The architecture of Fitcrack

distributed password cracking system

Technical report

Radek Hranický, Lukáš Zobal, Vojtěch Večeřa, Matúš Múčka

Technical report n. FIT-TR-2018-03

Faculty of Information Technology,
Brno University of Technology

Last modified: January 18, 2019

Table of Contents

The architecture of Fitcrack distributed password cracking system 3
Radek Hranický, Lukáš Zobal, Vojtěch Večeřa, Matúš Múčka
1 Introduction 3
1.1 Terminology 4
1.2 Structure of the document 5
2 Overview 6
2.1 Password cracking process 6
2.2 The cracking network 7
2.3 Task distribution 8
2.4 Adaptive scheduling 10
2.5 The architecture of client and server 12
3 Attack modes 14
3.1 Dictionary attack 14
3.2 Combination attack 17
3.3 Brute-force attack 18
3.4 Hybrid attacks 23
4 Server-side subsystems 25
4.1 Server directory structure 26
4.2 WebAdmin 26
4.3 WebAdmin frontend 27
4.4 WebAdmin backend 28
4.5 hashcat 29
4.6 Hashvalidator 29
4.7 maskprocessor 29
4.8 XtoHashcat 30
4.9 hcstat2gen 30
4.10 Generator 31
4.11 Validator 32
4.12 Assimilator 33
4.13 Trickler 33
4.14 Transitioner 33
4.15 Scheduler 33
4.16 Feeder 35
4.17 File deleter 35
5 Client-side subsystems 36
5.1 BOINC Client 36
5.2 BOINC Manager 37
5.3 Runner 37
5.4 hashcat 38
6 Client-server communication 40
6.1 Files transferred from server to client 40

6.2 Files transferred from client to server . . . 44
6.3 Trickle messages . . . 46
7 MySQL database . . . 47
7.1 The overview of BOINC tables . . . 47
7.2 The overview of Fitcrack tables . . . 48
7.3 fc_benchmark . . . 48
7.4 fc_charset . . . 48
7.5 fc_dictionary . . . 49
7.6 fc_hash . . . 49
7.7 fc_hcstats . . . 49
7.8 fc_host . . . 50
7.9 fc_host_activity . . . 50
7.10 fc_host_status . . . 50
7.11 fc_job . . . 51
7.12 fc_job_dictionary . . . 53
7.13 fc_job_graph . . . 53
7.14 fc_mask . . . 53
7.15 fc_masks_set . . . 53
7.16 fc_notification . . . 54
7.17 fc_protected_file . . . 54
7.18 fc_role . . . 54
7.19 fc_rule . . . 55
7.20 fc_settings . . . 55
7.21 fc_user . . . 56
7.22 fc_user_permissions . . . 56
7.23 fc_workunit . . . 56
8 Conclusion . . . 58
References . . . 58

The architecture of Fitcrack distributed password cracking system

Radek Hranický, Lukáš Zobal, Vojtěch Večeřa, Matúš Múčka

Brno University of Technology, email:
{ihranicky,izobal}@fit.vutbr.cz, {xvecer18,xmucka03}@stud.fit.vutbr.cz

Abstract. This technical report describes the architecture of Fitcrack distributed password cracking system developed within *Integrated platform for analysis of digital data from security incidents* project. Fitcrack serves as an open-source solution for recovering plaintext passwords from various cryptographic hashes, as well as a platform for research and development of new password cracking methodologies. The report documents both server and client sides of the system, provides detailed description of all subsystems and their interfaces, and clarifies the protocols used for communication between the server and clients.

1 Introduction

Fitcrack was initially created as a proof-of-concept tool for demonstrating the feasibility of using *Berkeley Open Infrastructure for Network Computing* (BOINC)[1] [2] as a task distribution platform for password cracking. The goal was to create an efficient, flexible, and scalable GPU-accelerated solution which is not limited to specific hardware and number of nodes. BOINC was initially designed as a public-resource computing solution, however, in our previous research, we have shown its applicability in password cracking even in private distributed networks [5]. In our use-case, BOINC handles the authentication of computing nodes, provides the distribution and automatic updates of executable binaries, OpenCL[2] kernels implementing the cryptographic algoritms for GPUs, and the input/output data of each cracking task [5].

The original version used a custom OpenCL-based software solution for computing hashes on the client side. The *Fitcrack client* was a C++ application capable of cracking password hashes from the following encrypted formats: PKZIP, WinZIP, SecureZIP, 7z, RAR versions 3 and 5, PDF up to version 1.7 Extension Level 3, and MS Office documents up to Office 2016. The algorithms for cracking procedures were implemented in three variants: i. a CPU-only implementation; ii. an OpenCL implementation for all formats; and iii. a CUDA[3] implementation

[1] https://boinc.berkeley.edu/
[2] https://www.khronos.org/opencl/
[3] https://developer.nvidia.com/cuda-zone

for all formats except 7z and RAR. Some of the GPU kernels were adopted from our single-machine tool *Wrathion* [6].

To achieve higher cracking speeds and get support for more hash formats, we replaced the original *Fitcrack client* with *hashcat*[4], a self-proclaimed "World's fastest password cracker". Considering speed, team hashcat won 5 of 7 years of *Crack me if you can* (CMIYC[5]) contest. Assessing features, hashcat supports over 200 different hash formats, and several different attack types: brute-force attack, dictionary attack, combinator attack and hybrid attacks; moreover, it supports the use of password-mangling rules including the ones used by popular *John the Ripper*[6] tool. All cracking algorithms are implemented using OpenCL which allows computing all OpenCL-compatible CPUs, GPUs, FPGAs, and DSPs.

This report documents the new hashcat-based version of Fitcrack, its architcture, the distribution of cracking tasks, and the implementation of various hashcat-compatible attack types.

1.1 Terminology

The document uses various terms which will be described in the following sections. Some of them may have different names in other cracking solutions. The most important are:

- **Fitcrack** - a distributed hash cracking software developed by Fitcrack team.
- **BOINC** - a framework for distributed computing used in Fitcrack.
- **hashcat** - world's fastest password cracker used for hash cracking in Fitcrack.
- **Attack mode** (or attack type) - the type of an attack signifying how the candidate passwords are obtained. Fitcrack supports the same attack types as hashcat:
  - *Dictionary* attack - taking passwords from a text file,
  - *Combination* attack - combining two dictionaries,
  - *Brute-force* attack - the exhaustive search,
  - *Hybrid* attacks - combine the previous types.
- **Job** - a single cracking task defined by a name, attack type, attack options and one or more hashes to be cracked.
- **Workunit** - a single piece of cracking work assigned to a host. It is a chunk created from *keyspace* by defining the range od *password indexes*.
- **Host** (client, cracking node) - computer used for the cracking.
- **Targeting** - a technique of creating concrete workunits for specific nodes only.
- **Password** - a sequence of characters serving as the plaintext input of the hash function.
- **Hash** - an output of the cryptographic hash function. The input for cracking.
- **Hash type** - a unique number[7] representing the format of a hash.

[4] https://hashcat.net/
[5] https://contest.korelogic.com/
[6] http://www.openwall.com/john/
[7] https://hashcat.net/wiki/doku.php?id=example_hashes

- **Input hash** - a hash that serves as the input of a cracking task. The goal is to find the plaintext string from which the hash was computed.
- **Correct password** - a password that we search for in a cracking task; the hash of the correct password is the input hash.
- **Candidate password** - a password which we test for correctness.
- **Candidate hash** - a cryprographic hash of the candidate password. The
- **Keyspace** - the number of candidate passwords implicating the complexity of a job. Higher keyspace means the job is more complex.
- **Password index** - a number within the keyspace representing a concrete candidate password. In brute-force attack, each workunit is defined by a range of password indexes signifying where to start and where to stop.
- **Dictionary** - a text file containing a password on each line.
- **Password-mangling rule** - a rule for modifying candidate password by replacing, inserting, or deleting characters. The rules were introduced within Jogn the ripper tool, and adopted to hashcat.
- **Character set** (charset) - a set of characters used for generating password candidates. For hahscat, charset files have '.hcchr' extension.
- **Mask** - a sequence of characters defining how candidate passwords may look like. Mask are used in *brute-force* attack and *hybrid* attacks.
- **Markov chain** - a stochastic mathematical model used for generating candidate passwords within a brute-force attack. Its states are represented by probability matrixes stored within a '.hcstat2' file.
- **User** - A person having an account to access the Fitcrack webadmin.

1.2 Structure of the document

The technical report is structured as follows. Section 2 provides the overview the password cracking process and the principles of work distribution used in Fitcrack. It also includes the basic scheme of a distributed network and defines two main participants: the server and clients. The subsystems implemented on the server-side are described in section 4, while section 5 aims at the client-side. The protocols used for communication between the two sides are described in section 6. Section 7 describes the schema of the SQL database used on the server to store all job-related information. Section 8 concludes the document.

2 Overview

This section describes the basic principles of password cracking followed by the principles of task distribution used in Fitcrack. Least but not last, it describes the architecture of a generic cracking network.

2.1 Password cracking process

The password cracking is based on systematic selection of *candidate passwords* (passwords we want to try), while each selected candidate password is verified for correctness. Eventually, the process ends with a correct password found, or with an exhausted set of assumed passwords, i.e. no password found. An algorithm or tool selecting the passwords could be called a *password generator*. Different attack types (see section 3) use different types of generators. Depending on the assignment, we have two types of cracking tasks:

- cracking a raw hash,
- cracking an encrypted medium.

Raw hashes are used for various purposes which include storing user passwords in web services, operating systems, and other software. Cracking a raw hash is quite straightforward. We continuously generate candidate passwords and from each password, we calculate a cryptographic hash called *candidate hash* and compared it with the hash we want to crack. Please note, that it is necessary to know the hash function used. The complexity of a task depends on the number of candidate passwords, as well as on the cryptographic function used. The speed of cracking may differ notably between various existing algorithms. For example, using hashcat and NVIDIA GTX 1080Ti GPU, the cracking speed[8] (in hashes per second: H/s) of MD5 [14] is about 31 GH/s, however the cracking speed of Bcrypt [12] with 4 rounds is 20 kH/s, which is more than 1,000,000 times slower.

Encrypted media include documents (Office, PDF, etc.), archives (ZIP, 7z, RAR, etc.), and other containers including disk partitions encrypted by VeraCrypt[9] or other software. The recovery process itself depends on the encrypted media type, concrete format and algorithms defined by the format's manufacturer. For most documents and archives only metadata is needed to verify the password. For example, encrypted PDF documents store the hash of the (modified) password we are looking for. The hash is called a *verification value* [1].

This is the simplest case and is depicted in figure 1. From each generated password, we need to compute one or more specific hash functions. Many formats like Office Open XML use thousands of hashing algorithm iterations [17]. The number of iterations is chosen to be high enough to make a possible attack more difficult, but low enough to prevent delays of a regular content viewing a with known password. The resulting hash is then compared with the *verification value*.

[8] https://gist.github.com/epixoip/973da7352f4cc005746c627527e4d073

[9] https://www.veracrypt.fr/

If they match, the password is considered correct. If not, another password is tried.

In some cases, the hashing block has additional input called *salt*, which is usually a random value located inside the document, and is a part of encryption metadata denoted above. The simplest way of use is to concatenate the salt with the password. The purpose of the salt is to make the attack harder and resistant to the use of *rainbow tables* [15]. Before comparison with the verification value, for some formats, the resulting hash is mixed with another value, often called *pepper*. The purpose is again, to increase the difficulty of an attack.

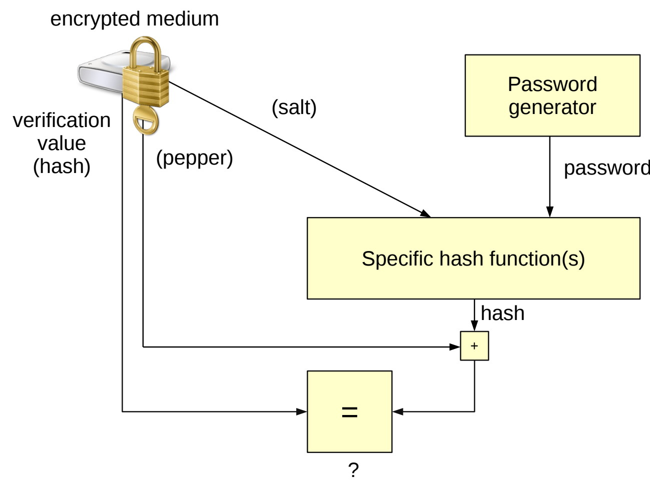

Fig. 1. Password cracking of encrypted media

2.2 The cracking network

The architecture of a distributed network consists of a project *server* and multiple *clients*. A client may use one or more OpenCL devices. Each device may be of a different type (CPU, GPU, FPGA, DSP), manufacturer (Intel, AMD, NVIDIA, etc.), and model (e.g. NVIDIA GTX 1080 Ti vs. RTX 2080 Ti). An example of such network is shown in figure 2.

If all nodes are equal, we say the network is *homogenous*; if they differ, teh network is *heterogenous*. If there are nodes of different OpenCL-device types, e.g. both GPU-equipped, and CPU-only nodes, we call this environment a *hybrid* network [7].

