerability Notes Database, Carnegie Mellon University, Pittsburgh, PA, USA. December 2009.

[56] Du, J.; Li, J.: Analysis the Structure of SAM and Cracking Password Base on Windows Operating System. *International Journal of Future Computer and Communication*. vol. 5, no. 2. 2016: page 112. ISSN 2010-3751.

[57] Duermuth, M.; Angelstorf, F.; Castelluccia, C.; et al.: OMEN: Faster password guessing using an ordered markov enumerator. In *Proceedings of the 7th International Symposium on Engineering Secure Software and Systems (ESSoS)*. Milan, Italy: Springer. 2015. ISBN 978-3-319-15618-7. pp. 119–132.

[58] Elcomsoft Co. Ltd.: Elcomsoft End User License Agreement. [Online; Accessed: 2020-11-25].
Retrieved from: https://www.elcomsoft.com/Elcomsoft_EULA.pdf

[59] Elcomsoft Co. Ltd.: Elcomsoft Distributed Password Recovery 1.0.19 Documentation. February 2006.
Retrieved from: https://web.archive.org/web/20060314183747/http://www.elcomsoft.com/help/edpr/index.html

[60] Elcomsoft Co. Ltd.: Distributed Password Recovery: Benchmarks. April 2020. [Online; Archived page from: 2020-04-06].
Retrieved from: https://web.archive.org/web/20200426111158/https://www.elcomsoft.com/edpr.html#tab_3

[61] Elcomsoft Co. Ltd.: Knowledgebase: Password recovery, How to use Elcomsoft Advanced Attacks. October 2020. [Online; Accessed: 2021-03-02].
Retrieved from: https://support.elcomsoft.com/index.php?/Knowledgebase/Article/View/19/2/how-to-use-elcomsoft-advanced-attacks

[62] Farmer, D.; Spafford, E. H.: The COPS security checker system. In *Proceedings of the Summer Usenix Conference, Anaheim CA*. 1990. pp. 165–170.

[63] Florencio, D.; Herley, C.: A Large-scale Study of Web Password Habits. In *Proceedings of the 16th International Conference on World Wide Web*. WWW '07. New York, NY, USA: ACM. 2007. ISBN 978-1-59593-654-7. pp. 657–666. doi:10.1145/1242572.1242661.

[64] Forensic Focus: Current Challenges In Digital Forensics. May 2016. [Online; Accessed: 2020-11-04].
Retrieved from: https://www.forensicfocus.com/articles/current-challenges-in-digital-forensics/

[65] Forensic Focus: Findings From The Forensic Focus 2018 Survey. October 2018. [Online; Accessed: 2020-11-04].
Retrieved from: https://www.forensicfocus.com/articles/findings-from-the-forensic-focus-2018-survey/

[66] Forget, A.; Chiasson, S.; Van Oorschot, P. C.; et al.: Improving text passwords through persuasion. In *Proceedings of the 4th symposium on Usable privacy and security*. 2008. pp. 1–12.

[67] Gardner, S.: New tool unlocks passwords. *IT World Canada*. vol. 12, no. 1. 2002: page 14.

[68] Garfinkel, S.: Anti-forensics: Techniques, detection and countermeasures. In *Proceedings of the 2nd International Conference on i-Warfare and Security (ICIW)*. Naval Postgraduate School, Monterey, California, USA. 2007. ISBN 978-1-905305-41-4. pp. 77–84.

[69] Garfinkel, S.; Spafford, G.: *Practical UNIX & Internet Security*. O'Reilly & Associates, Inc.. 1999. ISBN 1-56592-148-8.

[70] Ginsburg, S.: The Mathematical Theory of Context Free Languages. *Journal of Symbolic Logic*. vol. 33, no. 2. 1968: pp. 300–301. ISSN 0022-4812. doi:10.2307/2269905.

[71] Grassi, P. A.; Fenton, J. L.; Newton, E.; et al.: NIST special publication 800-63b: digital identity guidelines. *Enrollment and Identity Proofing Requirements*. 2017. Retrieved from: https://pages.nist.gov/800-63-3/sp800-63a.html

[72] Graves, R. E.: *High performance password cracking by implementing rainbow tables on nVidia graphics cards (IseCrack)*. Master's Thesis. Iowa State University, Edinburgh, UK. 2008.
Retrieved from:
https://lib.dr.iastate.edu/cgi/viewcontent.cgi?article=2860&context=etd

[73] Guillaume, Sogeti ESEC Lab: The undocumented password validation algorithm of Adobe Reader X. September 2011. Online; Accessed 2016-12-29].
Retrieved from: https://sogeti33.rssing.com/chan-61982469/all_p12.html

[74] Hafeez, M.; Asghar, S.; Malik, U.; et al.: Survey of MPI Implementations. In *Proceedings of the 1st Digital Information and Communication Technology and Its Applications (DICTAP), Part II*. Dijon, France. 2011. ISBN 978-3-642-22027-2. pp. 206–220.

[75] Hansen, T.: US Secure Hash Algorithms (SHA and HMAC-SHA). Request for Comments (RFC) 4634. Internet Engineering Task Force (IETF). July 2006. obsoleted by RFC 6234.
Retrieved from: http://www.ietf.org/rfc/rfc4634.txt

[76] Hansen, T.: US Secure Hash Algorithms (SHA and SHA-based HMAC and HKDF). Request for Comments (RFC) 6234. Internet Engineering Task Force (IETF). May 2011.
Retrieved from: http://www.ietf.org/rfc/rfc6234.txt

[77] Hartley, B.: PCI Express Expansion Limitations. Technical report. Concurrent Computer Corporation. February 2013.
Retrieved from: https://www.concurrent-rt.com/wp-content/uploads/2016/11/pci-express-expansion-limitations.pdf

[78] Hellman, M.: A cryptanalytic time-memory trade-off. *IEEE transactions on Information Theory*. vol. 26, no. 4. 1980: pp. 401–406. ISSN 0018-9448.

[79] Houshmand, S.; Aggarwal, S.: Using Personal Information in Targeted Grammar-Based Probabilistic Password Attacks. In *Proceedings of the 13th IFIP WG 11.9 International Conference on Digital Forensics*. Springer. 2017. ISBN 978-3-319-67208-3. pp. 285–303.

[80] Houshmand, S.; Aggarwal, S.; Flood, R.: Next Gen PCFG Password Cracking. *IEEE Transactions on Information Forensics and Security*. vol. 10, no. 8. 2015: pp. 1776–1791.

[81] Hranický, R.; Holkovič, M.; Matoušek, P.; et al.: On Efficiency of Distributed Password Recovery. *The Journal of Digital Forensics, Security and Law*. vol. 11, no. 2. 2016: pp. 79–96. ISSN 1558-7215.
Retrieved from: http://www.fit.vutbr.cz/research/view_pub.php?id=11276

[82] Hranický, R.; Lištiak, F.; Mikuš, D.; et al.: On Practical Aspects of PCFG Password Cracking. In *Proceedings of the 33rd IFIP WG 11.3 Annual Conference on Data and Applications Security and Privacy (DBSec)*. Charleston, South Carolina, USA: Springer International Publishing. 2019. ISBN 978-3-030-22479-0. pp. 43–60.

[83] Hranický, R.; and; Ondřej Ryšavý, P. M.; Veselý, V.: Experimental Evaluation of Password Recovery in Encrypted Documents. In *Proceedings of the 2nd International Conference on Information Systems Security and Privacy (ICISSP)*. SciTePress - Science and Technology Publications. 2016. ISBN 978-989-758-167-0. pp. 299–306.
Retrieved from: http://www.fit.vutbr.cz/research/view_pub.php.cs?id=11052

[84] 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.
Retrieved from: https://www.fit.vut.cz/research/publication/11961

[85] Hranický, R.; Zobal, L.; Ryšavý, O.; et al.: Distributed PCFG Password Cracking. In *Computer Security - ESORICS 2020*. Lecture notes in Computer Science. Springer Nature Switzerland AG. 2020. ISBN 978-3-030-58950-9. pp. 701–719.
Retrieved from: https://www.fit.vut.cz/research/publication/12183

[86] Hranický, R.; Zobal, L.; Večeřa, V.; et al.: Distributed Password Cracking in a Hybrid Environment. In *Proceedings of the 9th International Scientific Conference on Security and Protection of Information (SPI)*. University of Defence in Brno. 2017. ISBN 978-80-7231-414-0. pp. 75–90.
Retrieved from: http://www.fit.vutbr.cz/research/view_pub.php?id=11358

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

[88] Huawei: Additional XML Security Uniform Resource Identifiers (URIs). Request for Comments (RFC) 6931. Internet Engineering Task Force (IETF). April 2013.
Retrieved from: http://www.ietf.org/rfc/rfc6931.txt

[89] Industrial Security Research Group: Retrieving NTLM Hashes and what changed in Windows 10. Services & Research UGent-Howest. January 2018. [Online; Accessed: 2020-11-14].
Retrieved from: https://www.insecurity.be/blog/2018/01/21/retrieving-ntlm-hashes-and-what-changed-technical-writeup/

[90] ISO: Information technology - Open Document Format for Office Applications (OpenDocument) v1.0. Standard ISO/IEC 26300:2006. International Organization for Standardization. December 2006.

[91] ISO: Information technology - Document Container File - Part 1: Core. Standard ISO/IEC 21320-1:2015. International Organization for Standardization. 2015.
Retrieved from: https://www.iso.org/obp/ui/#iso:std:60101:en

[92] IWS - The Information Warfare Site: Password Crackers. [Online; Accessed: 2020-11-16].
Retrieved from:
http://www.iwar.org.uk/hackers/resources/digital%20rebels/passwd.htm

[93] Jackal: Cracker Jack, THE Unix Password Cracker [Read Me]. Doc's for Cracker Jack v 1.4. June 1993.
Retrieved from: http://justinakapaste.com/cracker-jack-the-unix-password-cracker-read-me/

[94] Jelinek, F.; Lafferty, J. D.; Mercer, R. L.: Basic methods of probabilistic context free grammars. In *Speech Recognition and Understanding*. Springer. 1992. pp. 345–360.

[95] Jens Steube: Hashcat: Example hashes. [Online; Accessed: 2020-11-25].
Retrieved from: https://hashcat.net/wiki/doku.php?id=example_hashes

[96] Jeremi M Gosney: 8x NVIDIA GTX 1080 Ti Hashcat Benchmarks, Hashcat 3.5.0-22-gef6467b, NVIDIA Driver 381.09. April 2017. [Online; Accessed: 2020-04-06].
Retrieved from:
https://gist.github.com/epixoip/ace60d09981be09544fdd35005051505

[97] Jones, P.: US Secure Hash Algorithm 1 (SHA1). Request for Comments (RFC) 3174. Internet Engineering Task Force (IETF). September 2001. updated by RFCs 4634, 6234.
Retrieved from: http://www.ietf.org/rfc/rfc3174.txt

[98] Kaliski, B.: PKCS #5: Password-Based Cryptography Specification Version 2.0. Request for Comments (RFC) 2898. Internet Engineering Task Force (IETF). September 2000.
Retrieved from: http://www.ietf.org/rfc/rfc2898.txt

[99] Kalyadin, O. A.; Ivanov, A. G.; Belenko, A. V.: Password recovery system and method. October 2006. U.S. Patent 7,809,130.

[100] Kang, S. J.; Lee, S. Y.; Lee, K. M.: Performance Comparison of OpenMP, MPI, and MapReduce in Practical Problems. *Advances in Multimedia*. August 2015: page 9.

[101] Karie, N. M.; Venter, H. S.: Taxonomy of challenges for digital forensics. *Journal of forensic sciences*. vol. 60, no. 4. 2015: pp. 885–893. ISSN 1556-4029.

[102] Karn, P.; Metzger, P.; Simpson, W.: The ESP Triple DES Transform. Request for Comments (RFC) 1851. Internet Engineering Task Force (IETF). September 1995.
Retrieved from: http://www.ietf.org/rfc/rfc1851.txt

[103] Kasabov, A.; van Kerkwijk, J.: Distributed GPU Password Cracking. *Universiteit Van Amsterdam*. May 2011. Research Project 1. Final version rev. 2. Technical report.

