The architecture of Fitcrack

distributed password cracking system, ver. 2

Technical report

Radek Hranický, Lukáš Zobal, Vojtěch Večeřa, Matúš Múčka, Adam Horák, Dávid Bolvanský, Tomáš Ženčák

Technical report n. FIT-TR-2020-05

Faculty of Information Technology,
Brno University of Technology

Last modified: February 24, 2021

Table of Contents

The architecture of Fitcrack distributed password cracking system, version 2

Radek Hranický, Lukáš Zobal, Vojtěch Večeřa, Matúš Múčka, Adam Horák, Dávid Bolvanský, Tomáš Ženčák

Brno University of Technology, email:
{ihranicky,izobal,vecerav,muckamat}@fit.vutbr.cz,
{xhorak69,xbolva00,xzenca00}@stud.fit.vutbr.cz

Abstract. This updated technical report describes the architecture of Fitcrack distributed password cracking system developed within the *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. In addition to the original report, this updated version contains the description of new modules and server daemons, PRINCE attack, PCFG attack, and other enhancements.

1 Introduction

Fitcrack system 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 [7]. 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 [7].

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

[1] https://boinc.berkeley.edu/
[2] https://www.khronos.org/opencl/

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 for all formats except 7z and RAR. Some of the GPU kernels were adopted from our single-machine tool *Wrathion* [10].

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.

The report documents the hashcat-based version of Fitcrack, its architcture, the distribution of cracking tasks, and the implementation of various hashcat-compatible attack types. In addition to the original report [14], this updated version documents:

- an improved adaptive scheduling algorithm (See Section 2.4.),
- the support for PRINCE attack (See Section 3.6.),
- the support for PCFG attack (See Section 3.5.),
- updated webadmin application with a detailed description of individual REST API endpoints (See Section 4.2.),
- updates to existing server daemons (See Section 4.),
- new server daemons: PCFG Monitor, Manager, and sysUsage,
- enhanced Runner with improved benchmarking and a new global mutex that provides native support for pipeline workunit processing (See Section 5.3.),
- an updated database scheme (See Section 7.).

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:

[3] https://developer.nvidia.com/cuda-zone
[4] https://hashcat.net/
[5] https://contest.korelogic.com/
[6] http://www.openwall.com/john/

  - *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.
- **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

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

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 [21] is about 31 GH/s, however the cracking speed of Bcrypt [19] 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 [26]. 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* [22]. 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.

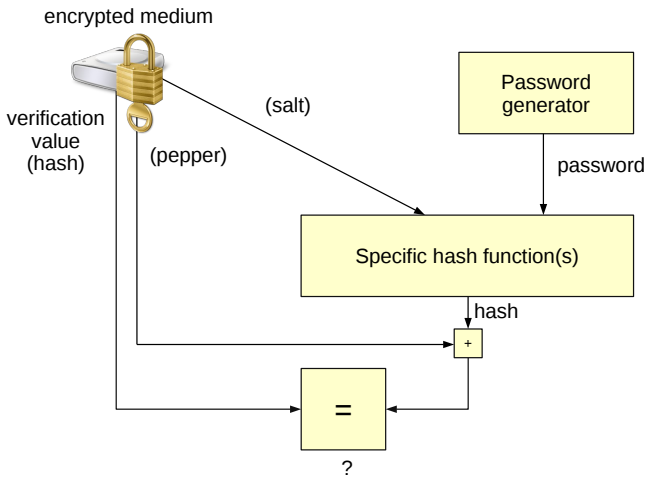

Fig. 1. Password cracking of encrypted media

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 [13].

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.