In Fitcrack, the actual computation of cryptographic hashes (as mentioned in section 2.1) is performed by the clients only. The server figures as a controller of the cracking process. The main objective of the server is to distribute work.

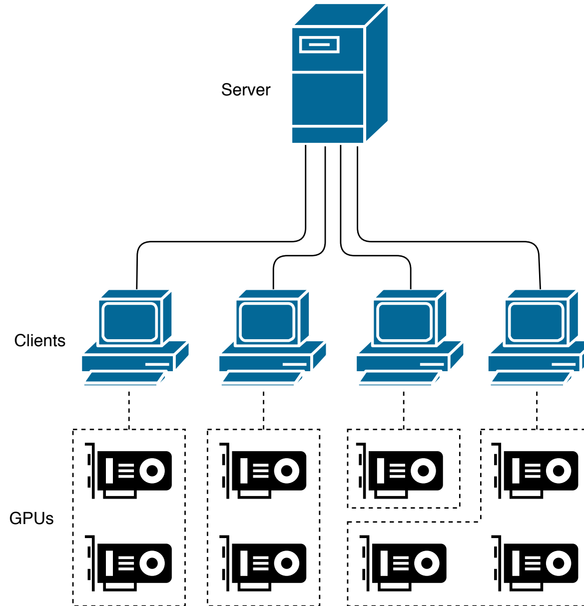

Fig. 2. An example of a cracking network

2.3 Task distribution

In our terminology, a *job* represents a single cracking task added by the *administrator*. Each job is defined by an attack type (see section 3), attack settings (e.g. which dictionary should be used), and one or more password hashes of the same type (e.g. SHA-1). There are three basic approaches how to distribute a job over multiple nodes:

- **Hash distribution** described by Pippin et. al. [11] uses the same candidate passwords on all nodes, however each node is cracking a different hash. Since hashcat is capable of cracking multiple hashes for each candidate password while the candidate hash is only generated once, we assume this approach ineffective.
- **Static chunk distribution** introduced by Lim et. al [8] 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.
- **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 (see sec-

tion 2.4). Moreover, losing the result of a workunit has lower impact due to its size.

The total number of candidate passwords within a cracking task is called *keyspace*. Let us assume that every candidate password p is a string over Σ alphabet, thus $p \in \Sigma^*$. The set of all candidate passwords is $P \subset \Sigma^*$, and $|P|$ is the *keyspace* of the job. The cardinality and elements of P depend on the type of attack. For the purpose of task distribution, let us assume that P is always a finite ordered set.

Based on the definitions above, we 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*.

Let us consider a simple incremental *incremental brute-force attack* (also known as exhaustive search) [5], where we want to generate all password of lenghts between 1 and 3 over alphabet $\Sigma = \{a, b, c, \ldots, z\}$. Then:

$$
\begin{aligned}
g(0) = a,\ \ldots,\ g(25) = z \\
g(26) = aa,\ \ldots,\ g(701) = zz \\
g(702) = aaa,\ \ldots,\ g(18277) = zzz.
\end{aligned}
\tag{1}
$$

Each *workunit* in Fitcrack is defined by the range of indexes: i_{min} a i_{max} while

$$
0 \leq i_{min} \leq i_{max} \leq (|P| - 1). \tag{2}
$$

The actual work lies in trying ever possible passwords given by generator $g(i)$ 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 *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 4.10) which specifies the range of indexes for each workunit. The size of the workunit is calculated using the *adaptive scheduling algorithm* described in section 2.4.

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 $i_{max} - i_{min}$.

While for dictionary attack without the use of *password-mangling rules* (see 3.1), 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.**

For example, assume a brute-force attack using mask (see section 3.3) `?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 7), 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.

2.4 Adaptive scheduling

A process called *targeting* defines which workunit is assigned to which host. BOINC supports two types of workunits based on the targeting:

- **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 dynamic heterogeneous environment working nodes 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. Our goal is to propose such distribution strategy that will maximize effectivity of working clients. It means that the higher-performance clients would receive a larger workunit than the lower-performance clients.

Solving such situation requires the use of *targeted* workunits. Our approach of adaptive time calculation estimates how much time it would take to verify the remaining candidate passwords on all the active clients. Based on this time, each active node will be assigned an appropriate size of the keyspace P. The size depends on the node performance (speed). More formally, let t_p be a process time (in sec) describing how much time would remaining verification take, s_i be the number of passwords (size) assigned to node i, and v_i be the current speed of node i in passwords per second.

Based on the speed, node i will be assigned a subset of keyspace P for verification, i.e., $s_i = t_p \cdot v_i$. Speed v_i is determined based on the previously solved task, i.e., $v_i = \frac{s_{prev}}{t_{prev}}$.

The problem is how to choose v_i for a newly connected client. One solution is to run a *benchmark* on the client to calculate its performance.

Estimation of remaining process time t_p cannot be determined by the node performance only. Too low or too high value can make the computing less effective:

- Lower t_p is more suitable for unstable environment where clients frequently disconnect or change their performance. Thus, the impact of a lost task 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 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 to be assigned to them.

For efficient task distribution, we define function $proctime(t_J, s_R, k)$ that adaptively computes expected process time t_p till the end of the keyspace processing. Parameter t_J is an elapsed time of the computing, s_R is a number of remaining passwords to be verified and k is a number of active nodes 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 $s_i = v_i.t_p$. Thus, the remaining keyspace will be distributed among working nodes according to their performance. In optimal case, all nodes complete their tasks in t_p as estimated.

Lines 2 to 7 of the algorithm compute the entire speed of all active nodes. Line 8 is a bit tricky. Normally, we would calculated t_p as $t_p = \frac{s_R}{v_{sum}}$. Here, we add parameter α called *distribution coefficient* that ranges from 0 to 1. In other words we say that not the entire remaining keyspace will be assigned, only its fraction. E.g., for $\alpha = 0.1$, 10% of the keyspace P is assigned among currently active nodes. Why not the entire keyspace is assigned? The answer lies in dynamic behavior of working nodes. In case that additional nodes connect to the network, there would be no task for them and distributed solution would become less effective. Thus, a part of the keyspace is put aside hoping that more nodes will participate on the computing in the future. If not, the rest of the keyspace will be distributed among current nodes according the above mentioned algorithm.

Value of t_p is 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 our experiments, we recommend t_{pmin} to be at least 1 minute and t_{pmax} to be about 60 minutes. When creating a new job in Fitcrack WebAdmin (see section 4.3), the administator can specify t_p as the *seconds per workunit* option.

Algorithm 1: Adaptive calculation of t_p

Input: t_J, s_R, k
Output: t_p

```
1:  v_sum = 0
2:  forall client_i ∈ {0, ..., k} do
3:      if client_i is active then
4:          v_i = s_prev / t_prev
5:          v_sum = v_sum + v_i
6:  t_p = s_R / v_sum · α
7:  if t_p < t_pmin then
8:      t_p = t_pmin ;                         // minimal task time
9:  else if t_J > t_pmax then
10:     t_p = min(t_p, t_pmax) ;                // maximal time
11: else
12:     t_p = min(t_p, t_J) ;                   // smaller tasks
13: return t_p
```

2.5 The architecture of client and server

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. While the server is responsible for management of cracking jobs, clients serve as "workers" who run the cracking process itself. Clients communicates with the server using an RPC-based *BOINC scheduling server protocol*[10] over HTTP(S). The current architecture of Fitcrack is shown in figure 3, and is fairy different from the original one described in [5]. Each side consists of multiple subsystems which will be defined in the following sections.

[10] https://boinc.berkeley.edu/trac/wiki/RpcProtocol

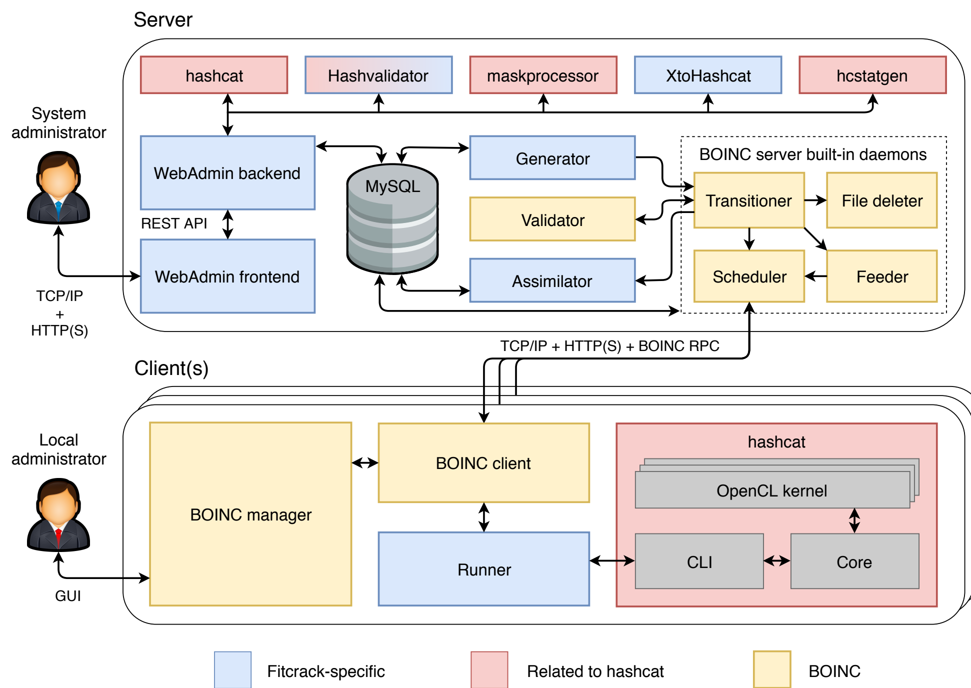

Fig. 3. The architecture of Fitcrack

3 Attack modes

As a cracking engine, Fitcrack uses *hashcat* version 4.2.1. The attack mode of hashcat is selected by by a number passed with the `-a` parameter. The allowed attack modes are: *dictionary* (straight) attack (0), *combination* attack (1), *brute-force* (mask) attack (3), and *hybrid* attacks (6 and 7). In this section, we show how we perform these attacks in the distributed environment of BOINC. In Fitcrack, we support all hashcat's attack modes, however, based on the attack configuration, we represent them internally by two numbers:

- **attack_mode** - corresponding to hashcat's attack mode,
- **attack_submode** - further specifying the attack.

As we discuss in section 3.4, we transform hashcat's hybrid attacks to a *combination attack*, therefore for the five *attack modes* of hashcat, Fitcrack only three. The numbering of modes and submodes is described by table 9.

mode	submode	description
0	0	Basic dictionary attack
0	1	Dictionary attack with *password-mangling rules*
1	0	Basic combination attack
1	1	Combination attack with *left rule*
1	2	Combination attack with *right rule*
1	3	Combination attack with *left* and *right rule*
3	0	Basic brute-force attack
3	1	Brute-force attack with custom hcstat file using 2D Markov
3	2	Brute-force attack with custom hcstat file using 3D Markov

Table 1. Attack modes and submodes in Fitcrack

For simplicity, we can merge the *mode* and *submode* together and define a unique two-digit *attack number*, e.g. `13` stands for a combination attack with both rules, `32` stands for a brute-force attack with user-defined 3D Markov model, etc.

Since we consider users to be familiar with hashcat attack modes, the front-end of Fitcrack *WebAdmin* (see section 4.3) provides an abstraction of Fitcrack's attack modes and thus the user controls the Fitcrack like hashcat.

The time and space complexity of the attacks is directly proportional to the *keyspace* $p = |P|$, i.e. the number of all password candidates defined in section 2.3. A formula for the calculation of p will be shown for each attack mode.

3.1 Dictionary attack

A *dictionary attack*, also referred to as a *wordlist attack* or *straight attack*, uses a text file called *password dictionary.* The dictionary contains password candidates, each placed on a separate line. Hashcat successively reads the password

candidates, calculates their hashes, and compares the results with the input hashes, i.e. those we are trying to crack, as described in section 2.1.

Such dictionaries may contain words from a native language, or real passwords obtained from various web service security leaks[11]. One of the most well-known leaked dictionary is *rockyou.txt* containing over 15 milion passwords. The dataset origins to the end of 2009 when user account information from RockYou portal leaked due to an attack[12].

Fitcrack supports the use of one, or multiple password dictionaries. From the mathematical perspective, we can consider each dictionary as an ordered set D, where the order is defined by the arrangement of passwords in the dictionary. For n password dictionaries, the keyspace p can be calculated as the sum of their cardinalities:

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

where D_i is the i-th used dictionary.

3.1.1 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 include over 70[13] different rules. Few examples of their practical use are illustrated in table 2.

Rule	Description	Input	Output
l	Converts A–Z to lowercase	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
]	Deletes the last character	p@SSw0rd	p@SSw0r
k	Swaps last two characters	p@SSw0rd	p@SSw0dr

Table 2. An example of password-mangling rules

To use password-mangling rules, the user has to define a file called *ruleset* which contains one or more rules on each line. The rules are applied to all candidate passwords in the following way: the first candidate password is modified by rules on the first line of the ruleset; the result is used. Then, the rules on the second line of the ruleset are applied to the original password; the result is used. Eventually,

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

the entire ruleset is processed. The same password-mangling principle is applied to the second candidate password, third candidate password, until we eventually reach the end of the password dictionary.