[104] Kelley, P. G.; Komanduri, S.; Mazurek, M. L.; et al.: Guess again (and again and again): Measuring password strength by simulating password-cracking algorithms. In *Proceedings of the 33rd IEEE Symposium on Security and Privacy*. San Fransisco, CA, USA: IEEE. 2012. pp. 523–537.

[105] Kim, K.: Distributed password cracking on GPU nodes. In *Proceedings of the 7th International Conference on Computing and Convergence Technology (ICCCT)*. Piscataway, NJ, USA: IEEE. December 2012. pp. 647–650.

[106] Kipper, M.; Slavkin, J.; Denisenko, D.: Implementing AES on GPU - Final report. *University of Toronto, Toronto, Canada*. 2009. Technical report.
Retrieved from: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.447.5356&rep=rep1&type=pdf

[107] Klemm, M.; Dahnken, C.: Recent Processor Technologies and Co-Scheduling. *Co-Scheduling of HPC Applications*. vol. 28. 2017: page 12. Advances in Parallel Computing series.

[108] Kshemkalyani, A. D.; Singhal, M.: *Distributed Computing: Principles, Algorithms, and Systems*. New York, NY, USA: Cambridge University Press. 2008. ISBN 978-0-52-187634-6. First edition.

[109] Kuo, C.; Romanosky, S.; Cranor, L. F.: Human selection of mnemonic phrase-based passwords. In *Proceedings of the second symposium on Usable privacy and security*. 2006. pp. 67–78.

[110] L0pht Holdings LLC: L0phtCrack Password Auditor v7: Documentation. 2019. [Online; Accessed: 2021-02-19].
Retrieved from: https://www.l0phtcrack.com/doc/

[111] Lange, L.: Hackers keep the heat on Windows NT Security. EE Times. April 1997. [Online; Archived page from: 1998-12-05].
Retrieved from: https://web.archive.org/web/19981205132055/http://pubs.cmpnet.com/eet/news/97/950news/hackers.html

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

[113] Luciano, L.; Baggili, I.; Topor, M.; et al.: Digital forensics in the next five years. In *Proceedings of the 13th International Conference on Availability, Reliability and Security*. 2018. pp. 1–14.

[114] Ma, J.; Yang, W.; Luo, M.; et al.: A Study of Probabilistic Password Models. In *Proceedings of the 35th IEEE Symposium on Security and Privacy*. May 2014. ISSN 1081-6011. pp. 689–704. doi:10.1109/SP.2014.50.

[115] Maia, J. D. C.; Urquiza Carvalho, G. A.; Mangueira Jr, C. P.; et al.: GPU linear algebra libraries and GPGPU programming for accelerating MOPAC semiempirical quantum chemistry calculations. *Journal of chemical theory and computation*. vol. 8, no. 9. 2012: pp. 3072–3081. ISSN 1549-9618.

[116] Malyshev, A. E.; Sklyarov, D. V.; Katalov, V. Y.; et al.: Fast cryptographic key recovery system and method. October 2006. U.S. Patent 7,599,492.

[117] Marechal, S.: Advances in password cracking. *Journal in computer virology and hacking techniques*. vol. 4, no. 1. 2008: pp. 73–81. ISSN 2263-8733. Springer.

[118] Marks, M.; Niewiadomska-Szynkiewicz, E.: Hybrid CPU/GPU Platform For High Performance Computing. In *Proceedings of the 28th European Conference on Modelling and Simulation*. rescia, Italy: European Council for Modelling and Simulation (ECMS). May 2014. pp. 523–537. doi:10.7148/2014-0508.

[119] Marr, D. T.; Binns, F.; Hill, D. L.; et al.: Hyper-Threading Technology Architecture and Microarchitecture. *Intel Technology Journal*. vol. 6, no. 1. 2002: page 1. ISSN 1535-864X.

[120] McAtee, M.; Morris, L.: CrackLord: Maximizing Computing Resources. Presentation the 18th Black Hat USA conference, Mandalay Bay, Las Vegas NV, USA. August 2015.

[121] McIlroy, M. D.: *A Research Unix reader: annotated excerpts from the Programmer's Manual, 1971-1986*. AT&T Bell Laboratories. 1987.

[122] McMillan, R.: The World's First Computer Password? It Was Useless Too. *Wired*. January 2012.
Retrieved from: https://www.wired.com/2012/01/computer-password/

[123] Merkle, R. C.: One way hash functions and DES. In *Proceedings of the 6th Conference on the Theory and Application of Cryptology (CRYPTO)*. Springer. 1989. pp. 428–446.

[124] Message Passing Interface Forum: A Message-Passing Interface Standard, Version 3.0. Technical report. University of Tennessee, Knoxville, Tennessee, USA. September 2012. Technical report.

[125] Microsoft Corporation: Chapter 3 - Operating System Installation. *Microsoft Windows 2000 Security Hardening Guide*. March 2009. [Online; Accessed: 2020-11-16].
Retrieved from: https://docs.microsoft.com/en-us/previous-versions/tn-archive/dd277300(v=technet.10)?redirectedfrom=MSDN

[126] Microsoft Corporation: Cryptography and encryption in Office 2016. December 2016. [Online; Accessed: 2020-11-16].
Retrieved from: https://docs.microsoft.com/en-us/DeployOffice/security/cryptography-and-encryption-in-office

[127] Microsoft Corporation: [MS-DOC]: Word (.doc) Binary File Format. December 2019. [Online; Accessed: 2020-11-16].
Retrieved from:
https://docs.microsoft.com/en-us/openspecs/office_file_formats/ms-doc/

[128] Microsoft Corporation: How to use the SysKey utility to secure the Windows Security Accounts Manager database. August 2020. [Online; Accessed: 2020-11-15].
Retrieved from: https://support.microsoft.com/cs-cz/help/310105/how-to-use-the-syskey-utility-to-secure-the-windows-security-accounts

[129] Middleton, B.: *Cyber crime investigator's field guide*. CRC Press. 2001. ISBN 978-0849327681.

[130] Moriarty, K.; Kaliski, B.; Rusch, A.: PKCS# 5: Password-Based Cryptography Specification Version 2.1. Request for Comments (RFC) 8018. Internet Engineering Task Force (IETF). 2017.
Retrieved from: https://tools.ietf.org/html/rfc8018

[131] Morris, R.; Thompson, K.: Password security: A case history. *Communications of the ACM*. vol. 22, no. 11. 1979: pp. 594–597. ISSN 0001-0782.
Retrieved from:
https://rist.tech.cornell.edu/6431papers/MorrisThompson1979.pdf

[132] Motorola Laboratories: HMAC SHA (Hashed Message Authentication Code, Secure Hash Algorithm) TSIG Algorithm Identifiers. Request for Comments (RFC) 4635. Internet Engineering Task Force (IETF). August 2006.
Retrieved from: http://www.ietf.org/rfc/rfc4635.txt

[133] Muffett, A.: Crack Version v5.0 User Manual. December 1996.
Retrieved from:
https://www.techsolvency.com/pub/src/crack-5.0a/c50a/manual.html

[134] Munshi, A.: The OpenCL specification. In *Proceedings of the 21st IEEE Hot Chips Symposium (HCS)*. Standford, CA, USA. August 2009. pp. 1–314.

[135] Murakami, T.; Kasahara, R.; Saito, T.: An implementation and its evaluation of password cracking tool parallelized on GPGPU. In *Proceedings of the 10th International Symposium on Communications and Information Technologies (SoICT)*. Hanoi, Vietnam. October 2010. pp. 534–538.
doi:10.1109/ISCIT.2010.5665047.

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

[137] NIST: FIPS Pub 180-1: Secure Hash Standard. 1995.
Retrieved from:
https://nvlpubs.nist.gov/nistpubs/Legacy/FIPS/NIST.FIPS.180.pdf

[138] NIST: FIPS Pub 180-2: Secure Hash Standard. August 1995.
Retrieved from: https://csrc.nist.gov/csrc/media/publications/fips/180/2/archive/2002-08-01/documents/fips180-2.pdf

[139] NIST: FIPS Pub 202. SHA-3 Standard: Permutation-Based Hash and Extendable-Output Functions. August 2015.
Retrieved from: https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.202.pdf

[140] Nowicki, B.: NFS: Network File System Protocol specification. Request for Comments (RFC) 1094. Internet Engineering Task Force (IETF). March 1989.
Retrieved from: http://www.ietf.org/rfc/rfc1094.txt

[141] NVIDIA Corporation: CUDA Toolkit documentation v10.1.243. 2019.

[142] Oechslin, P.: Making a faster cryptanalytic time-memory trade-off. In *Advances in Cryptology: Proceedings of the 23rd Annual International Cryptology (CRYPTO) Conference*. Santa Barbara, CA, USA: Springer. 2003. pp. 617–630.

[143] Oechslin, P.; Tissieres, C.; Mesot, B.: Ophcrack website. [Online; Accessed: 2021-02-19].
Retrieved from: https://ophcrack.sourceforge.io/

[144] Oleg Afonin: Unlocking BitLocker: Can You Break That Password? Elcomsoft blog. May 2020. [Online; Accessed: 2021-04-13].
Retrieved from: https://blog.elcomsoft.com/2020/05/unlocking-bitlocker-can-you-break-that-password/

[145] OnlineHashCrack: Hashcat Benchmarks on NVIDIA RTX 2080 Ti, Hashcat 6.1.1 , NVIDIA Driver 450.51.06. [Online; Accessed: 2020-11-23].
Retrieved from: https://www.onlinehashcrack.com/tools-benchmark-hashcat-nvidia-rtx-2080-ti.php

[146] Page, A. J.; Naughton, T. J.: Dynamic task scheduling using genetic algorithms for heterogeneous distributed computing. In *Proceedings of the 19th IEEE international parallel and distributed processing symposium (IPDPS)*. Denver, CO, USA: IEEE. April 2005. ISBN 0-7695-2312-9. pp. 189.1–189.8. doi:10.1109/IPDPS.2005.184.

[147] Palmer, G. L.: A Road Map for Digital Forensic Research. Technical report. First Digital Forensic Research Workshop (DFRWS). 2001.

[148] Passware: Passware Kit Forensic 2020: Quck Start Guide. [Online; Accessed: 2020-11-16].
Retrieved from: https://files.passware.com/resources/quickstart/PasswareKit2020-QuickStartGuide.pdf

[149] Passware, Inc.: Software License Agreement for Passware Software. [Online; Accessed: 2020-11-25].
Retrieved from: https://www.passware.com/files/Passware-EULA.pdf

[150] Passware, Inc.: Password Recovery Software. December 1998. [Online; Archived page from: 1998-12-12].
Retrieved from: https://web.archive.org/web/19981212030639/http://www.lostpassword.com:80/

[151] Passware, Inc.: Distributed Password Recovery. February 2010. [Online; Archived page from 2010-02-06].
Retrieved from: https://web.archive.org/web/20100206114223/http://www.lostpassword.com:80/distributed-password-recovery.htm

[152] Passware, Inc.: Passware Kit Forensic 2020 v3: File Types. 2020. [Online; Archived page from: 2020-08-05].
Retrieved from: https://web.archive.org/web/20200805132554/https://www.passware.com/kit-forensic/filetypes/

[153] Passware, Inc.: Passware Kit Forensic 2020 v3: Performance. 2020. [Online; Archived page from: 2020-08-05].