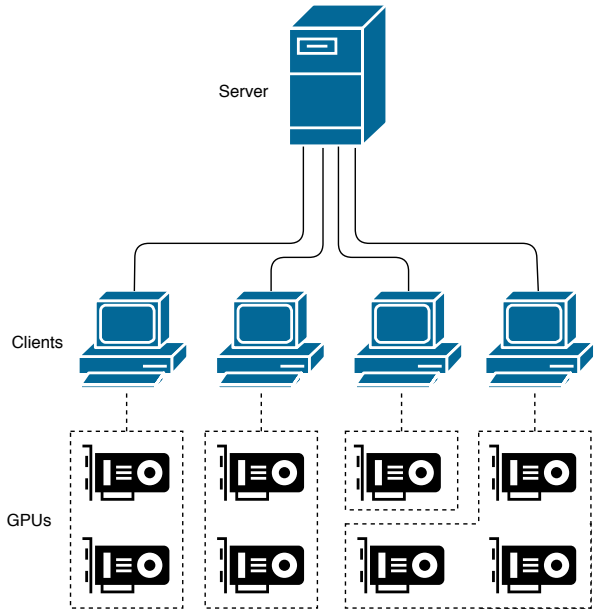

Fig. 2. An example of a cracking network

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 mode (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. [18] 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 [15] 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.

2.3.1 Index-based distribution

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) [7], 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.

2.3.2 Jobs and workunits

As mentioned above, a *job* in Fitcrack is a single cracking assignment. It has a defined hash algorithm, attack mode (See Section 3.), and configuration options (See Section 6). Each job in Fitcrack goes through a series of states. All possible states are enlisted in Table 13, while each has a unique numeric identifier from 0 to 12. Numbers above 10 mean the job is not running. Historically, not all

are numbers are used; some are reserved for future use. The lifetime of a job is illustrated in Figure 20. Three subsystems of Fitcrack are allowed to change the state of the job: the Generator, when a host asks for a new workunit, the Assimilator if a workunit result is received, and the Webadmin at an event of user's action. Every job starts in the *ready* state, created by a user, and added to the database by the WebAdmin backend. Once the user launches it, the job switches to the *running* state. All hosts assigned to a job also have status codes defining the stage of their participation. The host codes are shown in Table 14.

Each job consist of one or more workunits. A *workunit* is a piece of work assigned to a client. In Fitcrack, the creation of workunits is handled by the *Generator* module (see section 4.13) 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. For some attacks, the range of indexes defined above can be set directly by hashcat's `--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 $i_{max} - i_{min}$. For dictionary-based attacks or attacks that use an external password generator, Fitcrack uses different distribution strategies (See Section 3.). However, the general principle is the same, and with some abstraction, we can map every workunit to a range of password indexes.

2.3.3 The keyspace in hashcat

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 a 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 a distribution strategy that maximizes efficiency of the computing process. It means that the higher-performance clients would receive larger workunits than the lower-performance clients

Therefore, Fitcrack uses of *targeted* workunits. Whenever a host asks for work, it receives a workunit with keyspace calculated to fit its current performance. The size of the workunit is chosen with respect to the desired processing time.

More formally, let $P_R \subseteq P$ be the set of all remaining password candidates that need to be verified. Next, let t_p be the desired workunit processing time in seconds. Finally, let v_i be the current performance (cracking speed) of node i in passwords per second. Then, the size s_i of a new workunit assigned to node i is calculated as $s_i = min(t_p \cdot v_i, |P_R|)$. Speed v_i is determined from previously solved workunit as $v_i = \frac{s_{prev}}{t_{prev}}$ where s_{prev} is the size of a previous workunit assigned to the node, and t_{prev} is the time spent by its processing. Choosing v_i for a newly connected client is performed by running a special *benchmark* workunit on the client to calculate its performance. The question is how to choose the workunit processing time t_p. This critical variable affects the distribution's granularity. By specifying t_p, we define how long we want a host to process a workunit. Concretely:

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

Since the initial benchmark is often inaccurate, Fitcrack supports the *ramp-up* technique that creates smaller workunits at the beginning of each job. The technique also serves as a fail-safe mechanism to minimize the implications of unexpected host behavior. Potential failures range from GPU overheating, lack

of memory, through network problems up to possibly compromised nodes. This "suspicious" property of the algorithm does not let the host process full-sized workunits before it proves the ability to resolve smaller ones. Another problem may occur at the end of the job. Once all the remaining keyspace is distributed, some nodes may compute while the others are done and meaninglessly wait for the others. The problem is resolved by an optional *ramp-down* technique that creates progressively smaller workunits at the end of the job.