The rules enhance the repertoire of passwords, however, increase the total keyspace of the job. This is because Fitcrack applies every rule from the rule file to each dictionary password. The total keyspace is calculated as the sum of dictionary keyspaces multiplied by the number of rules in the rule file:

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

where r is the number of lines in the ruleset.

3.1.2 Distributed dictionary attack In distributed cracking, it is necessary to distribute the password candidates from the server to clients, i.e. the computing nodes. This effort has significant overhead, and for less-complex hash algorithms could lead to an inefficient distributed attack [7].

While it would be possible to send the whole dictionary to all hosts together with indexes, we chose another approach. The reason is the candidate lists might be very large and sending the whole file would increase the cracking time largely, as each host needs only a portion of the original list.

Therefore, a fragment of the original dictionary is created for each host with each workunit, which size depends on the host's current computing power. What's more, this number can vary in time, reflecting each hosts' performance changes. You can see a simplified scheme of this attack in figure 4.

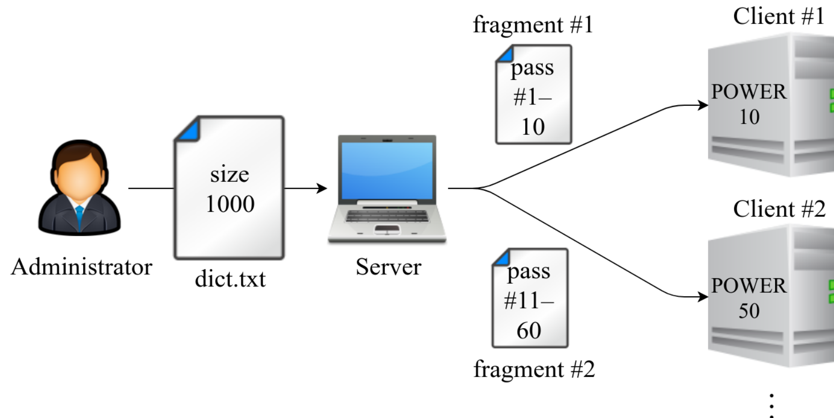

Fig. 4. Example of dictionary attack distribution

3.2 Combination attack

The *combination attack*, also referred to as *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 5.

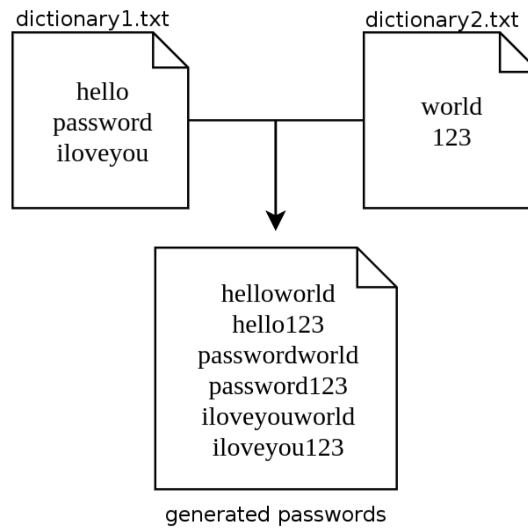

Fig. 5. Illustration of combination attack

Let D_1 be the left dictionary, and D_2 the right dictionary. Keyspace p can be calculated as:

$$p = |D_1| * |D_2|$$

When dealing with hashcat, we realized its keyspace computation doesn't consider the second dictionary. When the hashcat is supposed to verify one password in combinator attack, it, in fact, verifies $1 \times n$ passwords. With possibly huge dictionaries, the workunit size would be uncontrollable.

A simple solution to this problem would require generating all possible combinations to a single dictionary, proceeding with a dictionary attack, described above. This would, however, increase the space complexity in the sense of the transmitted passwords from linear, ideally $m+n$ passwords, to polynomial, $m \times n$, rapidly increasing the time needed to transfer data to all computing nodes.

To deal with this issue, we came up with the following solution. The first dictionary is distributed as a whole to all computing nodes in the first workunit,

also referred as a chunk. Then, with each workunit, only a small portion of the second dictionary is sent. This way, we can control the number of passwords in the second dictionary – n, while we can still limit the number of verified passwords in the first dictionary – m, using the hashcat mechanism. Also, we keep the linear complexity of the whole attack. You can see a scheme of such an attack in figure 6.

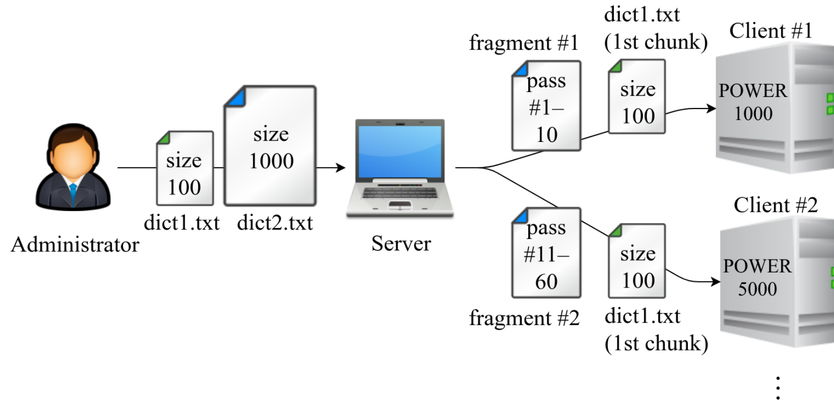

Fig. 6. Example of combinator attack distribution

3.3 Brute-force attack

The *brute-force* attack is an exhaustive search for correct password(s) trying every possible password candidate. In *hashcat*, the attack is based on *password masks*. The mask is a pattern defining the allowed form of candidate passwords - i.e. how candidate passwords "may look like". The user may define one or more mask for an attack. The cracking process then consist of generating **every possible sequence of characters** upon each mask.

3.3.1 Password mask A *password mask* is a template defining allowed characters for each position of the password candidates. The mask has the form of a string 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 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 has 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 on position i, or
- **a substitute symbol** (S_i) **for a character set** (C_i) - which defines the allowed values of c_i - i.e. possible characters on position i in the generated candidate passwords.

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

Table 3. 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 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 7. If there are concrete characters in a mask, the same characters at the same position 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 candidate passwords. For substitute symbols, all possible characters from corresponding character sets are 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 brute-force attack, the number of possible candidate passwords can be calculated as:

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

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`:

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

we have 2600 possible password candidates.

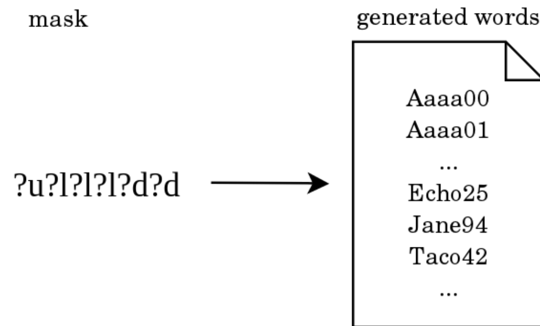

Fig. 7. Illustration of a brute-force mask attack

3.3.2 Markov chains In hashcat, the candidate passwords are **not** generated by the lexicographical order of the character sets. Instead, an algorithm based on *Markov chain* [10, 4] mathematical model, is used. The entire idea behind Markov chains is to use knowledge obtained by learning on existing wordlists to **generate more probable passwords first**. The difference between the two approaches is illustrated in figure 8 which shows examples of generated candidate passwords.

Markov model uses a matrix with character order statistics, saved inside a *.hcstat* file. Starting from hashcat 4.0.0, hashcat uses[14] LZMA[15] compression, and the extension changed from *.hcstat* to *.hcstat2*. The default file used for brute-force attack is `hashcat.hcstat`, respectively `hashcat.hcstat2`. However, `--markov-hcstat` option allows the user to specify a custom file.

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

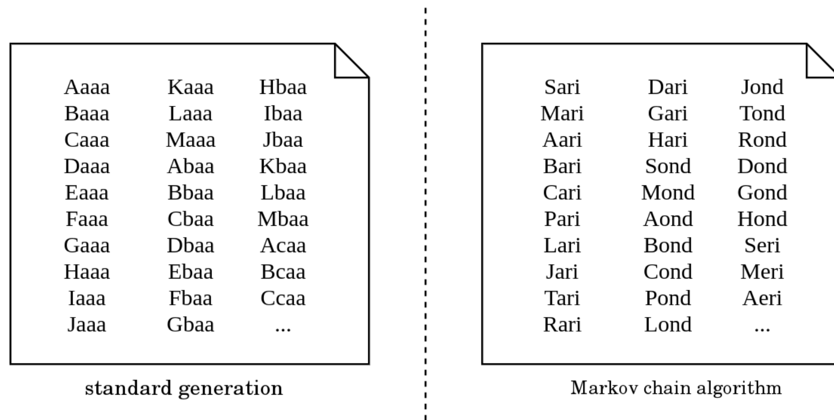

Fig. 8. Candidate password order using Markov chains

An example of the Markov chain matrix is shown in figure 9. In each row, the matrix shows different characters from the character set in order from the most probable, to less probable. The first row entitled with ε shows the most probable 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 on is `e`, etc.

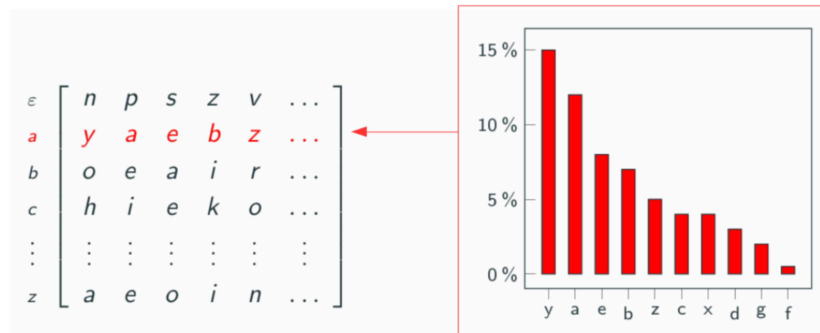

Fig. 9. Markov chain probability matrix

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 9, 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 3.3.1. 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 [4].

For now, let us ignore the keyspace optimization used by hashcat, described in section 2.3 – i.e. assume the keyspace is the actual number of password candidates. Figure 10 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.

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

Fig. 10. Example of Markov matrix with threshold set to 3

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 support two different models:

- **2D Markov model** (*classic*) - uses 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*) - is used by default in 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 matrixes, one per each password position. If the first character is generated, the first matrix is used, for second chracter, second matrix is used, etc.

3.3.3 Distributed brute-force attack

One of the biggest challenges of distributing the mask attack in hashcat was the way hashcat computes the keyspace of each mask. This number depends on many factors, which in result doesn't inform you about the real keyspace at all. However, the real keyspace is needed to compute the size of each workunit, depending on each host's current performance measured in hashes per seconds.

To overcome this obstacle, the real keyspace is computed from the mask before the attack starts, using our own algorithm. Comparing this number with hascat keyspace, we can determine how many real passwords are represented by a single hashcat index. With this knowledge, sending the mask with corresponding index range to verify is no longer a problem.

For each workunit, the only information we need to distribute is the mask with new index range. This makes a mask attack, in contrast with previously described attacks, very efficient in a distributed environment.

3.4 Hybrid attacks

Hybrid attacks combine the dictionary attack (see section 3.1) with brute-force attack (see section 3.3). There are two variations of hybrid attack supported by hashcat. 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 11.

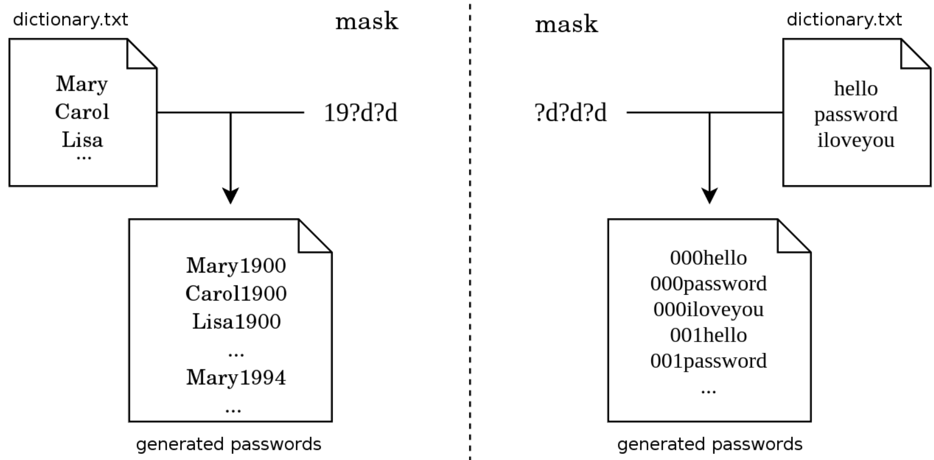

Fig. 11. The principle of hybrid attacks

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 tech-

nique. 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 symbol S_i. 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. Similar to the combinator attack, hashcat does not provide us with the keyspace of the whole attack but with the size of the dictionary only. This means, when instructed to verify one password, in fact, hashcat checks one dictionary password combined with the whole mask.