Retrieved from: https://web.archive.org/web/20200805135531/https://www.passware.com/kit-forensic/performance/

[154] Pavlov, D.; Veerman, G.: Distributed Password Cracking Platform. Univeriteit van Amsterdam, System & Network Engineering. February 2012. Technical report. Retrieved from: https://www.os3.nl/_media/2011-2012/courses/rp1/p13_report.pdf

[155] Pavlov, I.: 7-Zip method IDs for 7z and xz archives. June 2015. [Online; Accessed: 2017-01-03]. Retrieved from: http://cpansearch.perl.org/src/BJOERN/Compress-Deflate7-1.0/7zip/DOC/Methods.txt

[156] Pavlov, I.: 7z Format. 2015. [Online; Accessed: 2016-12-20]. Retrieved from: http://www.7-zip.org/7z.html

[157] Percival, C.; Josefsson, S.: The scrypt password-based key derivation function. Request for Comments (RFC) 7914. Internet Engineering Task Force (IETF). 2016. Retrieved from: https://tools.ietf.org/html/rfc7914.html

[158] Peslyak, A.: John the Ripper Changelog. [Online; Accessed: 2020-11-16]. Retrieved from: https://www.openwall.com/john/doc/CHANGES.shtml

[159] Peslyak, A.: John users: When was John created? [Online; Accessed: 2020-11-16]. Retrieved from: https://www.openwall.com/lists/john-users/2015/09/10/4

[160] Peslyak, A.; Marechal, S.: Password security: past, present, future. Presentation at Passwords 12: International Conference on Password Security, Oslo, Norway. 2012. Retrieved from: https://www.openwall.com/presentations/Passwords12-The-Future-Of-Hashing/

[161] Peterson, W. W.; Weldon, E. J.: *Error-correcting codes*. MIT press. 1972. ISBN 978-0-26-216039-1.

[162] Pippin, A.; Hall, B.; Chen, W.: Parallelization of John the Ripper Using MPI (Final Report). University of California, Santa Barbara, CA, USA. 2006. Technical report CS240A.

[163] PKWARE, Inc.: APPNOTE.TXT - .ZIP File Format Specification. September 2014. [Online; Accessed 2015-11-17]. Retrieved from: https://pkware.cachefly.net/webdocs/casestudies/APPNOTE.TXT

[164] Popov, A.: Prohibiting RC4 Cipher Suites. Request for Comments (RFC) 7465. Internet Engineering Task Force (IETF). February 2015. Retrieved from: http://www.ietf.org/rfc/rfc7465.txt

[165] Potter, B.: A review of L0phtCrack 6. *Network security*. vol. 2009, no. 7. 2009: pp. 14–17. ISSN 1353-4858. doi:10.1016/S1353-4858(09)70089-3. Elsevier.

[166] Proctor, R. W.; Lien, M.-C.; Vu, K.-P. L.; et al.: Improving computer security for authentication of users: Influence of proactive password restrictions. *Behavior Research Methods, Instruments, & Computers*. vol. 34, no. 2. 2002: pp. 163–169.

[167] Provos, N.; Mazieres, D.: A Future-Adaptable Password Scheme. In *Proceedings of the 5th USENIX Annual Technical Conference (ATC), FREENIX Track*. Monterey, CA, USA. 1999. ISBN 1-880446-33-2. pp. 81–91.

[168] Rabin, M. O.: Probabilistic automata. *Information and control*. vol. 6, no. 3. 1963: pp. 230–245. ISSN 0019-9958. doi:10.1016/S0019-9958(63)90290-0.

[169] Rabiner, L. R.: A tutorial on hidden Markov models and selected applications in speech recognition. *Proceedings of the IEEE*. vol. 77, no. 2. February 1989: pp. 257–286. ISSN 0018-9219. doi:10.1109/5.18626.

[170] Reyes-Ortiz, J. L.; Oneto, L.; Anguita, D.: Big Data Analytics in the Cloud: Spark on Hadoop vs MPI/OpenMP on Beowulf. *Procedia Computer Science*. vol. 53. 2015: pp. 121 – 130. ISSN 1877-0509. doi:0.1016/j.procs.2015.07.286.
Retrieved from:
http://www.sciencedirect.com/science/article/pii/S1877050915017895

[171] Rivest, R.: The MD4 Message-Digest Algorithm. Request for Comments (RFC) 1320. Internet Engineering Task Force (IETF). April 1992. obsoleted by RFC 6150.
Retrieved from: http://www.ietf.org/rfc/rfc1320.txt

[172] Rivest, R.: The MD5 Message-Digest Algorithm. Request for Comments (RFC) 1321. Internet Engineering Task Force (IETF). April 1992. updated by RFC 6151.
Retrieved from: http://www.ietf.org/rfc/rfc1321.txt

[173] Robinson, D.; Coar, K.: The Common Gateway Interface (CGI) Version 1.1. Request for Comments (RFC) 3875. Internet Engineering Task Force (IETF). October 2004.
Retrieved from: http://www.ietf.org/rfc/rfc3875.txt

[174] Robling Denning, D. E.: *Cryptography and data security*. Addison-Wesley Longman Publishing Co., Inc.. 1982. ISBN 978-0-201-10150-8.

[175] Roshal, A.: RAR 5.0 archive format. Technical report. RARLab. 2001. [Online; Accessed: 2017-01-03].
Retrieved from: http://www.rarlab.com/technote.htm

[176] Roussev, V.: Digital forensic science: issues, methods, and challenges. *Synthesis Lectures on Information Security, Privacy, & Trust*. vol. 8, no. 5. 2016: pp. 1–155. ISSN 1945-9742. Morgan & Claypool Publishers.

[177] Roussev, V.; Wang, L.; Richard, G.; et al.: A Cloud Computing Platform for Large-Scale Forensic Computing. In *Proceedings of the 5th IFIP WG 11.9 International Conference on Digital Forensics*. Orlando, Florida, USA: Springer Berlin Heidelberg. January 2009. ISBN 978-3-642-04155-6. pp. 201–214. doi:10.1007/978-3-642-04155-6_15.
Retrieved from: http://dx.doi.org/10.1007/978-3-642-04155-6_15

[178] Samek, J.: *Virtul GPU cluster*. Bachelor's thesis. Faculty of Information Technology. Czech technical university in Prague. 2016.

[179] Sanders, C.: How I Cracked your Windows Password (Part 1). *TechGenix*. January 2010. [Online; Accessed: 2020-11-16].
Retrieved from: http://techgenix.com/how-cracked-windows-password-part1/

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

[181] Schneier, B.; Kelsey, J.; Whiting, D.; et al.: *Twofish: A 128-Bit Block Cipher*. New York, NY, USA: John Wiley & Sons, Inc.. June 1998. ISBN 0-471-35381-7.

[182] Schuermann, K. U.: *Decrypting Open Document Format (ODF) Files*. Ringlord Technologies. June 2013. Technical report.
Retrieved from: https://ringlord.com/dl/Decrypting%20ODF%20Files.odt

[183] Shehhi, H. A.; Hamdi, D. A.; Asad, I.; et al.: A Forensic Analysis Framework for Recovering Encryption Keys and BB10 Backup Decryption. In *Proceedings of the 12th Annual International Conference on Privacy, Security and Trust (PST)*. Toronto, Canada. July 2014. ISBN 978-1-47-993504-8. pp. 172–178. doi:10.1109/PST.2014.6890937.

[184] Shuanglei, Z.: RainbowCrack 1.0 README. September 2003.
Retrieved from: https://web.archive.org/web/20080705140750/http://www.antsight.com/zsl/rainbowcrack#Download

[185] Shuanglei, Z.: RainbowCrack 1.8 README. August 2020.
Retrieved from: https://project-rainbowcrack.com

[186] Shuanglei, Z.: RainbowCrack Documentation. 2020. [Online; Accessed: 2021-02-19].
Retrieved from: https://project-rainbowcrack.com/documentation.htm

[187] Sprengers, M.: GPU-based Password Cracking: On the Security of Password Hashing Schemes regarding Advances in Graphics Processing Units. *Radboud University Nijmegen*. 2011. Master's thesis.
Retrieved from: https://www.ru.nl/publish/pages/769526/thesis.pdf

[188] Stallings, W.: The Whirlpool Secure Hash Function. *Cryptologia*. vol. 30. September 2006: pp. 55–67. ISSN 0161-1194. doi:10.1080/01611190500380090.

[189] Steigner, C.; Wilke, J.: Performance Tuning of Distributed Applications with CoSMoS. In *Proceedings of the 21st International Conference on Distributed Computing Systems*. Mesa, AZ, USA: IEEE. August 2001. ISBN 0-7695-1077-9. pp. 173–180. doi:10.1109/ICDSC.2001.918946.

[190] Steinkraus, D.; Buck, I.; Simard, P.: Using GPUs for Machine Learning Algorithms. In *Proceedings of the 8th International Conference on Document Analysis and Recognition (ICDAR'05)*. Seoul, South Korea: IEEE. September 2005. ISBN 0-7695-2420-6. pp. 1115–1120. doi:10.1109/ICDAR.2005.103.

[191] Steube, J.: hashcat-legacy. Hashcat Wiki. [Online; Accessed: 2020-11-16].
Retrieved from: https://hashcat.net/wiki/doku.php?id=hashcat-legacy

[192] Steube, J.: Hashcat Wiki: Description. [Online; Accessed: 2021-03-02]. Retrieved from: https://hashcat.net/wiki/doku.php?id=hashcat

[193] Steube, J.: oclHashcat (old version). [Online; Accessed: 2020-11-16]. Retrieved from: https://hashcat.net/wiki/doku.php?id=oclhashcat_old

[194] Tarjan, R.: Depth-first Search and Linear Graph Algorithms. *SIAM Journal on Computing*. vol. 1, no. 2. 1972: pp. 146–160. ISSN 0097-5397. doi:10.1137/0201010.

[195] Teufl, P.; Fitzek, A.; Hein, D.; et al.: Android encryption systems. In *2014 International Conference on Privacy and Security in Mobile Systems (PRISMS)*. May 2014. pp. 1–8. doi:10.1109/PRISMS.2014.6970599.

[196] The TrueCrypt Foundation: TrueCrypt User's Guide, version 7.1a. February 2012.

[197] Tilstone, W. J.; Savage, K. A.; Clark, L. A.: *Forensic science: An Encyclopedia of History, Methods, and Techniques*. ABC-CLIO. 2006. ISBN 1-57607-194-4.

[198] Turan, M. S.; Barker, E.; Burr, W.; et al.: Recommendation for password-based key derivation. U.S. Department of Commerce and National Institute of Standards and Technology. 2010. NIST Special Publication 800-132.

[199] Turner, S.; Chen, L.: Updated Security Considerations for the MD5 Message-Digest and the HMAC-MD5 Algorithms. Request for Comments (RFC) 6151. Internet Engineering Task Force (IETF). March 2011. Retrieved from: http://www.ietf.org/rfc/rfc6151.txt

[200] U. S. Const. amend. V. Art. 3. Retrieved from: https://constitution.congress.gov/browse/amendment-5/

[201] UK Public General Acts, 2000 c. 23.: Regulation of Investigatory Powers Act 2000, Part III: Investigation of electronic data protected by encryption etc. Retrieved from: https://www.legislation.gov.uk/ukpga/2000/23/part/III

[202] U.S. Department of Commerce/National Institute of Standards and Technology (NIST): Data encryption standard (DES). *Federal Information Processing Standard Publication 46-3*. 1999.

[203] Van De Zande, P.: The Day DES Died. *SANS Institute, Information Security Reading Room*. July 2001.

[204] Van Vleck, T.: How the Air Force cracked Multics Security. *Web Article, Multicians. org web site*. 1995. [Online; Accessed: 2020-11-09]. Retrieved from: https://www.multicians.org/security.html

[205] Venema, W. Z.: Murphy's Law and Computer Security. In *Proceedings of the 6th USENIX Security Symposium, Focusing on Applications of Cryptography*. San Jose, California, USA. 1996. ISBN 978-1-71-380388-1. pp. 187–193. doi:10.5555/1267569.

[206] Veras, R.; Collins, C.; Thorpe, J.: On Semantic Patterns of Passwords and their Security Impact. In *Proceedings of the 21st Network and Distributed System Security (NDSS) Symposium*. 2014. ISBN 1-891562-35-5. pp. 386–401. doi:10.14722/ndss.2014.23103.

[207] Vu, K.-P. L.; Proctor, R. W.; Bhargav-Spantzel, A.; et al.: Improving password security and memorability to protect personal and organizational information. *International Journal of Human-Computer Studies*. vol. 65, no. 8. 2007: pp. 744 – 757. ISSN 1071-5819. doi:10.1016/j.ijhcs.2007.03.007.