When a user creates a new job in Fitcrack, they specify the *seconds per workunit* value. The adaptive scheduling algorithm, however, applies the principles of ramp-up and ramp-down described above. Moreover, the user may specify how rapidly the techniques should affect the job, including options to disable both ramp-up and ramp-down completely.

To effiently calculate t_p, we define function $proctime(t_J, |P_R|, k)$. Parameter t_J is an elapsed time of the computing, parameter $|P_R|$ is the number of remaining passwords to be verified and k is a number of active hosts that participate on the computing. Parameters 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.

Algorithm 1: Adaptive calculation of t_p

Input: t_J, s_R, k
Output: t_p

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

Lines 2 to 5 of the algorithm compute the entire speed of all active nodes. The following lines are a bit tricky. Normally, we would have calculated t_p as

$t_p = \frac{|P_R|}{v_{sum}}$. Here, we add parameters α and and β, both ranging from 0 to 1. Parameter α is called *distribution coefficient* and defines the maximum fraction of the remaining keyspace that can be assigned in a single workunit. E.g., for $\alpha = 0.1$ means that maximally 10% of the remaining keyspace $|P_R|$ is assigned. The goal is to make sure there is enough keyspace left for potential new nodes. Next, β is the *ramp-down coefficient*. It ensures that the workunit size will be at least $t_{pmax} \cdot \beta$ to avoid excessively small workunits at the end of the job. Increasing β reduces the ramp-down effect. Setinng β to 1 disables the ramp-down completely, while 0 means full ramp-down where the workunit size is only limited by t_{pmin}.

The 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_{pmax} as the *seconds per workunit* option.

Finally, the boolean *rampup* variable defines if the ramp-up is on or off. If enabled, it is performed in an initial stage of the job. This stage is defined as the period before the elapsed time reaches the desired workunit time. For example, the desired time for a single workunit is 15 minutes. But the full-size workunits are not created in the first 15 minutes of the job. The algorithm reserves this time for stabilization to withstand any benchmark inaccuracies or unexpected host reactions. All the parameters, including α, β, *rampup*, and t_{pmin} can be customized in Fitcrack advanced settings through the WebAdmin interface.

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 [7]. Each side consists of multiple subsystems which will be defined in the following sections.