The same solution as in combination attack cannot be used, as there is no way to send just a portion of the mask to each host. To avoid generating all possible variants beforehand, which would cause the same problems described in the combinator attack above, we use the following technique. Dictionary is generated from the mask, using high performance *maskprocessor*[16]. This means, we have two dictionaries on input and we can proceed with performing a combination attack, as described in section 3.2. The mask transformation process can be seen in figure 12.

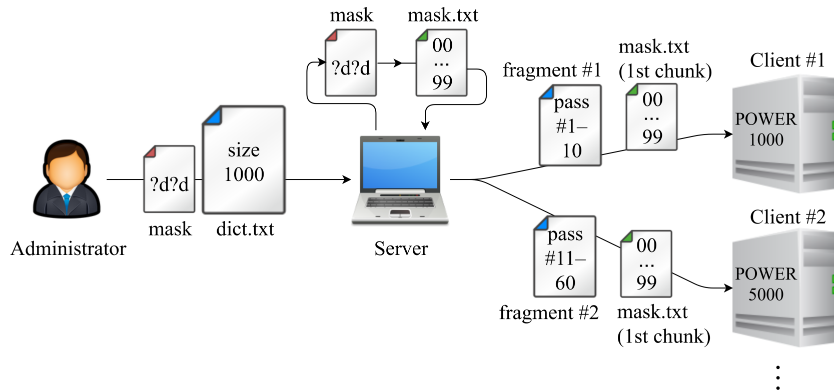

Fig. 12. Example of hybrid attack distribution

Since we transform hybrid attacks to combination attacks, Fitcrack supports the use of left and right password-mangling rule (see section 3.1.1) in the same way as with the combination attack.

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

4 Server-side subsystems

The server is responsible for the management of cracking jobs, and assigning work to clients. In terms of the *client-server* architecture, the service offered by the server is a *workunit* assignment.

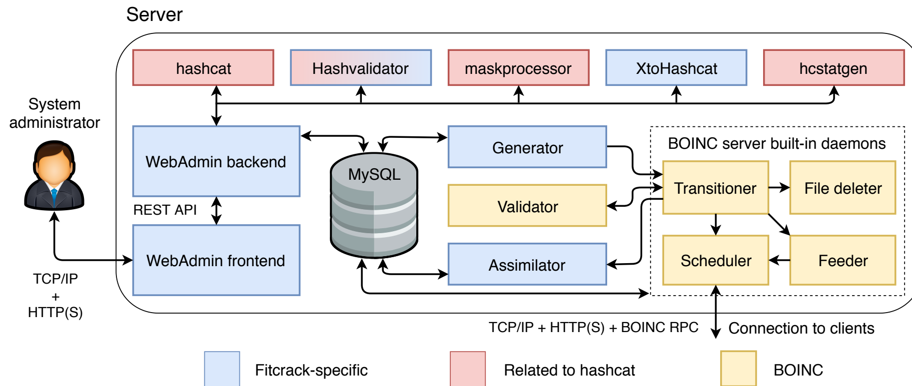

Fig. 13. The architecture of Fitcrack server

While for client, we support both Windows, and Linux nodes, the server has a Linux-only implementation. As illustrated in figure 13, Fitcrack server consist of multiple subsystems: *Generator* (see section 4.10), *Validator* (see section 4.11), *Assimilator* (see section 4.12), and *Trickler* (see section 4.13) are the main scheduling-related daemons implemented within Fitcrack. They closely relate to BOINC built-in subsystems which are: *Transitioner* (see section 4.14), *Scheduler* (see section 4.15), *Feeder* (see section 4.16), and *File deleter* (see section 4.17). To perform basic operations and remotely manage the entire system, Fitcrack provides a web-based user interface called Fitcrack *WebAdmin* (see section 4.2). WebAdmin uses a set of external tools as depicted in figure 13: *hashcat*, *Hashvalidator*, *maskprocessor*, *XtoHashcat*, and *hcstatgen*.

For storing all cracking-related information we use a MySQL[17] database. The structure of the database is described in section 7. Since both the system administrator, and BOINC client (on client-side) communicate via HTTP(S), we use Apache[18] HTTP server. Apache runs two applications: Fitcrack *WebAdmin*, and the set of CGI scripts of the *Scheduler* subsystem. All subsystems are run by one of two Linux users:

- **Apache user** (`apache`, or `www-data` by default) runs the Apache-based subsystems: WebAdmin and Scheduler,
- **BOINC user** (`boincadm` by default) runs the rest.

[17] https://www.mysql.com/
[18] https://httpd.apache.org/

4.1 Server directory structure

The subsystems are located in various directories:

- `PROJECT_ROOT` - the directory od BOINC Fitcrack project, by default: `/home/boincadm/projects/fitcrack`
  - `apps` - binaries of client applications: *hashcat*, and *Runner*
  - `bin` - binaries of server daemons (*Generator*, *Assimilator*, ...)
  - `cgi-bin` - CGI scripts of *Scheduler*,
  - `log_<hostname>` - logs of server daemons,
  - `pid_<hostname>` - PID files of server daemons,
  - `download` - data to be downloaded by client,
  - `html` - BOINC project website files,
  - `keys` - encryption keys,
  - `upload` - directory for client uploads,
  - `templates` - templates defining workunits,
- `APACHE_ROOT` - the document root of Apache HTTP server, by default: `/var/www/html`
  - `fitcrackFE` - frontend of *WebAdmin*,
  - `fitcrackAPI` - backend of *WebAdmin* with tools,
    - `src` - Python scripts of backend,
    - `hashcat-4.2.1` - hashcat binaries,
    - `hashcat-utils` - hashcat-related utilities,
    - `hashvalidator` - the *Hashvalidator* tool,
    - `maskprocessor` - the *maskprocessor* tool,
    - `xtohashcat` - *XtoHashcat* hash extraction tool,
- `COLLECTIONS_ROOT` - the directory for shared data, by default: `/usr/share/collections`.
  - `charsets` - user-defined character sets,
  - `dictionaries` - dictionaries for attacks,
  - `encrypted-files` - the inputs of XtoHashcat,
  - `markov` - user-defined Markov statistics files,
  - `masks` - files with password masks,
  - `rules` - files with password-mangling rules.

4.2 WebAdmin

We created a completely new solution for remote management of Fitcrack. The application is called *WebAdmin* and consist of two separate parts: *frontend* described in section 4.3 and *backend* described in section 4.4. The two parts communicate using a REST API.

4.3 WebAdmin frontend

The frontend is written in *Vue.js* and allows the administrator to manage different parts of the system as depicted in figure 14. Under *Jobs* tab, the administrator can add, modify and manage all cracking jobs. *Hosts* section provides an overview of connected clients, their software and hardware specification, jobs the clients were participating on, and workunits assigned to them. Every hash, cracked or not, can be viewed in a summary within *Hashes* tab. *Dictionaries* tab can be used to manage and add password dictionaries. Fitcrack supports three ways of adding new dictionaries: i. importing directly from the server; ii. uploading new via web using HTTP; iii. upload using SFTP/SCP, if configured. Using *Rules* tab, the administrator can manage **.rule* files contaning the password-mangling rules for hashcat. *Charsets* and *Masks* tabs allows to manage character sets and password masks used for mask attack. Since for mask attack, hashcat generates passwords using *Markov chains* [10], it is necessary to provide a **hcstat* / **hcstat2* (for hashcat 4+) file with per-position character statistics. In *Markov chains* tab, Fitcrack supports adding *hcstat* files either by uploading an existing file, or by generating a new one. The second option stands for an automated training on a password dictionary using *hcstat2gen* tool. Least but not last, in *Users* tab, WebAdmin allow to manage user accounts and permissions.

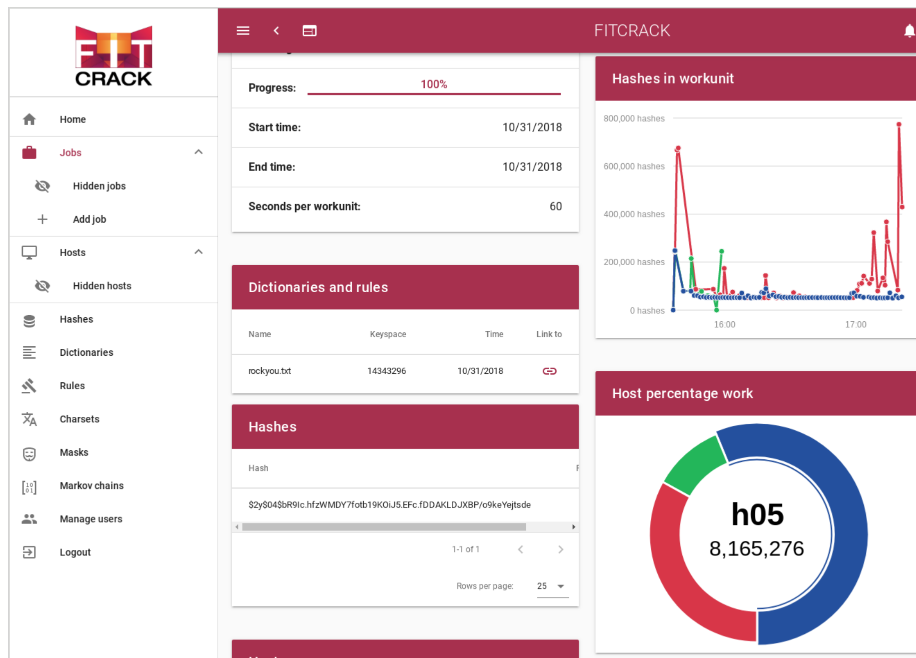

Fig. 14. The interface of Fitcrack WebAdmin

4.4 WebAdmin backend

The backend, written in Python 3, is based on Flask[19] microframework, communicating with Apache or NGINX HTTP server using Web Server Gateway Interface (WSGI). It implements all necessary endpoints of the REST API used by the frontend, e.g. handles requests for creating new jobs, and others. Using SQLAlchemy[20], the backend operates a MySQL database which server as a storage facility for all cracking-related data. For selected operations, the WebAdmin uses a set of external utilities and programs:

- **hashcat** - for calculating the keyspace of masks (see section 5.4),
- **HashValidator** - for validating hash formats (see section 4.6),
- **maskprocessor** - for creating dictonaries from masks in hybrid attack (see section 4.7),
- **XtoHashcat** - for hash extraction from (see section 4.8),
- **hcstat2gen** - for generating Markov statistics (see section 4.9).

For handling frontend requests, the backend provis the following endpoints:

- `/charset` - operations with charset collection,
- `/charset/<id>` - operations with specific charset,
- `/dictionary` - operations with dictionary collection,
- `/dictionary/<id>` - operations with specific dictionary,
- `/graph/hostPercentage/<job_id>` - fetch ratio of hosts work,
- `/graph/hostsComputing` - fetch hosts computed hashes,
- `/graph/hostsComputing/<id>` - fetch specific host, computed hashes,
- `/graph/jonProgress` - fetch data to draw jobs progress graph,
- `/graph/jonProgress/<id>` - fetch data to draw specific job progress graph,
- `/hashcat/hashTypes` - get supported hashtypes,
- `/hashes` - fetch page of hashes added to system,
- `/hosts` - operations with host collection,
- `/hosts/<id>` - operations with specific host,
- `/hosts/info` - fetch info about hosts,
- `/job` - operations with job collection,
- `/job/<id>` - operations with specific job,
- `/job/<id>/action` - operations with job (start, stop, restart),
- `/job/<id>/host` - fetch hosts for specific job,
- `/job/<id>/workunit` - fetch workunits for specific job,
- `/job/crackingTime` - get estimated cracking time for job,
- `/job/info` - fetch info about jobs,
- `/job/verifyHash` - validate hash in HashValidator (see 4.6),
- `/markovChains` - operations with `hcstat2` files collection,
- `/markovChains/<id>` - operations with specifix `hcstat2` file,
- `/markovChains/makeFromDictionary` - make `hcstat2` file from dictionary,
- `/masks` - operations with mask files collection,

[19] http://flask.pocoo.org/

[20] https://www.sqlalchemy.org/

- `/masks/<id>` - operations with specific mask file,
- `/masks/<id>/download` - download `hcstat2` file,
- `/notifications` - fetch netifications,
- `/notifications/count` - fetch unseen notifications count,
- `/protectedFiles` - upload encrypted file a get it's hash,
- `/rule` - operations with rule collection,
- `/rule/<id>` - operations with specific rule,
- `/serverInfo` - fetch server info,
- `/user` - operations with user collection,
- `/user/<id>` - operations with specific user,
- `/user/isLoggedIn` - find out if user is logged in system,
- `/user/login` - login user to system,
- `/user/logout` - logout user from system,
- `/user/role` - operations with user role collection,
- `/user/role/<id>` - operations with specific user role.

4.5 hashcat

Since the keyspace computed by hashcat may differ form the actual number of checked passwords, as described in 2.3, we need hashcat on the server-side as well. Every time the WebAdmin needs to calculate the keyspace for a given attack, it runs hashcat with the `--keyspace` argument. For more about hashcat see section 5.4.