[208] Walker, D. W.: Standards for message-passing in a distributed memory environment. Oak Ridge National Lab., TN, USA. August 1992. Technical report ORNL/TM-12147; CONF-9204185-Summ. ON: DE92019391; OSTI 7104668. Retrieved from: https://www.osti.gov/biblio/7104668

[209] Walker, D. W.; Dongarra, J. J.: MPI: a standard message passing interface. *Supercomputer*. vol. 12. 1996: pp. 56–68.

[210] Waltermann, R. D.; Challener, D. C.; Childs, P. L.; et al.: Method for protecting security accounts manager (SAM) files within windows operating systems. October 2010. U.S. Patent 7818567.

[211] Weir, C. M.: *Using probabilistic techniques to aid in password cracking attacks*. PhD. Thesis. Florida State University, Tallahassee, FL, USA. 2010.

[212] Weir, M.; Aggarwal, S.; Collins, M.; et al.: Testing metrics for password creation policies by attacking large sets of revealed passwords. In *Proceedings of the 17th ACM conference on Computer and communications security*. 2010. ISBN 978-1-4503-0245-6. pp. 162–175. doi:10.1145/1866307.1866327.

[213] Weir, M.; Aggarwal, S.; d. Medeiros, B.; et al.: Password Cracking Using Probabilistic Context-Free Grammars. In *Proceedings of the 30th IEEE Symposium on Security and Privacy*. Oakland, California, USA. May 2009. ISBN 978-0-7695-3633-0. pp. 391–405. doi:10.1109/SP.2009.8.

[214] Wu, X.; Hong, J.; Zhang, Y.: Analysis of OpenXML-based office encryption mechanism. In *Proceedings of the 7th International Conference on Computer Science Education (ICCSE)*. Melbourne, VIC, Australia. July 2012. ISBN 978-1-4673-0242-5. pp. 521–524. doi:10.1109/ICCSE.2012.6295128.

[215] Yang, C.-T.; Huang, C.-L.; Lin, C.-F.: Hybrid CUDA, OpenMP, and MPI parallel programming on multicore GPU clusters. *Computer Physics Communications*. vol. 182, no. 1. 2011: pp. 266–269. ISSN 0010-4655. doi:10.1016/j.cpc.2010.06.035.

[216] Yi-ming, J.; Sheng-li, L.: The Analysis of Security Weakness in BitLocker Technology. In *Proceedings of the 2nd International Conference on Networks Security, Wireless Communications and Trusted Computing*, vol. 1. Wuhan, Hubei, China. April 2010. ISBN 978-1-4244-6598-9. pp. 494–497. doi:10.1109/NSWCTC.2010.123.

[217] Zeilenga, K.: SASLprep: Stringprep Profile for User Names and Passwords. Request for Comments (RFC) 4013. Internet Engineering Task Force (IETF). February 2005. obsoleted by RFC 7613.
Retrieved from: http://www.ietf.org/rfc/rfc4013.txt

[218] Zhang, L.; Zhou, Y.; Fan, J.: The forensic analysis of encrypted Truecrypt volumes. In *2014 IEEE International Conference on Progress in Informatics and Computing*. May 2014. pp. 405–409. doi:10.1109/PIC.2014.6972366.

[219] Zhang, L.; Zhou, Y.; Fan, J.: The forensic analysis of encrypted Truecrypt volumes. In *Proceedings of the 2nd IEEE International Conference on Progress in Informatics and Computing*. Shanghai, China. May 2014. ISBN 978-1-4799-2030-3. pp. 405–409. doi:10.1109/PIC.2014.6972366.

[220] Zivadinovic, M.; Milenkovic, I.; Simic, D.: Cash, hash or trash-hash function impact on system security. In *Proceedings of the 15th SymOrg International Symposium, ICT and Management section*. Zlatibor, Serbia. June 2016. pp. 788–791.

[221] Zonenberg, A.: Distributed Hash Cracker: A Cross-Platform GPU-Accelerated Password Recovery System. Rensselaer Polytechnic Institute, Troy, NY, USA. May 2009. Technical report.

Appendix A

An overview of password-protected formats

This chapter provides an overview of selected password-protected formats. For each, I discuss the encryption mechanisms involved and denote the password verification procedure.

A.1 Documents

Digital documents mainly consist of text and images, however, some formats like PDF may contain video and sound content as well [8]. Some document formats (like PDF) are determined for publishing, some are suitable for further editing and expanding its content.

A.1.1 Portable Document Format

Portable Document Format (PDF) is primarily used for publishing documents and is designed to be read-only. Nevertheless, Adobe Acrobat Pro[1] some non-official tools can edit documents in this format. The main contribution of PDF is portability, making the document viewable on multiple platforms using different software. PDF documents are designed to look the same on all devices and browsers.

Each file begins with a `%PDF-x` signature, where x is the number of the PDF version. PDF uses a non-binary format and consists of objects like arrays and dictionaries, while each object has its unique ID. At the end of each file, there is the address of *XREF* table, which defines the position of all document's objects according to their ID. The XREF table is followed by the XREF trailer. The *trailer* is a structure that provides information for viewing software about how to read the document. It also contains the `Encrypt` flag telling whether the document is encrypted or not. Inside the trailer, there is the ID of an object called the *encryption dictionary* [8]. The exact content of the encryption dictionary depends on the document's version, Table A.1 shows its typical entries.

Older versions of PDF were encrypted with the RC4 stream cipher, now considered obsolete for known weaknesses [164]. Thus, newer versions of PDF use the AES algorithm [50]. For securing PDF documents, two passwords can be used: a *user password*, and an *owner password*. The owner password is only used to restrict selected operations like printing or extracting content, and can be ignored. The analyst is usually looking for the user password which is the actual password used for encryption of the document's contents.

[1] https://acrobat.adobe.com/hk/en/acrobat/acrobat-pro.html

Key	Type and length	Value
V	1-byte integer	A code specifying the algorithm for encryption and decryption of the document
R	1-byte integer	A number specifying the *security revision*.
O	32-byte string for $R \leq 4$ 48-byte string for $R \geq 5$	A string based on both *owner* and *user* passwords. It is used for computing the encryption key and for verification of the owner password.
U	32-byte string for $R \leq 4$ 48-byte string for $R \geq 5$	A string based on the *user password*. It is used for verification of the user password.
OE	32-byte string (only present if $R \geq 5$)	A string based on both *owner* and *user* passwords. It is used for computing the encryption key.
UE	32-byte string (only present if $R \geq 5$)	A string based on the *user password*, also used for computing the encryption key.
P	4-byte integer	It represents a vector of flags specifying permitted operations when the document is opened with user or owner access.
Perms	16-byte string (only present if $R \geq 5$)	A copy of permission flags encrypted with the file encryption key.
Encrypt metadata	boolean	Indicates whether the document-level metadata is encrypted as well. The default value is true.

Table A.1: Typical entries of the PDF encryption dictionary

The actual password verification process depends on the *security revision* (**R**):

- **In revisions 1 and 2** - the password in Latin-1 encoding is aligned to 32-bytes (by a vector of defined values called "passpad"), and concatenated with **O**, **P** strings and the document trailer's ID. This is for creating different encryption keys for documents, where the same passwords are used. From the result of the concatenation, MD5 hash is computed. The resulting hash represents the *encryption key*. The key is then used to encrypt the defined ("passpad") value using the RC4 stream cipher. The result is compared to **U**. If the values match, the password is correct [8].

- **Revision 3** uses the same principle, however, MD5 hashing is applied 50 times, and the RC4 encryption is performed 20 times [172].

- **In Revision 4**, the verification of a user password remains the same, however, if metadata is not encrypted (i.e. *EncryptMetadata* is false), the concatenation used for creating the encryption key is extended by `0xFFFFFFFF` value.

- **Revision 5** started with Adobe Acrobat 9 and PDF 1.7 documents with *Adobe Extension Level 3*. The Latin-1 encoding is replaced by UTF-8 and a different principle for password verification is used. It also introduces salt (see Section 2.2) to prevent the *rainbow table attack* [142]. From Table A.1, it is clear that starting from revision 5, **O** and **U** values were extended by 16 bytes. The first 8 bytes are called the *validation salt*, and the second 8 bytes are called the *key salt*. For verification, the password in the UTF-8 form is firstly processed with the SASLprep function [217]. Then it is concatenated with the *validation salt* from **U**. From the "salted" password,

an SHA-256 hash is computed. The resulting hash is then compared to the first 32 bytes of **U**. If the values match, the password is considered correct [7].

- **Revision 6** was introduced with Adobe Acrobat X and PDF 2.0 documents. The specification should be defined by *ISO 32000-2* which is not yet publicly available. Despite the absence of the official specification, the behavior of the new revision has been analyzed and the verification algorithm has been derived. The pseudo-code is available at Sogeti ESEC Lab webpage[2]. I personally agree with the author, that the algorithm itself is quite unique and does not resemble any known one. For verification, three variants of SHA hashing function are used: SHA-256, SHA-384, SHA-512 [75]. In addition, AES encryption algorithm with 128-bit key in CBC mode is also involved [50]. The password in the UTF-8 encoding is combined with 8 or 56 bytes of salt, producing a 256-bit hash. Then there can be up to 4096 iterations, while in each phase, one of the SHA functions, and an AES encryption are performed. The result of the algorithm itself is again a 32-byte hash, which is compared to **U** like in previous previsions [73].

Not all records from the *encryption dictionary* are used for the verification of the *user password*. Many pieces of metadata have use only when the document is decrypted, or viewed. However, Once the user password is discovered, the contents can decrypted and viewed by any document viewer that supports encrypted PDFs. For implementing automated work with PDF documents, Poppler library[3] can provide many useful features.

A.1.2 Microsoft Office - up to 2003

Before the release of Office Open XML format (see Section A.1.3), Microsoft Office documents used a binary format. However, as shown in Table A.2, the encryption techniques changed over the versions:

- **Version 95 and older versions** - used a weak encryption based on *XOR obfuscation*. A password was transformed into a 16-bit encryption key. With today's software and hardware, the password can be found immediately[4].

- **Versions 97 and 2000** - switched to the RC4 cipher with a 40-bit key. For password verification, the MD5 function is also needed. All necessary metadata are located in a structure called *EncryptionHeader*. Those include *salt*, *EncryptedVerifier*, and *EncryptedVerifierHash*; all are 16-byte values. The password is hashed using the MD5 function, and the first 5 bytes of hash are concatenated with a 16-byte *salt*. The resulting 21-byte value is copied 16 times into a 336-byte buffer, on which, MD5 is applied again. From the result, first 5 bytes are used again and concatenated with a 32-bit data block chosen by the application – for cracking, the block may contain zeros. The result is hashed with MD5 for the last time. The first 128 bits of the hash create the *encryption key*. With the key, *EncryptedVerifier* and *EncryptedVerifierHash* are decrypted. The decrypted verifier is hashed with MD5. If the computed hash matched the decrypted verifier hash, the password is correct. This approach is used for MS Word (DOC) and MS Excel (XLS) documents. MS PowerPoint documents did not support encryption in versions 97 and 2000 [127].

[2] https://sogeti33.rssing.com/chan-61982469/all_p12.html
[3] https://poppler.freedesktop.org/
[4] http://www.password-crackers.com/en/articles/12/#3.2

- **Versions XP and 2003** - introduced support for a longer 128-bit RC4 encryption key, and the *CryptoAPI EncryptionHeader*. The header contains values similar to the classic *EncryptionHeader* from older versions, extended with *KeySize* and *VerifierHashSize*. The password verification method is, however, slightly different. The password in Unicode UTF-16LE encoding is concatenated with *salt* and hashed by SHA-1 [75]. The resulting hash is concatenated with a 32-bit data block (which again could contain zeros), and hashed with SHA-1 again. The prefix with the length of *KeySize* represents the *encryption key*. The rest is identical to the 97/2000 technique. The only difference is that *EncryptedVerifierHash* is 20 bytes long. MS PowerPoint documents (PPT) support this encryption as well [127].