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

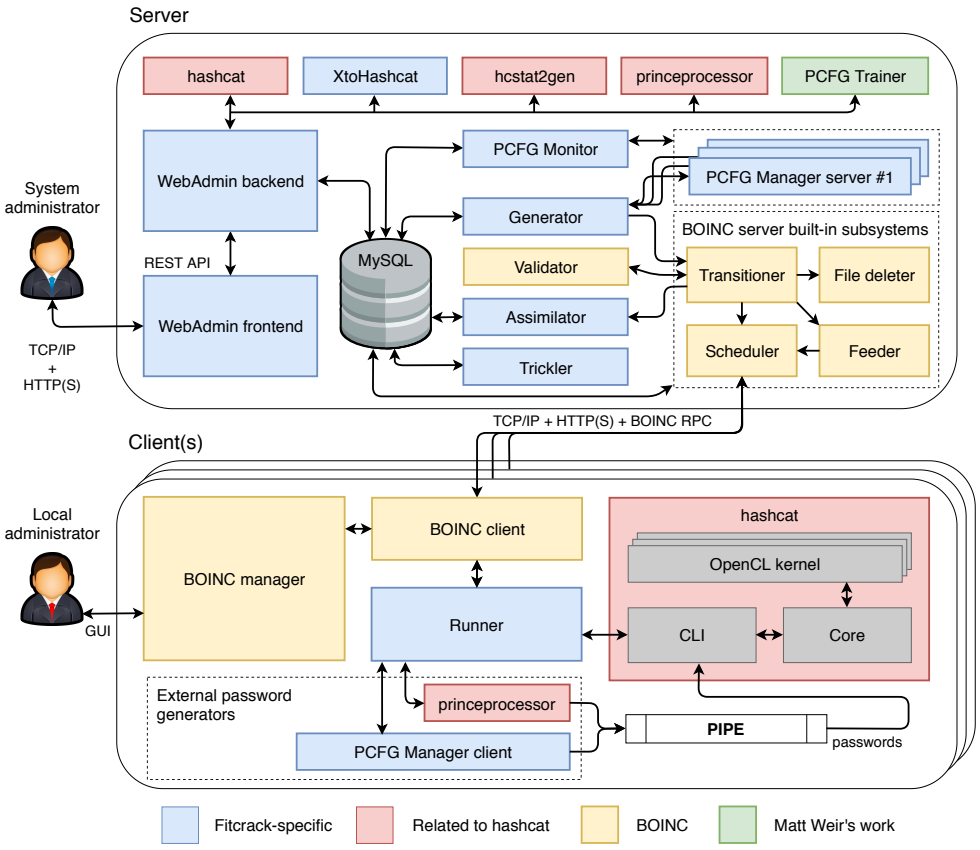

Fig. 3. The architecture of Fitcrack

3 Attack modes

As a cracking engine, Fitcrack uses *hashcat* version 5.1.0. 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), *hybrid* attacks (6 and 7), *PRINCE* attack (8) and *PCFG* attack (9). 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.

The numbering of modes and submodes is described by table 1.

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
6	0	Hybrid attack: wordlist + mask
6	1	Hybrid attack: wordlist + *left rule* + mask
7	0	Hybrid attack: mask + wordlist
7	2	Hybrid attack: mask + wordlist + *right rule*
8	0	PRINCE attack
8	1	PRINCE attack with *password-mangling rules*
9	0	PCFG attack
9	1	PCFG attack with *password-mangling rules*

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.
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, 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.

[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

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

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 [13].

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.

3.2 Combination attack

A *combination attack*, also referred to as a *combinator attack*, uses two separate password dictionaries: a *left* dictionary, and a *right* dictionary. Candidate passwords are crafted using a string concatenation: passwords from the left dictionary are extended by passwords from the right one. The goal is to verify combinations of all passwords in the two input dictionaries. An example of a combination attack is shown in figure 5.

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

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

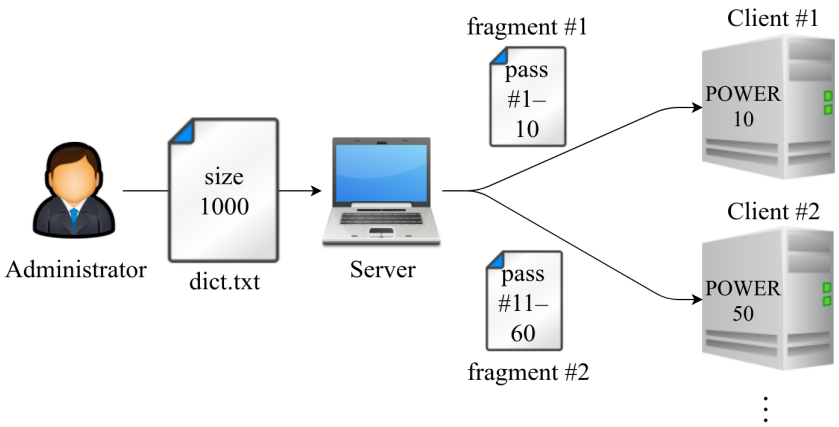

Fig. 4. Example of dictionary attack distribution

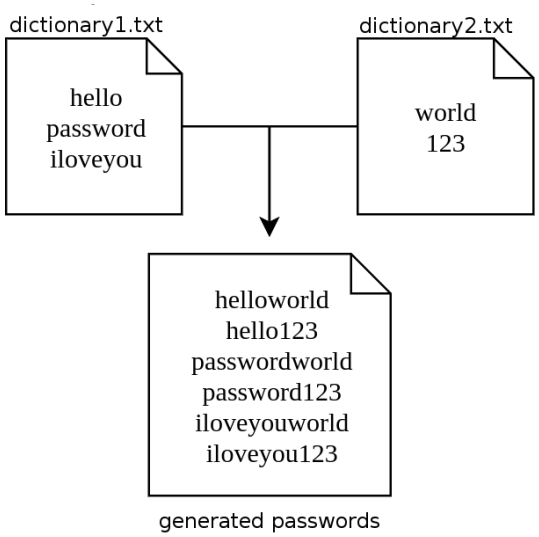

Fig. 5. Illustration of combination attack

When dealing with hashcat, we realized its keyspace computation doesn't consider the second dictionary. When 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 `--skip` and `--limit` mechanism. Also, we keep the linear complexity of the whole attack. You can see a scheme of such an attack in Figure 6.