4.6 Hashvalidator

HashValidator is our custom tool mostly based on the original hashcat sources. HashValidator is able to verify the syntax of input hashes. The tool is used by WebAdmin at the time of creating a new cracking job. Whenever a user enters one or more input hashes, Hashvalidator is called to verify the syntax. The usage is defined as follows:s

```
./hashValidator -m <hash_type> [ <hash> | <hash_file> ]
```

where *hash_type* is a unique number defining the type of hash (see section 1.1). The next argument is either a *hash* to be verified, or a text file containing hahes - one per line.

4.7 maskprocessor

For generating dictonaries from masks in the hybrid attack, we use *maskprocessor*[21]. It is high-performance word generator which can be used to transform masks to password dictionaries. For a given mask, it produces a dictionary of all candidate passwords generated from the mask. It has built-in charsets for standard symbol groups, namely `?l`, `?u`, `?d`, `?s`, `?a`, `?b`, and support for up to four custom character sets like in hashcat's *brute-force attack* (see section 3.3).

[21] https://hashcat.net/wiki/doku.php?id=maskprocessor

4.8 XtoHashcat

To get a password securing an encrypted container, it is necessary to extract all cracking-related metadata, as described in section 2.1. For using hashcat, users need to extract hashes manually, e.g. using third-party scripts. For easier use, Fitcrack provides an abstraction over this process, and thus accepts even the original encrypted containers as an input.

This is, where *XtoHashcat* comes to use. XtoHashcat is our custom tool written in Python 3. The tool can automatically detect the format of input encrypted media, and extract the hash necessary for cracking. For detection, it scans file signatures and optionally file extensions. Once the format is detected, one of the open-source scripts[22,23] is used to extract the hash.

Thanks to this approach, the uploading and cracking of the supported file is transparent for the user. The hash is extracted in the background without the need of entering the format number or running external extraction scripts. The usage is defined as follows:

```
./XtoHashcat.py <path> [-f <hash_type>]
```

The first argument, *path*, describes the location of the encrypted input file. The second argument, *hash_type* is optional, and can be used to specify the format in hashcat hash type format (see section 1.1). At the time of writing this report, XtoHashcat supports the following input formats:

- **MS Office documents** (`-f 9400-9800`) [17, 18],
- **PDF documents** (`-f 10400-10700`) [1],
- **RAR archives** (`-f 12500/13000`) [13, 9],
- **ZIP archives** (`-f 1300`) [3],
- **7z archives** (`-f 11600`) [16].

If the inputs are processed successfully, the output has two lines. The first one contains the extracted hash, and the second contains a number representing the detected hash type.

4.9 hcstat2gen

As described in section 3.3, brute-force attack uses Markov chains to generate symbols in password candidates. The technique requires a `.hcstat2` file with Markov statistics in the form of character probability matrixes. The user can either use a default `hashcat.hcstat2` one, or select a custom statistics file.

Fitcrack WebAdmin supports automatic creation of new `.hcstat2` files by processing existing dictionaries. For this purpose, it uses a utility called **hcstat2gen**[24]. This tool generates a custom Markov statistics file from selected dictionary. The usage is:

[22] https://github.com/stricture/hashstack-server-plugin-hashcat
[23] https://github.com/magnumripper/JohnTheRipper/
[24] https://hashcat.net/wiki/doku.php?id=hashcat_utils#hcstat2gen

```
./hcstat2gen.bin hcstat2_output_raw.bin < dictionary.txt
```

Starting from version 4.x, hashcat reqires the file to be LZMA-compressed[25] compression. . The compression can be done in the following way:

```
lzma --compress --format=raw --stdout -9e
  hcstat2_output_raw.bin > output.hcstat2
```

4.10 Generator

Generator is a server daemon responsible for creating new workunits for hosts. There are two types of workunits – benchmark and normal cracking tasks. The details of workunit types and parameters are described in section 6. The benchmark can be run for one format only or for all supported formats. The second case is called a *complete benchmark*, and is performed only once whenever a new client is connected to the server. The goal of the complete benchmark is to measure client's capabilities, i.e. achieveable cracking speeds for all hash supported algorithms.

The classical benchmark for a single format is run always at the start of the job to measure the current speed of connected hosts. The complete benchmark is run automatically after new host connects to server, when `default_bench_all` flag is set in *fc_ settings* table. Results are saved and then used for computing the expected attack duration before the attack itself starts.

There are always two cracking workunits ready for 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. This way, we minimize the communication overhead.

This daemon also deals with disconnected hosts and computation errors. When an incorrect result is delivered by a host or a workunit deadline is reached, this workunit is tagged with *retry* flag and a new copy is generated.

The Generator communicates with the rest of the server using the database only. This approach is similar to most of the BOINC daemons. The generator also creates the input files, which are sent to the hosts. The number of these files varies, depending on the type of the attack. *data* and *config* files are always sent, containing input hashcat hash and needed metadata respectively. For dictionary and combinator attacks, *dict1* and *dict2* input files may be needed, containing list of passwords. If the administrator wants to apply rules to the dictionary, *rules* input file is created. For a mask attacks using Markov chains, *markov* input file,

[25] `https://en.wikipedia.org/wiki/Lempel%E2%80%93Ziv%E2%80%93Markov_chain_algorithm`

containing *hcstat2* file is created. The simplified functionality of the Generator daemon is described by algorithm 2.

Algorithm 2: Generator daemon algorithm

```
1  while (1) do
2      // Inicialization
3      if Any Jobs reached deadline then
4          Set them to Finishing status (12)
5      foreach Running Job (status ≥ 10) do
6          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 Host is in Running status (10) then
16                 Generate a new workunit or a make a copy from retry
17                 if No workunits could be generated then
18                     Set Job to Finishing status (12)
19             if Host is in Finishing status (12) then
20                 Try to copy a retry workunit, otherwise set Host to Done
                    status (3)
21         // Job finished
22         if Job status is Finishing (12) and no Jobs are generated then
23             Check the end conditions
                (Finished/Exhausted/Timeout/Paused)
24     Wait a short time interval before the next iteration
```

4.11 Validator

Incoming BOINC results must be validated before parsing. This process may validate syntax only or may compare multiple results for the same workunit. This way, we can detect the corrupted nodes, which supply us with incorrect results. However, with each host receiving the same workunit, the speed halves. Fitcrack system was designed to be used in private networks, so this functionality is not used.

Because of this, Fitcrack uses default BOINC validator, which checks the syntax of the result only.

4.12 Assimilator

Assimilator is a server daemon which parses the results supplied by hosts. As mentioned in section 4.10, there are three possible results – the benchmark for one format, complete benchmark, and a normal cracking job. The Assimilator, depending on the type of result, is able to modify the database or cancel running workunits.

The results are sent in a custom format, where pieces of information are separated by a newline. The meaning of the lines varies, depending on the type of workunit. However, the first two lines always inform us about the workunit type and the result. At the first line, letter `b` is signalling a result from the benchmark workunit, letter `a` result from complete benchmark, and letter `n` result from a normal cracking task. At the second line, there is always a result code. Generally, code 0 is signaling a successful result while codes greater than two are signaling computation error. The simplified functionality of Assimilator daemon is described by algorithm 3.

4.13 Trickler

Trickler daemon was created to enable information exchange between server and clients even during the cracking. It is used to periodically send progress of the current cracking workunit. This way, we know the current state of the cracking even with several hours long workunits.

The messages are in XML format and are saved into the database. From here, Trcikler daemon reads them and updates the Fitcrack table *fc_workunit* with progress. The old entries in the database are periodically removed.

4.14 Transitioner

Transitionor is default BOINC daemon that keep databased synchronized. It updates workunits and their results when needed. All other daemons depends on Transitioner's work. Fitcrack system uses deafult BOINC implementation without any modifications.

4.15 Scheduler

Scheduler is another BOINC program. It is responsible for communication with hosts. The communication consists of periodical exchange of scheduler request and reply messages in XML format. In those, all information is sent, including new workunits. In this case, the workunit must be generated first by the Generator daemon.

Although Fitcrack system uses the default Scheduler implementation, some adjustments were made. Most importantly, the program was modified so that with every reply sent to host, the Fitcrack *fc_host_status* table is updated with the current timestamp. Using this modification, we can see which hosts are currently up and running.