Office version	Hash function	Encryption algorithm	Key length
95 and older	-	XOR obfuscation	16-bit
97, 2000	MD5	RC4	40-bit
XP, 2003	SHA-1	RC4	40 to 128-bit
2007	SHA-1 (50,000 rounds)	AES	128/192/256-bit
2010	SHA-1 (100,000 rounds)	AES	128/192/256-bit
2013, 2016	SHA-2 (100,000 rounds)	AES	128/192/256-bit

Table A.2: Cryptographic algorithms used in different MS Office versions

A.1.3 Microsoft Office - Office Open XML

Starting from MS Office 2007, Microsoft introduced *Office Open XML* (OOXML) document format, using the document extensions like DOCX, XLSX, PPTX, etc. The document is in fact a ZIP archive containing multiple subdirectories and files with either XML or binary data. Encryption is applied to the files themselves, not on the entire archive. In OOXML, the RC4 algorithm was replaced by AES [214]. In Office 2007 and 2010, the AES encryption key is 128-bit long by default, which was replaced by 256-bit in Office 2013. By editing registry entries, local security policy, or domain group policy, the length can be recondigured. Supported key lengths are 128, 192, or 256 bits[5]. In the root directory, there is a file called *EncryptedPackage* with the encrypted data. The root directory also contains the *EncryptedInfo* file with information about the hash function used, the encryption algorithm, cryptographic salt, and other necessary metadata. OOXML supports two encryption methods:

- **Standard encryption** - is used in version 2007, and the encryption metadata is stored in *EncryptionHeader* with binary format. The structure is similar to CryptoAPI from versions XP and 2003. The only difference is in flags and encryption algorithm. A 32-bit *AlgID* value defines the length of AES encryption key, which can be 128, 192, or 256 bytes long [50]. The *salt* is concatenated with Unicode-encoded password, and hashed with SHA-1 [75]. Then, 50,000 iterations of SHA-1 are performed on a string consisting made an *iterator* concatenated with the result from the previous iteration. The *iterator* is a 32-bit value starting from zero and linearly increasing up to 49,000. After that, the resulted hash is concatenated with the 32-bit

[5] https://msdn.microsoft.com/en-us/library/cc313071%28v=office.12%29.aspx

data block (which again could be zeros) and hashed with SHA-1 for the last time. Let the resulting hash be X_h. Unlike with CryptoAPI, the encryption key is not created yet [214].

$$X3 = X1 \cdot X2 \tag{A.1}$$

To obtain the encryption key, one needs to create a 64-bit buffer with all bytes set to `0x36`. Then, it is necessary to perform XOR of the first 20 bytes in buffer with X_h. The result is called $X1$. The same procedure is performed again with buffer bytes set to `0x5C`. The result is called $X2$. As shown in Equation A.1, concatenation of $X1$ with $X2$ gives value $X3$. The *encryption key* is the prefix of $X3$ with length defined by *KeySize*. The rest of the verification is identical with the CryptoAPI (see Section A.1.2).

Key	Description	Values
saltSize	Size of the salt in bytes	up to 64
blockSize	Size of a single encrypted block	possibilities depend on cipherAlgorithm
keyBits	Length of encryption key in bits	128 (default in 2010) 192 256 (default in 2013/16)
spinCount	Number of hash function iterations	100,000 (default) 10,000,000 (maximum)
cipherAlgorithm	Algorithm used for encryption	**AES** (default), RC2, DES, DESX, 3DES, 3DEX_112
cipherChaining	Mode of encryption algorithm	CBC (default) CFB
hashAlgorithm	Algorithm used for hashing	**SHA-1** (default in 2010), SHA-256, SHA-384, **SHA-512** (default in 2013/16), MD2, MD4, MD5, RIPEMD-128, RIPEMD-160, WHIRLPOOL

Table A.3: OOXML: Agile encryption options

- **Agile encryption** - The agile encryption was introduced in MS Office 2010 and by design supports hash and encryption algorithms that are accessible from the Windows API [126]. Unlike in previous versions, the *EncryptionInfo* has XML format enabling to defined different security options. The most important ones are shown in Table A.3.

  The derivation of the *encryption key* is similar to the *standard encryption*, however, different keys are created for decryption of *encryptedVerifier* and *encryptedVerifierHash*. The *salt* in base64 encoding is converted to a binary form and concatenated with a Unicode-encoded password. The result is hashed by a given *HashAlgorithm*. The number of hashing iterations is *spinCount - 1*. The result is concatenated with a *blockKey*. The values are different for both keys. For decrypting *encryptedVerifier*, it

is `0xFE, 0xA7, 0xD2, 0x76, 0x3B, 0x4B, 0x9E, 0x79` sequence. For decrypting *encryptedVerifierHash*, it is `0xD7, 0xAA, 0x0F, 0x6D, 0x30, 0x61, 0x34, 0x4E` sequence. For each key type, the result of the concatenation is hashed again. To get each key, the hash needs to be aligned to length defined by *keyBits*. If the hash is too short, it is extended by adding `0x36` bytes to fit the size. If the hash is too long, the suffix is truncated.

With two keys, we can decrypt both *encryptedVerifier* and *encryptedVerifierHash* using the algorithm defined by *cipherAlgorithm* in the mode defined by *cipherChaining*. As an initialization vector, it is necessary to use the *salt* extended with `0x00` bytes to fit the *blockSize*. Same as in the standard encryption, the hash of the verifier is compared to the hash from the document. If the values match, the password is considered correct.

A.1.4 OpenDocument

OASIS Open Document Format for Office Applications, or simply *OpenDocument* format (ODF), is used by open-source office suites like OpenOffice[6] or LibreOffice[7] providing a free alternative to commercial solutions. The specification is publicly available and was released as an ISO [90] standard in 2006.

Like OOXML, OpenDocument files are represented by a ZIP container with multiple sub-files. All versions from 1.0 to 1.2 support encryption. If encrypted, the file contents are compressed by the DEFLATE algorithm [53] and encrypted using one of the encryption algorithms from Table A.4.

Version	Hash function	Encryption algorithm	Key length
ODF 1.0	SHA-1	Blowfish in 8-bit CFB	128-bit
ODF 1.1	SHA-1	Blowfish in 8-bit CFB	128-bit
ODF 1.2	SHA-256	AES/3DES/Blowfish	128/256-bit

Table A.4: Cryptographic algorithms used in different OpenDocument versions

Necessary metadata are present inside the *manifest.xml* file located in the `META-INF` directory. Those include the types of hash algorithm, encryption algorithm, and key derivation function. The *manifest:checksum* attribute specifies a base64-encoded digest for detecting the password correctness [36]. The password in UTF-8 form is hashed by SHA-1 or SHA-256 [75]. The result is then "strengthened" using PBKDF2 algorithm [98] with 1024 iterations of SHA-1 or SHA-256 [75] and the *salt*. The output is a 128 or 256-bit *encryption key*. Using the specified encryption algorithm and the initialization vector from manifest, the attacker can decrypt one of the document's subfiles. To verify the password, it is possible to simply recompute the checksum of the decrypted file and compare it to *manifest:checksum* value. If the values match, the password is considered correct [182].

[6] https://www.openoffice.org/
[7] https://libreoffice.org/

A.2 Archives

An *archive* is a container for files. The archive may be compressed to save disk space in exchange for time required for compression and decompression. For instance, bare UNIX `.tar` files are uncompressed. Applying of additional compression is usually denoted by the file extension. For instance, the file name of a tarball compressed by GZIP typically ends with `.tar.gz` or `.tgz`. In formats like RAR or ARJ, on the other hand, the archiving and compression are bound together. ZIP and 7z format use the compression by default, but is is possible to change the compression ratio to zero.

The choice of a compression tool depends on the user's needs. Tools like KGB Archiver[8] with the PAQ6 algorithm, offer exceptionally high compression rates, and can literally convert 1 GB files into 7 MB archives without any data loss. On the other hand, the decompression may take even hours. The most commonly used formats are ZIP, RAR, and 7z on MS Windows systems, TAR archives on UNIX-like systems.

For security reasons, the archives may be encrypted. While TAR archives do not natively support encryption, many formats like ZIP, RAR, or 7z do. The techniques for encryption and password verification differ between the formats, however, AES seems to be the most widely-used algorithm for encrypting file archives [50].

A.2.1 ZIP

The ZIP archive format was created by Phil Katz in 1989, and was first implemented in the PKZIP utility released by PKWARE, Inc[9]. The format standard is described by PKZIP's APPNOTE[10], and in 2015, it was standardized by the ISO [91]. ZIP is also a native format of a popular WinZIP tool[11], and is supported by many other compression tools. Despite the popularity of WinZIP, PKZIP development continued and in PKZIP 8.0, the professional edition was renamed to SecureZIP. ZIP format specification supports multiple compression algorithms: Store (no compression), UnShrinking, Expanding, Imploding, Tokenizing, Deflating, Enhanced Deflating, BZIP2, LZMA, WavPack a PPMd [163].
Figure A.1 shows the internal structure of a ZIP file. At the end of the file, there is the *central dictionary* containing a relative offset for each contained file, identifying its position from the beginning of the document. Each compressed file has its *local header*, containing its name, size, compression method, modification timestamp, CRC-32 checksum and other information.

ZIP format natively supports encryption, but only for file data. Metadata like file names are not encrypted. Therefore, users can see what files are inside the archive without entering the correct password. Early versions used the weak [29] PKZIP stream cipher [163]. The introduction of AES [50] in later versions was slightly controversial since WinZIP used a file structure that is different from the official PKWARE specification. In 2002, PKWARE introduced *Strong Encryption Specification* (SES) for the Professional Edition of PKZIP. However, before updating the APPNOTE, WinZip released its own AES-256 support with WinZIP 9.0 public beta in 2003 [45]. The WinZIP's AES-encrypted archive uses a different file format, than the SES supported by PKZIP and SecureZIP [163]. Currently, ZIP files can be encrypted by one of the following techniques:

[8] https://sourceforge.net/projects/kgbarchiver/
[9] https://www.pkware.com/
[10] https://support.pkware.com/display/PKZIP/APPNOTE
[11] http://www.winzip.com/

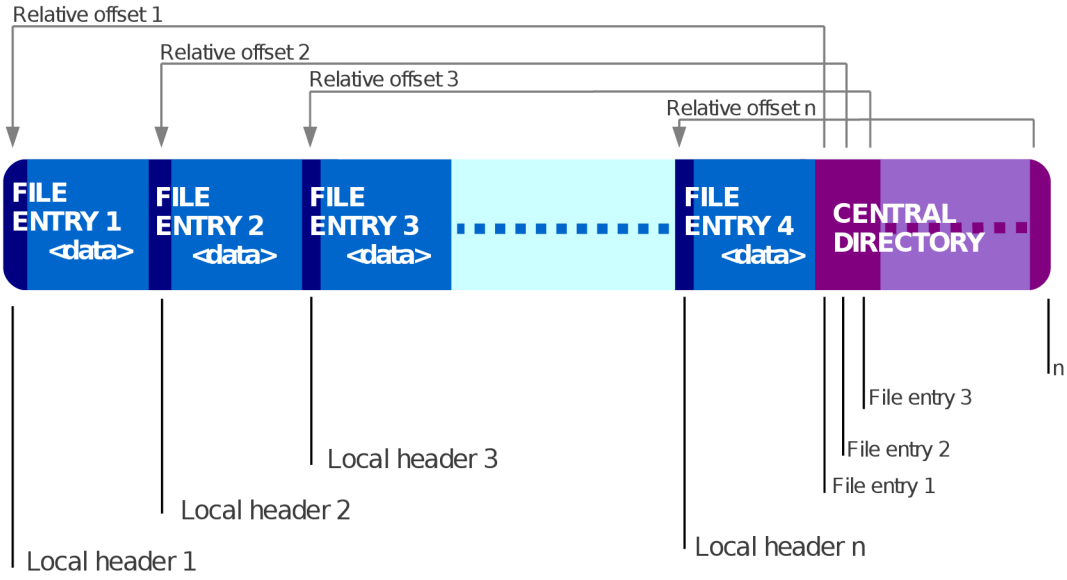


Figure A.1: The structure of a ZIP file (Source: WikiMedia Commons)

- **PKZIP stream cipher** - is the traditional PKWARE encryption used in the early versions of the ZIP format. Compared with today's cracking capabilities, the cipher is very weak and it is vulnerable to a known-plaintext attack described by Biham et al. [29]. The encryption uses PKWARE's own algorithm for initializing keys which are used for decryption of each file. Before the data itself, each file contains a 12-byte long *initialization vector* created by random values followed by two, or one (from ZIP 2.0) most significant bytes from the file's CRC. To verify the password, it is necessary to first perform the initialization phase that returns encryption keys and a 8-bit verification value. If the password is correct, then this value is identical to the lowest 8-bits of file's CRC. With only 8 bits to check, there is a high possibility of getting *false positive* passwords, i.e., situation when the the 8-bit value matches, but the password is not correct. Using the incremental brute-force attack, almost every 256th password is false positive. Thus, a good approach is to apply password-checking method to every file. The probability of *false positive* password occurrence is getting lower with the increasing number of files. For n files in the archive, the probability can be calculated as $\frac{1}{2^{8n}}$. Nevertheless, for a reliable password verification, it is necessary to check each "potentially-positive" password for correctness. To do so, one needs to decrypt a file, recompute its checksum, and compare it with the checksum in the file's *local header* shown in Figure A.1. Only if these values match, the password can be considered correct password [163]. This is an example of the *checksum-based password verification* described by Figure 2.4.