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                 Switch the Job state to Finished (1)
13                 Cancel all running Workunits of the Job
14                 Set finished flag to all Workunits
15                 Read the cracking time and save it
16             else
17                 if No passwords found (code 1) then
18                     Modify the workunit size according to 1.
19                     Update the current index used for planning
20                 else
21                     // Computation error
22                     Cancel host workunints
23                     Set Host status to Benchmark (0)
24         case bench_ all do
25             if Result is OK (code 0) then
26                 Read the power list and save it to database
27             else
28                 Plan a new benchmark
```

4.16 Feeder

Fitcrack system uses default BOINC implementation of Feeder daemon. It works closely with Scheduler and is responsible for distributing chunks of shared memory.

4.17 File deleter

File deleter is one of many BOINC server daemons. Fitcrack system uses default BOINC implementation of this daemon. It's responsibility lies in deleting input and output files of completed and assimilated workunits. This daemon can be run periodically in user defined intervals to clear the disk space.

5 Client-side subsystems

Clients represent the actual cracking nodes. Fitcrack can be run on 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 *BOINC Client* (see section 5.1), and optionally *BOINC Manager* (see section) providing a graphical user interface to BOINC Client.

Once the BOINC Clients connects and authenticates to the server, all necessary binaries are downloaded automatically before the actual work is assigned. The binaries involve two applications: *hashcat* as the "cracking engine" (see section 5.4), and *Runner* (see section 5.3) which server as a wrapper encapsulating and controlling operations with hashcat. The architecture of Fitcrack client is illustrated in figure 15.

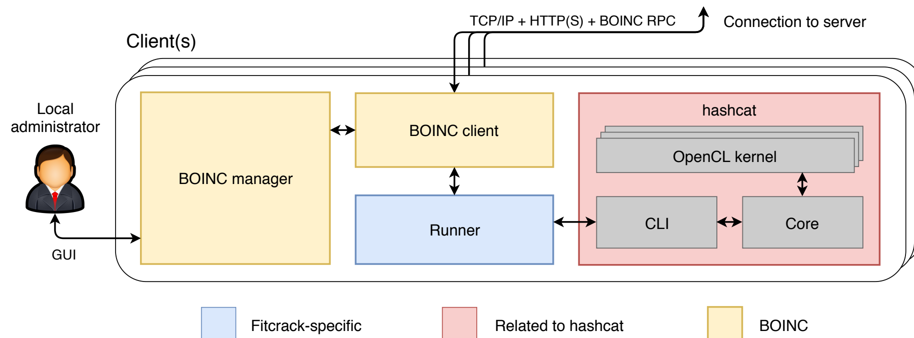

Fig. 15. The architecture of Fitcrack server

5.1 BOINC Client

BOINC Client[26], also referred to as *core client*, is the main host application of Fitcrack system. It is required to be manually installed to the host system. The application ensures communication with the project server. Both the initialization of project on the host and the retrieving and reporting of individual workunits. Project initialization consists of the authentication of user, download of the project specific binaries and information.

Another job of the application is the execution of project specific host applications. On receive of every new workunit it:

1. Creates copy of the default project files into new empty directory.
2. Downloads the workunit specific file from the server.
3. Adds files containing soft-links to workunit specific files as to the directory created in item 1.

[26] https://boinc.berkeley.edu/wiki/BOINC_Client

4. Executes the host application in that directory with specified parameters.
5. Retrieves the exit code of the host application.
6. Reports the generated result to the server.
7. Deletes the created directory with all of its subfolders and files.

BOINC Client can be configured to run the computations only when certain conditions are met. Some of them are processor utilization, disk space usage, network transfer limits, exclusive applications aren't running. Also, it is possible to set daily schedules and others.

BOINC Client can be executed as daemon by cron, manually or at the startup of the system. It can also run as CLI application in terminal. It communicates with the server via *BOINC scheduling server protocol* using either HTTP or HTTPS. It executes and controls the host application via system calls.

5.2 BOINC Manager

BOINC Manager[27] is in general graphical interface of BOINC Clien. It allows to add projects, control progress of tasks, review application logs, configure user setting and logging preferences. It communicates with the client over graphical user interface remoter procedure calls (GUI RPS).

BOINC Manager can be run either at the system start-up or manually and can be shutdown at any time without affecting the computation process. It requires BOINC Client to be running for its proper functioning.

5.3 Runner

Runner is a wrapper of hashcat, designed to be used as either standalone tool, simplifying control of hashcat, or as middleware in *BOINC* system. The code uses the C++98 standard extended by few functions from C99 standard. It is written in the way to be compilable into a static binary for both Linux and Windows.

5.3.1 Basic operation Once runner is started, it:

1. reads the `config` file (see section 6.1), and coverts the options to *hashcat* parameters,
2. launches hashcat,
3. monitors the cracking progress of cracking,
4. gathers results, and creates and output file (see section 6.2) which is passed to BOINC.

Runner is launched by the BOINC client. All information needed by the application is stored in several files which are required to be in the same directory as the executable. The files are:

[27] https://boinc.berkeley.edu/wiki/BOINC_Manager

- required files:
  - `hashcat_files_v421_1.zip` - containing all *hashcat* file like OpenCL kernels, `*.hcstat`, `*.hctune`, etc.,
  - `config` - the workunit input configuration file – see section 6.1,
  - `data` - file with *hashcat* acceptable hash to be cracked,
- optional files:
  - `dict[1-2]` - files with the dictionary of passwords saved in them, in *hashcat* acceptable format,
  - `rules` - a file with password-mangling rules (see section 3.1.1),
  - `markov` - a `.hcstat2` file with Markov character statistics (see section 3.3.2),
- output files:
  - `out` - this output file with cracking results in the server understandable format – see section 6.2,
  - `stderr.txt` - execution log.

All mentioned files can contain *BOINC*-like soft-link to the real file. Soft-link may be created by making text files with following XML element with path to the real file as its value `<soft_link></soft_link>`. *Runner* resolves such soft-link to the absolute path of the file which it then uses instead of the soft-link file.

The arguments of hashcat, and the behavior of Runner depends on selected attack mode, attack submode, and options specificied in the workunit `config` file. For detailed information, see section 6.1.

5.3.2 Host specific configuration

Runner also supports host specific specification of *hashcat* parameters. It is designed to be used for the specification of workload-`w`), which OpenCL devices to use(`-d`), whether should *hashcat* ignore errors(`--force`) and such thing which aren't generalized for all host by their nature.

File with such additional configurations has to be placed at `/etc/<BOINC_project_name>.conf` on Linux and under `C:\ProgramData\BOINC\<BOINC_project_name>.conf` on Windows. If you would like to run *Runner* as standalone then create config using the same directories but name it `standalone.conf`. It is just inlined into *hashcat* command.

5.4 hashcat

Fitcrack uses *hashcat* as a tool realizing the actual password recovery. It allows the system to support a lot of the hash formats / algorithms. Also, it uses the kernels written in OpenCL C for the implementation of hash function and alogrithm. OpenCL enables the use of the hardware accelerators such as graphics cards and feild programmable gate arrays (FPGA) or even CPUs. For *hashcat* to function correctly the proper driver with OpenCL support has to be installed for chosen processing device. The tool itself doesn't have to be install as the system ships its own *hashcat* binaries to the clients.

The *hashcat* tool is executed via the module Runner(section 5.3). The module sets the execution parameters and processes the outputs of the tool and its exit code.

6 Client-server communication

While the underlaying communication is handled by BOINC using *BOINC scheduling server protocol*[28], the inputs and outputs of each *workunit* are controled by Fitcrack. For each job, Fitcrack defines a number of input and output files. Which files are used depends on the attack mode. The attack modes and their numbers are defined in section 3.

In BOINC, all workunit-related files are described by input and output *templates*[29] located on the server in `PROJECT_ROOT/templates` directory (see section 4.1). While *input templates* define input files downloaded by a client from the server before a workunit is started, the *output templates* define the output files which are sent by the client back to the server after the job is finished. In Fitcrack, we have three types of workunits:

- **Benchmark** workunit - is sent to each client at the beginning of each job. The goal is to determine the client's current speed which is used in the adaptive scheduling algorithm (see section 2.4). Once finished, the result of the benchmark for given hash type is stored in `fc_benchmark` table within the database (see 7). If the record already exists, it is updated by the newly-measured one. In workunit config file (see below), the computation mode is set to "`b`".
- **Benchmark all** - is a workunit which lets the client perform the benchmark over all supported hash algorithms. It is used only within a special (hidden) job called `BENCH_ALL` which is always present in the system. By default, the complete benchmark is performed only once, and is started whenever a new client is connected to the server. The goal is to scan the capabilities of the client. The resulting speeds for all hash types are saved to `fc_benchmark` table within the database (see 7). In workunit config file, the computation mode is se to "`a`" which stands for "all".
- **Normal** workunit - is a regular piece of cracking work sent to the client. What hash type and how exactly is cracked specifies the `config` file described in section 6.1. In workunit config file, the computation mode is set to "`n`".

6.1 Files transferred from server to client

In Fitcrack, we have six different input templates. The `bench_in` template used for bechmarking. For rest, each template corresponds to a number of attack modes and submobdes (represented by the *attack number* - see 3):

- **bench_in** - used for *benchmark* workunits (no attack),
- **dict_in** - used for *dictionary* attacks (00),
- **rule_in** - used for *dictionary* attacks with *rules* (01),
- **combinator_in** - used for *combination* attacks (10, 11, 12, 13),

[28] https://boinc.berkeley.edu/trac/wiki/RpcProtocol
[29] https://boinc.berkeley.edu/trac/wiki/JobTemplates

- **mask_in** - used for *brute-force* attacks with mask (30),
- **markov_in** - used for *brute-force* attacks with mask and user-defined Markov statistics file (31, 32).

Each template defines the use of one of more of the following input files:

- **config** - a configuration of the *workunit* (see below),
- **data** - a file containing one, or more *input hashes*,
- **dict1** - a password dictionary number one,
- **dict2** - a password dictionary number two,
- **rules** - a file with *password-mangling rules*,
- **markov** - a *.hcstat* file with Markov statistics (see section 3.3).

Table 4 shows the relationship between templates, attack modes, and input files. Each row stands for a single template. The first column shows the name of the template. The second column contains numbers of attacks in which the template is used. For each input file, there is a column containing "X" if the file is used within the template. For example, template `mask_in` define the use of two input files: `config` and `data`.

template	attacks	config	data	dict1	dict2	rules	markov
bench_in		X					
dict_in	00	X	X				
rule_in	01	X	X	X		X	
comb_in	10, 11, 12, 13	X	X	X	X		
mask_in	30	X	X				
markov_in	31, 32	X	X				X

Table 4. Input templates used by boinc

The **config** is a text file defining the workunit, e.g. its attack mode and *keyspace* (see 2.3). For easy parsing on the client-side, we use the *Type-length-value*[30] (TLV) representation. Each line of the config file has the following syntax:

```
|||name|type|length|value|||
```

The *name* identifies the configration parameter. Allowed names are listed in table 6. The *type* matches one of the data types defined in table 5. The *length* says how many characters are in the *value* part. For example:

```
|||hash_type|UInt|4|9400|||
```

describes parameter names `hash_type` which should be saved as a 32-bit unsigned integer. The value is 9400 of 4 digits.

[30] https://named-data.net/doc/NDN-packet-spec/current/tlv.html

On the client-side, the configuration parameters are interpreted by the *Runner* subsystem (see 5.3). Many of them affect the arguments hashcat is started with. Table 6 shows all workunit parameters supported by Fitcrack together with hashcat's arguments they are related to. For example, we can see the connection of the `start_index` and `hc_keyspace` parameters to hashcat's `--skip` and `--limit` arguments, as discussed in section 2.3.

Bool	Boolean: 0 means FALSE, 1 means TRUE
Char	C-like 8-bit unsigned char
String	C-like sequence of chars
Int	32-bit signed integer
UInt	32-bit unsigned integer
BigInt	64-bit signed integer
BigUInt	64-bit unsigned integer

Table 5. Data types supported in Fitcrack config

name	**describtion**	**hashcat arg.**
attack_mode	attack mode (see 3)	`-a`
attack_submode	attack submode (see 3)	
hash_type	type of the hash (see 1.1)	`-m`
name	the name of the cracking job	
charset1	user-defined charset number 1 (see 3.3)	`-1 charset1`
charset2	user-defined charset number 2	`-2 charset2`
charset3	user-defined charset number 3	`-3 charset3`
charset4	user-defined charset number 4	`-4 charset4`
rule_left	rule for the left dictionary (see 3.2)	`-j`
rule_right	rule for the right dictionary	`-k`
start_index	starting password index (see 2.3)	`--skip`
hc_keyspace	keyspace of the workunit	`--limit`
mask_hc_keyspace	keyspace of the entire mask	
dict_hc_keyspace	keyspace of the dictionary fragment	
markov_threshold	threshold for Markov model (see3.3)	`--markov-threshold`

Table 6. Parameters used in the config file

Not all config parameters are used in every workunit, e.g. in dictionary attack, we have no mask, etc. Table 7 shows in which attack modes and submodes the parameters are used. If a parameter is used in given mode and submode, "X" is displayed in the corresponding column. For example, `mask_hc_keyspace` parameter defining the hascat's keyspace of a given mask is only used within a *brute-force attack*, so "X" is in columns related to *attack mode* number 3. For

benchmark workunits, the *attack mode* and *attack submode* are both set to letter "B" in the table.

Another important aspect of the workunit config parameters is by which part of Fitcrack system they were added, and in which context they are considered valid. Depending on their nature and origin, we can distinguish between two types of parameters:

- **job-wide** parameters - which remain valid and constant throughout the lifetime of the entire job. Such parameters include `attack_mode`, `hash_type`, `name` of the job, etc. These parameters are added to the config **only once** by *by WebAdmin*. In the "O" column (meaning the origin of the parameter), these parameters have letter "W" which stands for *WebAdmin*.
- **workunit-specific** parameters - are valid and constant only within each workunit, e.g. `start_index`, or `hc_keyspace` defining the range of indexes, as described in section 2.3. In other workunits of the same job, these parametrs may have completely different values. When the *Generator* subsystem (see section 4.10) creates a new workunit, it creates the input configuration file (named `config`) by first filling it with all *job-wide* parameters, and then it appends the *workunit-specific* ones. In the "O" column, the *workunit-specific* parameters have letter "G", since there are added by the *Generator*.

		attack mode	B	0		1				3		
		attack submode	B	0	1	0	1	2	3	0	1	2
name	**type**	O										
attack_mode	UInt	W	X	X	X	X	X	X	X	X	X	X
attack_submode	UInt	W	X	X	X	X	X	X	X	X	X	X
hash_type	UInt	W	X	X	X	X	X	X	X	X	X	X
name	UInt	W	X	X	X	X	X	X	X	X	X	X
charset1	String	W								X	X	X
charset2	String	W								X	X	X
charset3	String	W								X	X	X
charset4	String	W								X	X	X
rule_left	String	W					X		X			
rule_right	String	W						X	X			
mask	String	G								X	X	X
start_index	BigUInt	G		X	X	X	X	X	X	X	X	X
hc_keyspace	BigUInt	G		X	X	X	X	X	X	X	X	X
mask_hc_keyspace	BigUInt	G								X	X	X
mode	String	G	X	X	X	X	X	X	X	X	X	X
markov_threshold	UInt	G									X	X

Table 7. Use of config parameters within different attacks

An example of a concrete workunit `config file` is:

```
|||attack_mode|UInt|1|0|||
|||attack_submode|UInt|1|0|||
|||hash_type|UInt|4|9400|||
|||name|String|4|test|||
|||start_index|BigUInt|1|0|||
|||hc_keyspace|BigUInt|6|135985|||
|||mode|String|1|n|||
```

The config defines a *workunit* within a cracking *job* named `test`. Attack *mode* 0 defines a dictionary attack, attack *submode* 0 stands for the classic dictionary attack without the use of password-mangling rules. *Start index* equal is typical in dictinary attack, since we only send fragments of the original dictionary, as described in section 3.1. The total *hashcat's keyspace*, in this case the number of passwords in dictionary fragment, is 135985. The *mode* is set to "n" which stands for (n)ormal cracking.

6.2 Files transferred from client to server

While workunits have multiple input files, the output file described by `app_out` template is only one - the `out` file containing:

```
<mode>
<status_code>
<info>
```

where *mode* refers to the *computational mode*: i) "b" for benchmark, ii) "a" for benchmark all, and ii) "n" for normal tasks. The meaning of status codes is described in table 8.

code	benchmark (b)	benchmark all (a)	normal task (n)
0	successfull	(partial) success	at least one hash cracked
1	-	-	finished, no hash cracked
3	-	-	input error
4	computation error	computation error	computation errror

Table 8. Meaning of codes in *out* file.

The *info* part is different for each *computational mode* and *status code*. The contents of the *info* part is defined as follows:

- **Benchmark workunits** (mode = **b**); the allowed status codes are:
  - **0** - means the benchmark was successfull, the *info* part consist of two lines: the first one contains the cracking speed in hashes per second (integer), the seconds one contains the total time of the benchmark (double). The contents of the `out` are:

```
b
0
<cracking_speed(power)> - integer
<cracking_time> - double
```

- **4** - is used if an error occured during the benchmark. The *info* part consist of two lines: the first one contains hashcat's return code, the seconds one contains the error message returned by hashcat. The contents of the `out` are:

```
b
4
<hashcat_exit_codes> - integer
<hashcat_exit_info> - string (may be empty)
```

- **Benchmark all workunits** (mode = **a**); the allowed status codes are:
  - **0** - means the complete benchmark was successfull at least for some hash types. The *info* part consist of lines containing the number of a *hash type* number, colon (:), and measured cracking speed in hashes per second:

```
a
0
<cracking_time> - double
<hash_type>:<cracking_speed>
<hash_type>:<cracking_speed>
<hash_type>:<cracking_speed>
...
```

    If benchmarking of any hash type encountered an error, the `cracking_speed` is set to 0.

  - **4** - the benchmark was not successful for any hash type. This means that no run of hashcat was successful. The *info* part contains the error message returned by hashcat. The contents of the `out` are:

```
a
4
<hashcat_exit_info>
```

- **Normal workunits** (mode = **n**); the allowed status codes are:
  - **0** - means the password was found for one or more hashes. The *info* part consist of a line contaning the cracking time in hashes per second, and one or more lines containing the cracked *hash*, a semicolon (:), and the cracked password in hex form. The *out* file has the following contents:

```
n
0
<cracking_time> - double
<cracked_hash>:<password(hexa encoded)> - string
<cracked_hash>:<password(hexa encoded)> - string
<cracked_hash>:<password(hexa encoded)> - string
...
```

- **1** - no password was found within the workunit. The *info* part contains the cracking speed in hashes per second. The *out* file has the following contents:

```
n
1
<cracking_time> - double
```

- **3** - means the client has encountered an error due to incorrect inputs, and thus hashcat was not able to start successfully. The *info* part contains two lines. The first line contains the exit code of hascat, while the second line contains the error message displayed by hashcat. The *out* file has the following contents:

```
n
3
<hashcat_exit_code> - integer
<hashcat_error_info> - string
```

- **4** - means a computational error occured within the cracking. The *info* part contains two lines. The first line contains the exit code of hascat, while the second line contains the error message displayed by hashcat. The *out* file has the following contents:

```
n
4
<hashcat_exit_code> - integer
<hashcat_error_info> - string
```

6.3 Trickle messages

Whereas previous sections described client-server communication done before a workunit is started, and after the workunit is finished, the client-side also informs the server about partial progress on currently-computer workunit. This is performed in *Runner* subystem (see section 5.3) using BOINC *Trickle message API*[31] which is used to send *trickle messages* to the server. The messages are processed by the *Trickler* daemon (see section 4.13). Each trickle message has the following syntax:

```
<workunit_name>wuName</workunit_name>
<progress>wuProgress</progress>
<speed>wuSpeed</speed>
```

where *wuName* corresponds to the name of the workunit in BOINC `workunit` tabe, *wuProgress* is a value from 0.0 to 100.0 defining the current progress on the workunit, and *wuSpeed* stands for the cracking speed in hashes per second.

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

7 MySQL database

To store all cracking-related information, Fitcrack server (see section 4) uses a MySQL database. At the time of writing this technical report, Fitcrack is compatible with MySQL[32] server 4.0.9 or higher, respectively MariaDB[33] server 10.0 or higher. The database contains two types of tables:

- **BOINC tables** - created by BOINC `make_project` script and maintained by BOINC server daemons: *Transitioner*, *Scheduler*, *Feeder*, and *File Deleter* (see section 4). Fitcrack-specific subsystems use only read-only access to these tables. BOINC tables are described in section 7.1.
- **Fitcrack tables** - created by SQL scripts of Fitcrack server, respectively Fitcrack installer, and used by Fitcrack *WebAdmin*, *Generator*, *Assimilator*, *Validator*, and *Trickler*. Fitcrack tables are described in section 7.2.

7.1 The overview of BOINC tables

BOINC tables respect the BOINC database scheme[34]. The most important tables are:

- **platform** - defining compilation targets of the core client and/or applications. The core client is treated as an application; its name is *core_client*.
- **app_version** - defining versions of client-side application binaries. Each record contains an URL which the BOINC client uses for downloading the binaries, and the MD5 checksum to verify application integrity.
- **user** - describes user accounts used by BOINC client/manager to authenticate with Fitcrack server.
- **host** - lists all *hosts*, also referred to as *clients*, or *cracking nodes*. BOINC *Scheduler* daemon automatically adds new record to the database, whenever a new host is connected to the cracking network described in section 2.2.
- **workunit** - contains workunits, the smallest pieces of work assigned to hosts in terms of BOINC. The records include the count of results linked to the workunit, the number of workunits sent, succeeded, and failed.
- **result** - is filled with *workunit* results, whenever a result is dispatched by a host. The reords store information about CPU time spent within the workunit, exist status, and validation status.

There are also other tables used by the BOINC, defined on the BOINC website[35].

[32] https://www.mysql.com/
[33] https://mariadb.org/
[34] https://boinc.berkeley.edu/trac/wiki/DataBase
[35] https://boinc.berkeley.edu/

7.2 The overview of Fitcrack tables

Besides the default BOINC tables, Fitcrack uses the following additional tables:

- **fc_benchmark** - tady bude nejaky popis
- **fc_charset** - character sets,
- **fc_dictionary** - password dictionaries,
- **fc_hash** - password hashes,
- **fc_hcstats** - Markov statistics files,
- **fc_host** - actively cracking hosts,
- **fc_host_activity** - mapping of hosts to jobs,
- **fc_host_status** - status of hosts,
- **fc_job** - cracking jobs,
- **fc_job_dictionary** - mapping dictionaries to jobs,
- **fc_job_graph** - points in job progress graph,
- **fc_mask** - password masks,
- **fc_masks_set** - sets of password masks,
- **fc_notification** - various notifications,
- **fc_protected_file** - input files to XtoHashcat,
- **fc_role** - user roles in WebAdmin,
- **fc_rule** - files with password-mangling rules,
- **fc_settings** - global server sertings,
- **fc_user** - user accounts in WebAdmin,
- **fc_user_permissions** - per-job user permissions,
- **fc_workunit** - workunits (chunks of keyspace).

7.3 fc_benchmark

The table is used to store benchmarking results of hosts. Each record represents the cracking speed of given host and given hash algorithm. The structure of the table is defined as follows:

- **id** - primary key,
- **boinc_host_id** - host ID in BOINC `host` table,
- **hash_type** - hashcat's number[36] of hash algorithm,
- **power** - measured cracking speed in hashes per second,
- **last_update** - time of last update of the record.

7.4 fc_charset

This table stores information about user-defined character sets used for brute-force attack and hybrid attacks (see section 3). Each record corresponds to a single charset file located in `COLLECTIONS_ROOT/charsets` directory. The structure of the table is defined as follows:

- **id** - primary key,
- **name** - name of the charset (displayed in WebAdmin),
- **path** - the real name of the charset file,
- **time** - time the charset file was added to the system,
- **deleted** - flag (0/1) saying if the charset was deleted.

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

7.5 fc_dictionary

This table stores information about password dictionaries used for *dictionary attack* and *hybrid attacks* (see section 3). Each record corresponds to a single dictionary file located in `COLLECTIONS_ROOT/dictionaries` directory. The structure of the table is defined as follows:

- **id** - primary key,
- **name** - name of the dictionary (display in WebAdmin),
- **path** - the real name of the dictionary file,
- **keyspace** - the number of passwords in the dictionary,
- **time** - time the dictionary was added to the system,
- **deleted** - flag (0/1) saying if the charset was deleted.
- **modification_time** - last modification time of the dictionary file; used for decision, if the keyspace should be updated or not.

7.6 fc_hash

The table contains various hashes which are cracked within the jobs. Each record stands for a single hash. The structure of the table is defined as follows:

- **id** - primary key,
- **job_id** - ID of the corresponding job `fb_job` table,
- **hash_type** - hashcat's number defining the type of the hash,
- **hash** - the value of the hash,
- **result** - plaintext input defining the correct password, if found.
- **added** - time the hash was added to the system,
- **time_cracked** - time the hash was was cracked.

7.7 fc_hcstats

This table stores information about Markov `.hcstat2` statistics files used for *brute-force* and *hybrid attacks* (see section 3). Each record corresponds to a single `.hcstat2` located in `COLLECTIONS_ROOT/markov` directory. The structure of the table is defined as follows:

- **id** - primary key,
- **name** - name of the Markov statistics file (displayed in WebAdmin),
- **path** - the real name of the `.hcstat2` file,
- **time** - time the file was added to the system,
- **deleted** - flag (0/1) saying if the file was deleted.

7.8 fc_host

Is the table for storing information about active BOINC *hosts* (also referred to as *clients*, or *cracking nodes*) which are currently working on a cracking job. Each record represents an involvement of a host in a cracking job. In other words, the table binds records in BOINC `host` table to the records in *fc_job* table. The structure of the table is defined as follows:

- **id** - primary key,
- **boinc_host_id** - host ID in BOINC `host` table,
- **power** - cracking speed measured within host's last benchmark,
- **job_id** - ID of the job, the host is currently working on,
- **status** - the status of the host's involvement. Allowed states are:
  - **0 - benchmark** - the host is waiting for, or working on a benchmark,
  - **1 - normal** - the hosts is working on a cracking job,
  - **2 - validation** (currently not used) - the host was asked to validate if a password is truly correct (NOTE: *designed as a solution to untrusted environment*),
  - **3 - done** - the host has finished all work on the given job, and is not participating on the job anymore,
  - **4 - error** - the host encountered an error during the computation,
- **time** - time the record was added to the database.

7.9 fc_host_activity

While `fc_host` table says which hosts are "currently working on" which job, `fc_host_activity` says which host "should participate" in which job. The table is strongly connected to *host mapping* section in Fitcrack WebAdmin. Every time a user assings a host to a job, a new record in this table is created. The structure of the table is defined as follows:

- **id** - primary key,
- **boinc_host_id** - host ID in BOINC `host` table,
- **job_id** - ID of the job, the host is mapped to.

7.10 fc_host_status

The table is used for displaying online/offline status of hosts in Fitcrack WebAdmin. Each record represents a state of the host (cracking node). The structure of the table is defined as follows:

- **id** - primary key,
- **boinc_host_id** - host ID in BOINC `host` table,
- **last_seen** - time the host was last seen online,
- **deleted** - flag (0/1) representing if the host is hidden from WebAdmin, or not.

7.11 fc_job

The table is used to store cracking *jobs*. Each record represents a cracking job with a defined *attack mode*, *submode*. Within a job, we have one or more *hashes* (stored in `fc_hash` table) of the same *hash type*. The structure of the table is defined as follows:

- **id** - primary key,
- **token** - unique identifier of a user WebAdmin session within which the job was created. It is calculated as a SHA-256 hash of current timestamp concatenated with user's IP address and job ID.
- **attack_mode** - attack mode (see section 3), a value from table 9,
- **attack_submode** - attack submode (see section 3), a value from table 10,
- **hash_type** - number of hashcat's hash type,
- **status** - job status code, a value from table 9,
- **keyspace** - the real number of candidate passwords,
- **hc_keyspace** - hashcat's normalized keyspace (see section 2.3),
- **indexes_verified** - nuber of processed hashcat-indexes form keyspace,
- **current_index** - keyspace index from which the next workunit will start, a number in range $0..(hc_keyspace - 1)$. In combination attack, it is the offset in the second dictionary,
- **current_index_2** - offset in the first dictionary (for combination attack),
- **time** - time the job was added to the database,
- **name** - name of the job,
- **comment** - optional user comment,
- **time_start** - timestamp defining when the job should start,
- **time_end** - timestamp defining then the job has to end,
- **cracking_time** - total sum of host cracking times,
- **seconds_per_workunit** - time period for a workunit used in the adaptive scheduling algorithm (see 2.4),
- **config** - WebAdmin-filled part of workunit config (see 6),
- **dict1** - password dictionary no. 1,
- **dict2** - password dictionary no. 2,
- **charset1** - user-defined character set no. 1 in hex form,
- **charset2** - user-defined character set no. 2 in hex form,
- **charset3** - user-defined character set no. 3 in hex form,
- **charset4** - user-defined character set no. 4 in hex form,
- **rules** - name of the file with password-mangling rules (for attack 01), or NULL (for others),
- **rule_left** - left password-mangling rule (for attacks 11, 13),
- **rule_right** - right password-mangling rule (for attacks 11, 13),
- **markov_hcstat** - name of the Markov `.hcstat2` file,
- **markov_threshold** - threshold liming the number of states processed within the Markov model (0 = no limit aka "full brute-force"),
- **replicate_factor** - says how many hosts should work on a single workunit (1 = no replication),
- **deleted** - flag (0/1) defining if the job was hidden, or not.

mode	submode	description
0	0	Basic dictionary attack
0	1	Dictionary attack with *password-mangling rules*
1	0	Basic combination attack
1	1	Combination attack with *left rule*
1	2	Combination attack with *right rule*
1	3	Combination attack with *left* and *right rule*
3	0	Basic brute-force attack
3	1	Brute-force attack with custom hcstat file using 2D Markov
3	2	Brute-force attack with custom hcstat file using 3D Markov

Table 9. Attack *modes* and *submodes* in Fitcrack

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 due to exceeded `time_end`.
10	running	Computation is in progress.
11	validating	Validating hashes. (not used)
12	finishing	All keyspace assigned, some hosts still compute.

Table 10. Job *status codes* in Fitcrack

7.12 fc_job_dictionary

This allows Fitcrack to use multiple dictionaries within a *dictionary* or a *combination* attack. *Generator* subsystem (see section 4.10) loads all dictionaries which are not processed yet (`current_index` $\neq$ `keyspace` in `fc_dictionary`), and continuously creates fragments, as described in section 3. The structure of the table is defined as follows:

- **id** - primary key,
- **job_id** - job ID in `fc_jobs` table,
- **dictionary_id** - dictionary ID in `fc_dictionary` table,
- **current_index** - current index in a dictionary,
- **is_left** - flag (0/1) defining it is a *left dictionary* in a *combination* attack.

7.13 fc_job_graph

The table is used for displaying progress graph in Fitcrack *WebAdmin* (see section 4.2). Each record represents a point in the graph. The structure is defined as follows:

- **id** - primary key,
- **progress** - job progress as a double between 0 and 1,
- **job_id** - job ID from `fc_job` table,
- **time** - time the point was added to the graph.

7.14 fc_mask

The table stores *password masks* which are used in a *brute-force* attack and *combination attacks*, as described in section 3. Each record represents a single password mask. The structure of the table is defined as follows:

- **id** - primary key,
- **job_id** - job ID from `fc_job` table,
- **mask** - the password mask,
- **current_index** - keyspace index from which the next workunit will start, a number in range $0..(hc_keyspace - 1)$.
- **keyspace** - the real number of password candidates generated from the mask,
- **hc_keyspace** - hashcat's keyspace of the mask.

7.15 fc_masks_set

Since the *WebAdmin* allows to import and export set of masks in the form of text files with `.hcmask` extension, it is necessary to store information about mask files present in the system. Each record corresponds to a single mask file located in `COLLECTIONS_ROOT/masks` directory. The structure of the table is defined as follows:

- **id** - primary key,
- **name** - name of the mask set (displayed in WebAdmin),
- **path** - the real name of the mask set file,
- **time** - time the mask set was added to the system,
- **deleted** - flag (0/1) saying if the mask set file was deleted.

7.16 fc_notification

To inform the user about important events (e.g. cracking job is finished, etc.), Fitcrack uses a system of notifications which are display in *WebAdmin*. Each record in the table represents a single notification. The structure of the table is defined as follows:

- **id** - primary key,
- **user_id** - ID of a recipient (in `fc_user` table) of the noficitaion,
- **source_type** - type of the source (0 = job, others not used yet),
- **source_id** - ID of the source, e.g. job ID from `fc_job`,
- **old_value** - original value (for notifications about value change),
- **new_value** - new value (for notifications about value change),
- **seen** - flag (0/1) saying if the recipient has seen the notification,
- **time** - time the notification was created in the system.

7.17 fc_protected_file

Since Fitcrack *WebAdmin* also supports password-protected files (e.g. encrypted containers) as an input, it is necessary to store them in the database before the files are processed by *XtoHashcat* tool (see section 4.8). Each record in the table represents a single protected file. The structure of the table is defined as follows:

- **id** - primary key,
- **name** - name of the protected file (displayed in WebAdmin),
- **path** - the real name of the protected file,
- **hash** - hash exported from the file by *XtoHashcat*,
- **hash_type** - hashcat's number defining the type of hash,
- **time** - time the protected file was added to the system.

7.18 fc_role

Each user of Fitcrack *WebAdmin* is assigned a *role*. The role defines user's permissions - what the used is allowed to do. Each record represents a role defined by ID, name and a list of permission flags. The structure of the table is defined as follows:

- **id** - primary key,
- **name** - name of the role (e.g. administrator),
- **MANAGE_USERS** - flag (0/1) allowing the user to manage users and roles,

- **ADD_NEW_JOB** - flag (0/1) allowing to add new jobs,
- **UPLOAD_DICTIONARIES** - flag (0/1) allowing the user to add new password dictionaries to the system,
- **VIEW_ALL_JOBS** - flag (0/1) allowing the user to view all jobs,
- **EDIT_ALL_JOBS** - flag (0/1) allowing the user to edit all jobs,
- **OPERATE_ALL_JOBS** - flag (0/1) allowing the user to operate (start, stop, restart, etc.) all jobs,
- **ADD_USER_PERMISSIONS_TO_JOB** - flag (0/1) allowing the user to add job-specific permissions (NOTE: *not implemented yet*!).

7.19 fc_rule

In *dictionary* attack (see 3.1), the user can select a *ruleset*. Each ruleset has the form of file containing a list of *password-mangling rules*. On client-side the rules are applied to all candidate passwords. Each record in the table defines a single file with password-mangling rules. The structure of the table is defined as follows:

- **id** - primary key,
- **name** - name of the ruleset (displayed in WebAdmin),
- **path** - the real name of the ruleset file,
- **time** - time the ruleset was added to the system,
- **deleted** - flag (0/1) saying if the ruleset was deleted.

7.20 fc_settings

This table contains global settings of the Fitcrack server. The settings define the behavior for creating and handling cracking jobs, as well as default values of various job parameters. The table has only one record. The structure of the table is defined as follows:

- **id** - primary key,
- **delete_finished_workunits** - flag (0/1, default: 0) defining if the system should delete (1) or preserve (0) finished workunits,
- **default_seconds_per_workunit** - number of seconds (default: 3600) specifying the default value for `seconds_per_workunit` job parameter,
- **default_replicate_factor** - the default value for `replicate_factor` job parameter,
- **default_verify_hash_format** - flag (0/1, default: 1) defining if WebAdmin should verify the format of input hashes using *HashValidator* tool (see section 4.6),
- **default_check_hashcache** - flag (0/1, default: 1) defining if the system should search for each has in the already cracked hashes before it starts cracking,
- **default_workunit_timeout_factor** - the number (default: 1) multiplying the workunit timeout - the time after a workunit is considered failed (and re-assignedú, if no result is received.
- **default_bench_all** - flag (0/1, default: 1) - defining if the newly-connected hosts should perform the complete benchmark, as described in section 6.

7.21 fc_user

This table is used to store accounts of users who have access to Fitcrack WebAdmin. Each record represents a single user account with a given username, e-mial, password, and role. The structure of the table is defined as follows:

- **id** - primary key,
- **username** - name of the user,
- **password** - hash of user password created using PBKDF2 with 50000 iterations of SHA-256 algorithm,
- **mail** - e-mail address of the user,
- **role_id** - ID of user's role (from `fc_role` table),
- **deleted** - flag (0/1) saying if the user was deleted.

7.22 fc_user_permissions

The table is designed to store per-job (non-global) user permisions, however this feature has not been implemented yet. Each record represents a permission of a user to do specific operation with a specific job. The structure of the table is defined as follows:

- **id** - primary key,
- **job_id** - ID of the job (from `fc_job`) which is operated,
- **user_id** - ID of the user (from `fc_user`),
- **view** - flag (0/1) allowing the user to view the job,
- **modify** - flag (0/1) allowing the user to modify the job,
- **operate** - flag (0/1) allowing the user to operate the job,

7.23 fc_workunit

In Fitcrack, a *workunit* is a single piece of cracking work. Workunits are created continuously by the *Generator* daemon. Every workunit belongs to a job from which the workunit was created. Every record in `fc_workunit` table is connected to an existing record in BOINC `workunit` table. Each record represents a workunit created within a given job. The structure of the table is defined as follows:

- **id** - primary key,
- **job_id** - ID of a job in `fc_job` table,
- **workunit_id** - ID of a record in BOINC `workunit` table,
- **host_id** - ID of a host to which the workunit is assigned,
- **boinc_host_id** - BOINC ID of the host,
- **start_index** - starting password index from the job keyspace (i_{min} value from section 2.3); defines where the computation starts; in *combination attack*, the value represents the offset in the second dictionary,
- **start_index_2** - offset for the first dictionary in *combination attack* (see section 3.2),

- **hc_keyspace** - hashcat's keyspace of the workunit,
- **progress** - host's progress on the workunit (value between 0 and 1),
- **mask_id** - ID of a mask, if used,
- **dictionary_id** - ID of a dictionary, if used,
- **duplicated** - flag (0/1) defining if the job was split to multiple ones due to exceeded timeout (currently not ued),
- **duplicate** - if `duplicated`=1, contains the ID of the original workunit,
- **time** - time the workunit was added to the database,
- **cracking_time** - time spend by the host by computing the workunit,
- **retry** - flag (0/1) defining if it is a re-assigned workunit,
- **finished** - flag (0/1) defining if the workunit was completed.

8 Conclusion

Fitcrack is a software system for distributed password cracking, also referred to as password recovery. It uses BOINC as a framework for task-distribution, and hashcat as the client-side cracking engine. The technical report described the basic principles of hash cracking, task distribution, and most importantly - described the design of the proposed system.

All other relevant information is located on Fitcrack website[37]. The most recent version of Fitcrack is located on NES@FIT GitHub page[38]

References

[1] Adobe Systems Incorporated. *Document management — Portable document format — Part 1: PDF 1.7*. 32000-1:2008. Geneva, Switzerland: ISO, July 2008.

[2] D. P. Anderson. “BOINC: a system for public-resource computing and storage”. In: *Grid Computing, 2004. Proceedings. Fifth IEEE/ACM International Workshop on*. Nov. 2004, pp. 4–10.

[3] Corel Corporation. *AES Encryption Information: Encryption Specification AE-1 an AE-2*. Version 1.04. Jan. 2009.

[4] Peter Gazdík. “Use of Heuristics for Password Recovery with GPU Acceleration”. Czech. Bachelor’s thesis. Brno, CZ: Faculty of Information Technology, Brno University of Technology, 2015. URL: http://www.fit.vutbr.cz/study/DP/BP.php?id=18210.

[5] Radek Hranický, Martin Holkovič, Petr Matoušek, and Ondřej Ryšavý. “On Efficiency of Distributed Password Recovery”. In: *The Journal of Digital Forensics, Security and Law* 11.2 (2016), pp. 79–96. ISSN: 1558-7215. URL: http://www.fit.vutbr.cz/research/view_pub.php.cs?id=11276.

[6] Radek Hranický, Petr Matoušek, Ondřej Ryšavý, and Vladimír Veselý. “Experimental Evaluation of Password Recovery in Encrypted Documents”. In: *Proceedings of ICISSP 2016*. Roma, IT: SciTePress - Science and Technology Publications, 2016, pp. 299–306. ISBN: 978-989-758-167-0. URL: http://www.fit.vutbr.cz/research/view_pub.php.cs?id=11052.

[7] Radek Hranický, Lukáš Zobal, Vojtěch Večeřa, and Petr Matoušek. “Distributed Password Cracking in a Hybrid Environment”. In: *Proceedings of SPI 2017*. Brno, CZ: University of defence in Brno, 2017, pp. 75–90. ISBN: 978-80-7231-414-0. URL: http://www.fit.vutbr.cz/research/view_pub.php?id=11358.

[8] Ryan Lim. “Parallelization of John the Ripper (JtR) using MPI”. In: *Nebraska: University of Nebraska* (2004).

[37] https://fitcrack.fit.vutbr.cz/

[38] https://github.com/nesfit/fitcrack

[9] Dávid Mikuš. "Password Recovery of RAR, BZIP, and GZIP Archives Using GPU". Czech. Bachelor's thesis. Brno, CZ: Faculty of Information Technology, Brno University of Technology, 2015. URL: http://www.fit.vutbr.cz/study/DP/BP.php?id=18740.

[10] Arvind Narayanan and Vitaly Shmatikov. "Fast Dictionary Attacks on Passwords Using Time-space Tradeoff". In: *Proceedings of the 12th ACM Conference on Computer and Communications Security*. CCS '05. Alexandria, VA, USA: ACM, 2005, pp. 364–372. ISBN: 1-59593-226-7. DOI: 10.1145/1102120.1102168. URL: http://doi.acm.org/10.1145/1102120.1102168.

[11] Andy Pippin, Brent Hall, and Wilson Chen. *Parallelization of John the Ripper Using MPI (Final Report)*. Tech. rep. 2006.

[12] Niels Provos and David Mazieres. "A Future-Adaptable Password Scheme." In: *USENIX Annual Technical Conference, FREENIX Track*. 1999, pp. 81–91.

[13] *RAR file format*. [Online; accessed 2017-01-03]. URL: http://acritum.com/winrar/rar-format.

[14] R. Rivest. *The MD5 Message-Digest Algorithm*. Tech. rep. 1321. Updated by RFC 6151. Apr. 1992. URL: http://www.ietf.org/rfc/rfc1321.txt.

[15] V. L. Thing and H.-M. Ying. "Making a faster cryptanalytic time-memory trade-off". In: *Advances in Cryptology* (2003), pp. 617–630.

[16] Vojtěch Večeřa. "Password Recovery of ZIP Archives Using GPU". Czech. Bachelor's thesis. Brno, CZ: Faculty of Information Technology, Brno University of Technology, 2015. URL: http://www.fit.vutbr.cz/study/DP/BP.php?id=18211.

[17] X. Wu, J. Hong, and Y. Zhang. "Analysis of OpenXML-based office encryption mechanism". In: *2012 7th International Conference on Computer Science Education (ICCSE)*. July 2012, pp. 521–524. DOI: 10.1109/ICCSE.2012.6295128.

[18] Lukáš Zobal. "Microsoft Office Password Recovery Using GPU". Czech. Bachelor's thesis. Brno, CZ: Faculty of Information Technology, Brno University of Technology, 2015. URL: http://www.fit.vutbr.cz/study/DP/BP.php?id=18341.