- **WinZIP's AES encryption** - uses AES in CTR mode [50]. The key length can be chosen from 128, 196, or 256 bits. For better integrity, instead of CRC, HMAC with SHA-1 is used. In the file's header, there is a *salt*, and a 16-bit *verification value*. The *encryption key* is calculated by using PBKDF2 [98] with 1,000 iterations of HMAC-SHA1 with *salt* from the header [75]. For a key of length n, one needs to generate $2n + 16$ bits using PBKDF2. First n bits represents the *encryption key*,

the second n bits is the key for HMAC function and the last 16 bits should match the *verification value* for a "potentially correct" password. The probability of getting a *false positive* password is $\frac{1}{2^{16}}$. For final verification, decryption and decompression are not necessary since the HMAC is calculated from the encrypted data [163]. To optimize the cracking, instead of computing PBKDF2 for all data blocks, one may use only the block with the 16-bit *verification value*. This can make the attack up to 4 times faster. The detailed structure of AES-encrypted ZIP files is described in a specification published by Corel Corporation [45].

- **Strong encryption specification (SES)** - is an AES-based technique standardized by PKWARE. The technique is used by newer versions of PKZIP, and by SecureZIP. Encryption-related metadata are located in the *Extra Fields* part of the *central directory* (see Figure A.1). Unlike in WinZIP's solution, the user can be authenticated not only by a password, but also by a certificate, or using both ways combined. The encryption metadata include the *initialization vector* (IV), the length of the key, the length of the data (Dlen), a piece of encrypted random data (Erd), encrypted *verification value* (EVV), and a CRC-32 checksum. Algorithm 12 shows how to use these values to verify a password [163].

Algorithm 12: Password verification in ZIP using Strong encryption (SES)

Input: *password, IV, Erd, Dlen, EVV*
Output: *true* if the password is correct, *false* if not

1 $K_1 = \text{PBKDF2}(\text{SHA-1}(password))$ `/* Getting the key to decrypt Erd */`
2 $rd = \text{AS-decrypt}(Erd, K_1, IV)$ `/* Decryption of random data */`
3 $K_2 = \text{PBKDF2}(\text{SHA1}(rd + IV))$ `/* Getting the key to decrypt EVV */`
4 $VV = \text{AES-decrypt}(EVV, K_2, IV)$ `/* Decryption of verification value */`
5 $CRC = VV[Dlen - 4]$ `/* Last two bytes are CRC */`
6 **return** $CRC \neq CRC\text{-}32(EVV, Dlen - 4)$

Since version 5.0, the ZIP format standard also officially supports DES, 3DES, RC2, and RC4 ciphers [163]. Altough its possible to manually use these algorithms, I found no free, or commercial ZIP-compression tool supporting them. Consulting this with PKWARE developers then confirmed my surmise.

A.2.2 7z

7z is an archive format used by an open-source 7-Zip[12] application, both created by Igor Pavlov. The format became popular thanks to a high compression ratio, and its open and modular structure. It supports multiple compression methods: LZMA (default), LZMA2, PPMD, BCJ, BCJ2, BZip2, and Deflate. In the philosophy of 7z, the encryption using AES-256 with SHA-256 is also represented as a coder. 7z archive using LZMA can provide up to 70% higher compression rate than ZIP format. 7z supports AES encryption with a 256-bit key generated by the SHA-256 hash function. By default, 7z does not encrypt file names, but this option is available in contrast to ZIP [156].

[12] http://www.7-zip.org/

Another difference from ZIP is slightly more complex and flexible structure of 7z files, making it more difficult to extract all metadata necessary to verify the password. Internally, the data of compressed files are saved as continuous streams. Multiple input files can also be concatenated and treated as a single stream. This techniques allows to achieve much higher compression ratio than compressing each file independently. The input data is processed using *coders* that represent pre-processing filters and compression algorithms. The following are supported: LZMA, LZMA2, PPMD, BCJ, BCJ2, BZip2, Deflate. To achieve even higher compression ratio, 7z can use multiple coders chained together [156].

A 7z archive resembles a database of *folders* and *streams* of data. A *folder* is a solid block of data that contains one or more encoded streams. Each stream may be encoded using different coders. A decoded stream may create multiple substreams. Each decoded substream corresponds to a single input file. The structure is described by the `7zFormat.txt` specification [155]. Each archive consist of four parts:

- **Start header** (32 bytes) - The header starts with a signature `7, z, 0xBC, 0xAF, 0x27, 0x1C` followed by general information about the archive. Those include 7z version, CRC of the start header, and link to the *end header*.
- **Compressed data of files** - This part contains streams of compresed (and possibly encrypted) file data.
- **Compressed metadata block for files** - The block contains links to compressed data, file names, sizes, information about compression methods, CRC of the files, timestamps and other metadata.
- **End header** - The end header has a variable number of items, each with a one-byte identifer. The allowed values range from *0x00* to *0x19*. In total, the header may contain up to 25 items. If the archive contains data, the header contains a link to the *compressed metadata block*.

The password verification is checksum-based (see Section 2.4) using CRC-32. For each folder, the archive contains the size and CRC-32 checksum of the decompressed data of the first file contained. In a nutshell, to verify a password, it is necessary to decrypt and decompress the file's data, recalculate the checksum and compare it to the stored one. If the values match, the password is correct.

First, it is necessary to locate the *end header* and get information about data streams from items identified by `0x03` and `0x04`. This information is represented by the `Streams Info` structure that includes `PackInfo`, `CodersInfo`, and `SubStreamsInfo`. The `PackInfo` structure provides information about folders: their amount, location, sizes, and CRC. The `CodersInfo` contains information about data coders that are used in the archive. Finally, the `SubStreamsInfo` provides information about substreams, i.e., data streams for individual files.

Once the structures are located, one needs to browse individual coders to search if there is any file encrypted with the AES-256 + SHA-256 method. Such a coder has identified by `0x06, 0xF1, 0x07, 0x01`. Moreover, it is necessary to find out what coders are used data pre-processing and compression. All these coders need to be supported by the password cracking tool.

Algorithm 13 shows the procedure when the LZMA compression is used. [156]. The encrypted data (encData) need to be extracted from the data stream of the file. The initialization vector (IV), the number of SHA-256 iterations (iterations) for deriving password,

Algorithm 13: Password verification in 7z archives compressed by LZMA

Input: *password, IV, iterations, encData, lzmaData, CRC*
Output: *true* if the password is correct, *false* if not

1 $password$ = UTF-16($password$) `/* Convert password to UTF-16 */`
2 K = derive_key($password, iterations$) `/* SHA-256 iterations */`
3 $compressedData$ = AES-decrypt(K, IV, $encData$) `/* data decryption */`
4 $data$ = decompress($compressedData$, $lzmaData$) `/* data decompression */`
5 **return** $CRC \neq$ *CRC-32(data)*

and the initialization data of LZMA dictionaries (lzmaData) are located in the corresponding coder structure. The CRC checksum is located in the file `PackInfo` structure for the corresponding folder.

A.2.3 RAR

RAR is an archive file format developed by Eugene Roshal for WinRAR archiving application. The name RAR stands for **R**oshal **Ar**chive. It supports compression, error recovery, and file spanning, i.e. splitting into multiple files. RAR is a proprietary format licensed by win.rar GmbH[13]. RAR allows encrypting not only the data of individual files but also their headers making it impossible to detect what files are present. Version 2.9 (WinRAR 3.0) introduced encryption of both data and headers using AES in CBC mode with 128-bit key. Starting from version 5.0, the key length was increased to 256 bits [175].

- **RAR 3.0** - From the main archive's header, we can get CRC checksum, header type (should be `0x73`) and 2 bytes of header flags. If the flags contain `0x04`, it indicates, the file's headers are also encrypted. On `0x400` address, we can find 8 bytes of *salt*. Processing a password and salt with SHA-1, we get the *initialization vector* for AES algorithm [50]. For password verification, we need to decrypt and decompress a file, recalculate it's CRC and compare it with the CRC from the header. If they match, the password is considered correct, if they do not match, the password may be wrong, however, it may also indicate a corrupted archive. Unfortunately, in RAR 3.0, it is impossible to distinguish between an incorrect password and a corrupted archive [6].

- **RAR 5.0** - can be identified by `0x52 0x61 0x72 0x21 0x1A 0x07 0x01 0x00` signature. The file consists of multiple blocks while each block has a header. The header contains CRC checksum, header size, header type, etc. If file headers are also encrypted, there is an extra header called the *encryption header* containing the common header values like checksum, size, and flags, but also a 16-byte *salt*, 8-byte *check value* for password verification, and a value called *KDF count* defining the number of PBKDF2 function for deriving an encryption key [98]. For each encrypted file, there is a special record resembling the *encryption header* which also contain an *initialization vector* for AES encryption algorithm [175, 50].

  After getting necessary metadata, using a password and salt we compute *KDF count* iterations of PBKDF2-HMAC-SHA-256 to get a 256-bit *encryption key* [98, 50]. By computing another 32 iterations, we get a 32-byte value called *pswcheck* on which we

[13] http://www.win-rar.com

apply XOR operations in the following way: byte no. 0 with byte no. 8, byte no. 16 with byte no. 24, etc. As a result, we get an 8-byte value which we compare with the *check value* extracted from the header. If they match, the password is considered correct [175].

A.3 Disk volumes

Not only files but entire filesystems can be encrypted to protect sensitive data. Such a filesystem may represent either a physical disk partition or a virtual one located inside a regular file. Encryption mechanisms could be provided natively by the operating system or by external tools. If the system partition is encrypted, it is impossible to boot the machine unless a correct password is entered.

A.3.1 TrueCrypt

TrueCrypt was a popular open-source software for creating encrypted disk partitions under Windows, Mac OS X, and Linux operating systems. The tool offered the following options for encrypted filesystems:

- **an encrypted file** with a nested virtual filesystem,
- **an encrypted partition** on an existing storage device, or
- **encryption of the entire storage device** supplied with a bootloader for *pre-boot authentication* in case of a system disk. [196, 218].

To make cryptoanalysis more difficult, TrueCrypt filled the empty with random bytes. TrueCrypt even supported creating of hidden partitions. The medium could contain two partitions, each unlockable by a different password. This created a suitable solution for potential extreme cases, where the medium's owner would be held hostage and threatened for surrender of the password. The owner could then provide a working password to one of the partitions, making the captors unaware of the existence of the other partition containing the true confidential content [196].

On 28 May 2014, for unknown reasons, the original project's website[14] announced the end of the project and recommended to use alternative solutions. It was stated, that the software may contain unfixed security issues. The recent audit[15] of TrueCrypt's source code found minor defects, however, no critical security issues or indications of the backdoor presence were detected. The true reasons for the project's closure is then a subject of speculations. Based on the TrueCrypt's source code, two new open-source projects were created: VeraCrypt (see Section A.3.2) and CipherShed (see Section A.3.3).

Independent of the method used, TrueCrypt allowed a user to choose from different encryption algorithms, hashing algorithms, and a working mode. TrueCrypt 7.1a supported the following encryption algorithms: AES [50], Serpent [28], Twofish [181], or their cascade combinations: AES-Twofish, AES-Twofish-Serpent, Serpent-AES, Serpent-Twofish-AES, and Twofish-Serpent. The cascade use possibly offered more security redeemed by longer encryption and decryption intervals. The cryptographic hashing algorithms supported were: RIPEMD-160 [54], SHA-512 [75], or Whirlpool [188]. Possible operation modes are LWR and XTS which was then set as default [218, 196].

[14] http://truecrypt.sourceforge.net/
[15] http://istruecryptauditedyet.com/

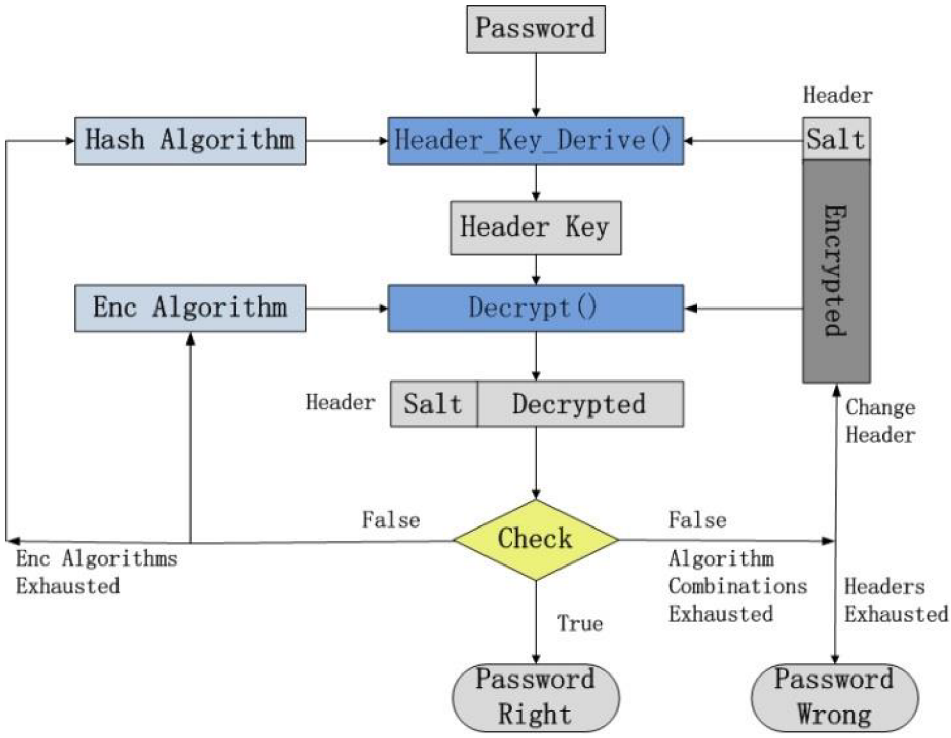


Figure A.2: TrueCrypt password verification (Source: Zhang et al. [218])

The password verification process is described by Figure A.2. Using a hash function, the *header key* is derived from a password and *salt* which is the first 64 bytes of the volume's header. Using the *header key* and encryption algorithm, the header is decrypted. The password is correct if Equation A.2 to Equation A.4 hold [218].

$$head[64..67] = \text{“}TRUE\text{”} \tag{A.2}$$

$$head[72..75] = CRC32(head[256..511]) \tag{A.3}$$

$$head[252..255] = CRC32(head[64..251]) \tag{A.4}$$

The number of hashing iterations depends on the function and partition type, specifically for the system partition for system partition, it is 1,000 iterations of PBKDF2-RIPEMD-160. For other partitions and standard file containers, TrueCrypt uses 2,000 for RIPEMD-160, and 1,000 for SHA-512 or Whirlpool [218, 196]. The main challenge for the attacker, however, is the fact, that the information about the encryption and hashing algorithms used are not saved anywhere. Since that, the attacker has to try $8 \times 3 = 24$ options, possibly multiplied by two, if LWR mode is also assumed as possible [218].

A.3.2 VeraCrypt

VeraCrypt[16] is a successor of TrueCrypt developed by IDRIX[17]. After the TrueCrypt's audit (see Section A.3.1) was finished, the developers of VeraCrypt claimed to have implemented security improvements based on the audit's results.

[16] https://veracrypt.codeplex.com
[17] https://www.idrix.fr/

Since VeraCrypt is a fork of TrueCrypt, the verification procedure is similar. However, the number of hashing iterations have been increased. For system partitions, it uses 327,661 iterations of PBKDF2-RIPEMD-160. For other partitions and containers, VeraCrypt uses 655,331 iterations of RIPEMD-160, and 500,000 iterations of SHA-2. The known-plaintext attack searches for work "VERA" instead of "TRUE." Also, Camellia and Kuznyechik ciphers were added[18] , and the only supported mode is XTS [54, 98].

A.3.3 CipherShed

Ciphershed is, or used to be, another fork of TrueCrypt. On February 1, 2016, the authors made a first official release. However, since December 28, 2016, the CipherShed website [19] appears to be down and may no longer be available.

A.3.4 BitLocker

BitLocker is a feature of Microsoft Windows first introduced in Windows Vista, and present in Pro and Enterprise versions of newer systems, also featuring in Windows Server 2008 and later. It enables *full disk encryption* (FDE) protecting entire disk volumes. BitLocker uses AES encryption in CBC or XTS mode using a 128-bit or 256-bit key. Unlike TrueCrypt or VeraCrypt which require having the disk partition formatted before encryption, BitLocker can perform the encryption process while still having the files on the disk [183].

The data is encrypted using the *Full Volume Encryption Key (FVEK)*, encrypted by *Volume Master Key (VMK)*. These two keys are located in an encrypted form in the volume metadata. To secure these keys, BitLocker uses mechanisms called key *protectors*[20]. The following key protector methods and combinations are available:

- **TPM only** – BitLocker uses the computer's Trusted platform module (TPM) to protect volume keys. The TPM builds the chain of trust during the boot sequence. TPM support in BIOS is required. The hardware of the TPM contains a *Storage Root Key* (SRK) tha is used for decrypting volume keys.
- **TPM and PIN** – BitLocker uses the combination of TPM and user-supplied PIN. The length of the PIN is 4 to 20 digits. If allowed, enhanced PINs with non-digit symbols can be used.
- **TPM, PIN, and startup key** – Requires TPM, PIN, and additional external key (startup key) from USB memory device.
- **TPM and startup key** – Requires TPM and the external startup key.
- **Startup key** – Only the startup key is required.
- **Password** – BitLocker uses a user password.
- **Recovery key** – BitLocker uses recovery key from a USB memory device.
- **Recovery password** – BitLocker uses a 48-digit recovery password.

[18] https://veracrypt.codeplex.com/wikipage?title=Encryption%20Algorithms
[19] https://www.ciphershed.org/
[20] https://docs.microsoft.com/en-us/powershell/module/bitlocker/add-bitlockerkeyprotector

- **Active Directory DS account** – BitLocker uses Domain authentication to unlock volume keys. Cannot be used with OS volumes.

The type of protector used defines what kinds of attacks are possible. Essentially, there are the following options:

- **Unlocking with the recovery key** – All protector options allow decrypting the volume if the recovery key is entered. The recovery key is created by Windows and automatically uploaded into the user's Microsoft account. If logged into the account, the user may request the key. In the case of domain-based authentication, the user may also sign into the Azure Active Directory account and recover the key.

- **Cold boot attack** – The key can be obtained at boot time. This is easiest with the TPM-only protector since no user interaction is necessary. If an additional protector (PIN, USB key) is used, the attack is not possible unless these values are known [144]. While TPM should prevent attackers from using bootkits – rootkits that tamper with the system boot files, Yi-ming et al. a security weakness in key management so that bootkits can make a break the booting protection [216].

- **Memory dump attack** – The encryption keys can also be obtained from a memory dump, as described by Al Shehhi et al. [183]. Afonin also describes that the RAM can be dumped using a FireWire/Thunderbolt attack [144].

- **User passsword attack** – If the volume is secured by password without TPM, it is possible to perform an offline attack [9]. First, it is necessary to extract the volume metadata. The password verification procedure is:

  1. Calculate SHA-256 twice on the password.
  2. Calculate 1,048,576 iterations of SHA-256. In each, the input is the last 32-byte hash concatenated with the hash from the first step, 16-byte salt, and iteration number. The final result is the intermediate key for AES-encrypting the *Initialization vector* (IV) derived from a nonce.
  3. Get the decrypted *Message Authentication Code* (MAC) as XOR of the encrypted IV and encrypted MAC.
  4. Get the decrypted *Volume Master Key* (VMK) as XOR of the encrypted IV and encrypted VMK.
  5. Calculate mac of the decrypted VMK and compare it to the decrypted MAC. If they mach, the password is correct.

- **Recovery password attack** – Technically, it is also possible to perform an attack on the recovery password. The recovery password is made of 48 digits made of eight 6-digit groups. Each group must be divisible by 11 and lower than 720,896. The sixth digit is checksum. Such rules redude the keyspace a bit but the number of candidate passwords is still enormous. Totally, there are $65,536^6$ possible combinations [9].

From the password cracking tools discussed in Section 2.4 the following can recover BitLocker passwords: hashcat, John the Ripper, Passware Password Recovery Toolkit, and Elcomsoft Forensic Disk Decryptor. There is also a tool called BitCracker[21] that is dedicated for recovering BitLocker password with attacks on the user password or on the recovery password [9].

[21] https://github.com/e-ago/bitcracker

A.3.5 PGP

Pretty Good Privacy (PGP) is a software developed by Symantec[22]. It was designed for signing and encrypting e-mails, files and disk partitions, following the OpenPGP standard [37]. Unlike Truecrypt, PGP and Bitlocker can perform encryption while keeping the files on the disk. Al Shehhi et al. also confirmed possibility of recovering the PGP password by using Elcomsoft Forensic Disk Decryptor [183].

A.3.6 Mac Disk Utility

Mac Disk Utility is a software developed by Apple which allows encryption of Mac OS X Extended disk partitions[23]. It uses AES with 128-bit or 256-bit key starting from MAC OS X v10.5. Al Shehhi et al. were successful with mounting the volume using Passware Password Recovery, however not with the recovery of the password [183].

A.3.7 FileVault

FileVault and FileVault 2 introduced in Mac OS X Version 10.7 (Lion) are another tools to encrypt Mac OS X disk partitions. It uses AES in XTS mode with a 128-bit or 256-bit key [50]. Choudary et al. analyzed the properties of FileVault encryption, and developed an open-source cross-platform library named `libfvde` which can decrypt and mount FileVault-encrypted volumes. To decryption and mount volumes one needs to possess a correct *authentication token*. The token is provided by the user and can be extracted from a memory dump [44].

A.4 Portable devices

With the expansion of portable devices like tablets, and cellphones, the necessity of achieving data confidentiality became crucial. Thus, portable OS developers like Google or Apple designed mechanisms for encrypting private data inside the device's memory.

A.4.1 Android

Android OS uses a *virtual machine* (VM) for running each user application in its own *sandbox*, i.e., separately from each other with restricted access to individual resources. This security-enhancing principle is similar to the Java Virtual Machine (JVM). Before Android 4.4 was introduced, the system used Dalvik VM. In Android 4.4, it is possible to choose between Dalvik VM, and Android runtime (ART) VM. Starting from version 5.0, the only VM supported is ART [32, 18]. The Android system also takes advantage from Security-Enhanced Linux (SELinux) which enables to define policies defining what operations on what resources each application can perform. SELinux was partially introduced in Android 4.3, and fully deployed in Android 5.0 [18]. For securing confidential data, Android OS basically supports two encryption schemes:

- **Full-disk encryption (FDE)**[24] - This scheme was introduced in Android 4.4, and further enhanced in Android 5.0. FDE enables the encryption of all user-created data

[22] https://www.symantec.com
[23] https://support.apple.com/en-us/HT201599
[24] https://source.android.com/security/encryption/full-disk.html

on a device. The data is automatically encrypted before committing to the disk and decrypted before returning to the calling process. Full-disk encryption is supported up to Android 9. Since Android 10, FDE is deprecated, and devices must use file-based encryption instead.

- **File-based encryption (FBE)**[25] - The FBE is supported by Android 7.0 and above. It enables different files to be encrypted with different keys that can be unlocked independently. Early versions were not able to use FBE together with adoptable storage (e.g. SD card). Since Android 9, devices can use both. For devices with Android 10 and higher, FBE is required.

FDE uses encryption scheme called DM-Crypt depicted by figure A.3. From a PIN or Passcode (password) and salt using a *key derivation function* (KDF), we receive a *derived key*. The salt value is generated randomly during the activation of the encryption process eliminating the attack using pre-calculated keys. This derived key is used to protect the the *File-system key* also called the *master key* which is on the top of the key hierarchy and protects the actual data.

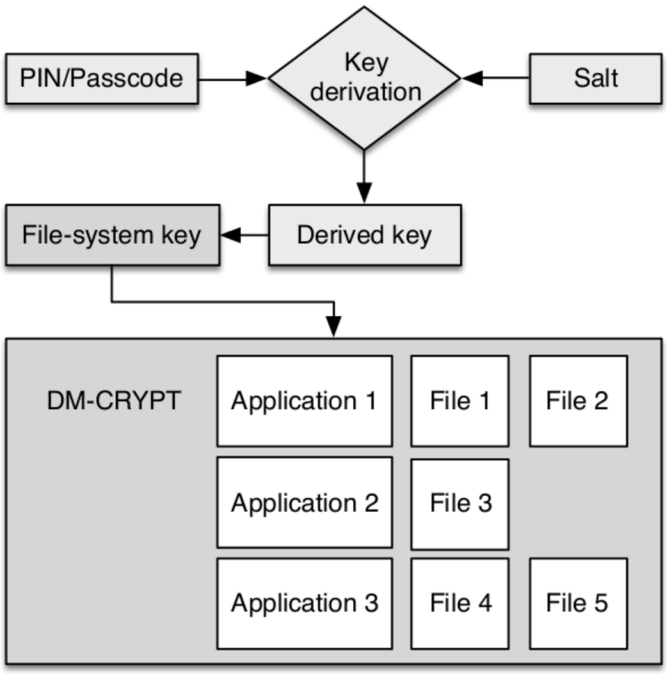


Figure A.3: DM-Crypt encryption system (Source: Teufl et al. [195])

Even before FDE scheme was introduced, Android OS supported encryption, starting from Android 3.0, however, the *key derivation function* (KDF) was different from newer versions. From Android 3.0 to Android 4.3, the KDF used standard hash-algorithms. The key was derived from a PIN or Passcode (password) and a random 16-byte salt using 2,000 iterations of PBKDF2 [98]. Teufl et al. performed various experiments with brute-force attack on the FDE. A 4-digit numerical key could be cracked almost instantly on the device itself. However, using off-device attack using GPUs, even a 10-digit key could have been

[25] https://source.android.com/security/encryption/file-based.html

cracked within a day. Alphanumeric or more complex passphrases having 6 characters or less were also crackable within a single day [195].
In Android 4.4, the PBKDF2 was replaced with the SCRYPT function. SCRYPT was specifically designed to prevent a GPU-accelerated attack by requiring a large (and configurable) amount of memory making highly-parallel cracking on GPU a serious problem[26]. Using a distributed CPU attack, PIN-codes 8-digit or shorter, or alphanumeric passphrases shorter than 5 characters were also crackable in less than a day [195].

Starting from Android 5.0 Lollipop, the system is using the full enforcement mode of SELinux [18]. Furthermore, the FDE relies on a hardware-bound key making the attack practically impossible. Using *online attack* has the advantage of having the hardware-bound key available to the operation system. However, there is a limited number of attempts, after which the device will be wiped and reset to the factory defaults[27]. Using *offline attack*, i.e. cracking on the different device, the hardware key is not available, making the only choice trying all possible combinations of a 128-bit AES key giving about 10^{38} possible solutions. Android 7.0 and FBE also relies on a hardware-bound-key, but it was practically exploitable on devices using Qualcomm chips from which it was possible to extract the hardware-bound key[28]. With different chips, the attack is, currently, impossible to be performed at an acceptable time.

A.4.2 Apple iOS

The iOS (previously known as iPhone OS) system developed by Apple Inc. is used as the operating system of the company's portable devices: iPhone, iPad, and iPod touch. The most recent version is currently iOS 10 released on September 13, 2016. To secure sensitive content, iOS utilizes a mechanism called *Data protection* that encrypts data on the device.

iOS 3 (iPhone OS 3.x)

Data protection was first introduced in iOS 3 and iPhone 3GS. The AES [50] encryption was applied on the entire filesystem using a single 128-bit *device key* called Key `0x835` and a random initialization vector. The key was based purely on the device UID "burned" at the chip, and could not be changed. The key was derived as follows:

```
key0x835 = AES-128(UID, "01010101010101010101010101010101")
```

Using the `0x835` key and a defined *initialization vector* (IV), the data was encrypted in the following way:

```
ciphertext = IV + AES-128(key835, data + SHA-1(data), IV)
```

The data was, however, still accessible transparently from a custom ramdisk. Moreover, it was possible to extract the key from the device for further offline use. Thus, no password cracking was necessary [23, 24].

iOS 4

The iOS 4 finally enabled users to choose their own passcode. Contents of almost all files are encrypted and each protected file has a unique key. The encryption scheme requires not

[26] http://nelenkov.blogspot.cz/2014/10/revisiting-android-disk-encryption.html
[27] http://theconversation.com/what-if-the-fbi-tried-to-crack-an-android-phone-we%2Dattacked-one-to-find-out-56556
[28] https://bits-please.blogspot.cz/2016/06/extracting-qualcomms-keymaster-keys.html

only Key `0x835`, but also another UID-based key called Key `0x89B` which can be derived as follows:

```
key0x89B = AES-128(UID, "183e99676bb03c546fa468f51c0cbd49")
```

Encrypted content is divided into 11 protection classes based on availability requirements, e.g., class 4 represents data available only when the device is unlocked, class 8 represents data which are always available, etc. For encryption of filesystem metadata and unprotected files (with no class specified), the "EMF key" derived from Key `0x89B` is used in a similar way to iOS 3 [23].

Each protection class has a *master key* that is stored in an encrypted form inside the *system keybag*. To decrypt the system keybag, one needs the device UID and the user-defined passcode. The process of unlocking the system keybag is illustrated in Figure A.4.

Using the master key, it is possible to access the per-file keys that protect the actual contents. Those are located in so-called keychains. A *keychain* is the storage of file keys that has the form of SQLite database with 4 tables containing not only keys, but also certificates, and other attributes. A keychain belonging to class $c \in \{1..11\}$ is protected by class c master key in the system keybag. If a *jailbreak* (i.e. getting root access) is performed on a device, one can use a graphical tool called Keychain Viewer to browse the keychain's content [23].

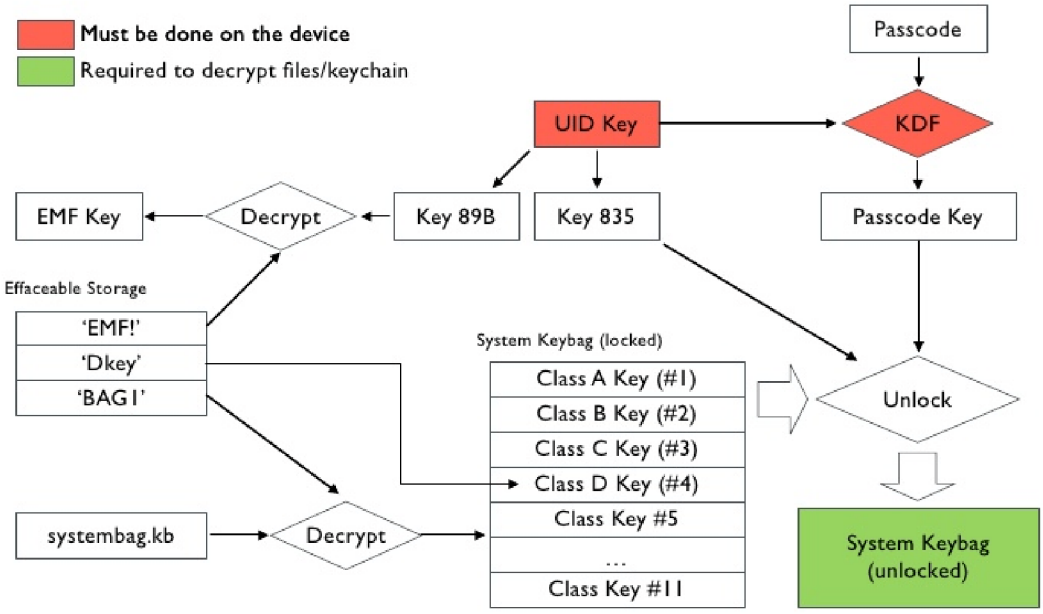


Figure A.4: iOS 4 key hierarchy (Source: Belenko et al. [24])

Moreover, iOS 4 offers an *Effaceable storage* with three 960-byte storage areas that can be quickly erased if necessary. Those include the payload key and initialization vector for decryption of the system keybag, or the filesystem encryption key "EMF!" that is used for obtaining the EMF key.

In addition to the key hierarchy, iOS 4 uses the *Escrow keybag* that allows iTunes to unlock the device. This keybag contains the same master keys as the system keybag. The Escrow keybag is protected by a 256-bit random passcode that is stored on the device. With each backup, iOS also generates a *Backup keybag* that contains keys for decryption of files and system keychains [25].

Belenko et al. from Elcomsoft experimented with the recovery of iOS passwords. The *online*, i.e. on-device, attack has shown to bee too slow, specifically 2.1 passwords per second on iPhone 3G and 7 passwords per second on iPad. As they suggested, the system keybag contains a hint on the *passcode* complexity, so it is possible to detect the type of the passcode, specifically: 4-digit PIN, digit-only passcode with different length, and any-length passcode possibly containing non-digit symbols. However, as Belenko et al. claimed, passcode is not "usually" a problem and it can be recovered by using a bootrom/iBoot exploit [24].

iOS 5 and newer

Newer versions of iOS brought further enhancements of the encryption system. Starting from iOS 5, AES mode was changes to GCM from the original CBC mode. Devices utilize a new "UID+" hardware key instead of UID. For protecting keychains, the system encrypts all attributes, not only the file keys. Starting from iOS the devices use new paritioning scheme called *Lightweight Volume Manager* (LwVM) that allows encryption of any partition. Filesystem encryption uses a different initialization vector that encryption of files [25]. Note, iOS 4 and iOS 5 had a security weakness[29] that allowed to bypass the screen locking protection and the device without knowing the passcode or encryption keys.

[29] https://www.technorms.com/7095/ios-5-shows-security-weaknesses

Appendix B

The contents of the attached storage medium

The attached DVD contains the following dictionaries and files:

- `experiments/` - various datasets for experiments related to the thesis,
- `fitcrack/` - the sources of the Fitcrack password cracking system,
- `pcfg/` - proof-of-concept tools related to grammar-based password cracking,
- `thesis-tex/` - LaTeX sources of this doctoral thesis with figures and bibliography,
- `works/` - all my published works,
- `cv.pdf` - curriculum vitae with an overview of my publications and outputs,
- `README.txt` - the description of the directory structure.
- `thesis.pdf` - this doctoral thesis in the PDF format,
- `summary.pdf` - the summary of the doctoral thesis.