Algorithm 2: Limiting the combination attack to the desired keyspace

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

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

Algorithm 2 describes the process of calculating workunit size. In a nutshell, it decides how many passwords from the right dictionary to send and whether to use the `--skip` and `--limit` parameters. The algorithm uses five inputs: $k_{desired}$

is the desired keyspace of the workunit, k_{left} and k_{right} is the keyspace of the left and the right dictionary, respectively. To indicate the current position in both dictionaries, it uses two indexes: i_{left} and i_{right}. The condition o line 10 is a heuristic that adds some tolerance to prevent over-fragmenting. For example, if the desired keyspace is 100 and there are 101 passwords remaining, it will assign all 101 instead of creating two workunits with 100 and 1 password. Essentially, there are three possible situations:

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

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

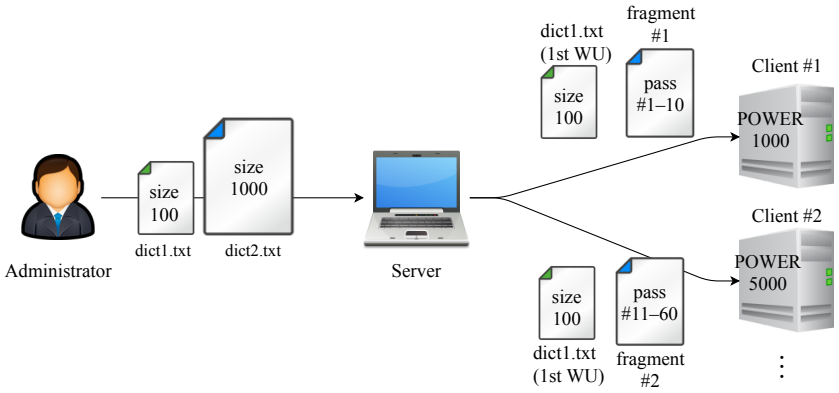

Fig. 6. Example of combinator attack distribution

An example of the proposed strategy is illustrated in Figure 6. In this example, we have two dictionaries, the left dictionary contains 100 passwords, while the right one contains 1000 passwords. The first one is distributed to all nodes, while for the second one, Fitcrack uses fragmentation. Client #1 can verify 1000 passwords per time unit, and thus with the first chunk, the client receives the first 10 passwords from the second dictionary since $10 \cdot 100 = 1000$. Client #2 has five times higher performance and can verify 500 passwords per time unit. Therefore, it receives the following 50 passwords since $50 \cdot 100 = 5000$. In this

case, everything was solved by fragmenting just the right dictionary. The `--skip` and `--limit` parameters were not necessary.

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

3.3 Brute-force attack

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

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

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

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

Table 3. The substitute symbols and corresponding character sets

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 illistrated by Figure 7. If there are concrete characters in a mask, the same characters at the same positions are used in the generated candidate passwords – i.e., if for all $i \in [1, n]$, if $s_i = c_i$, character c_i is used at the i-th position in all generated passwords. For substitute symbols, all possible characters from corresponding character sets are eventually used. If there is more than one one substitute symbol, candidate passwords are generated as a cartesian product of all used corresponding character sets.

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

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

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.

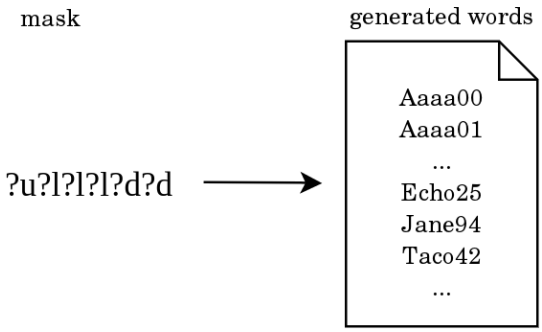

Fig. 7. Illustration of a brute-force mask attack

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* [17, 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.

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 one is `e`, etc.

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

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

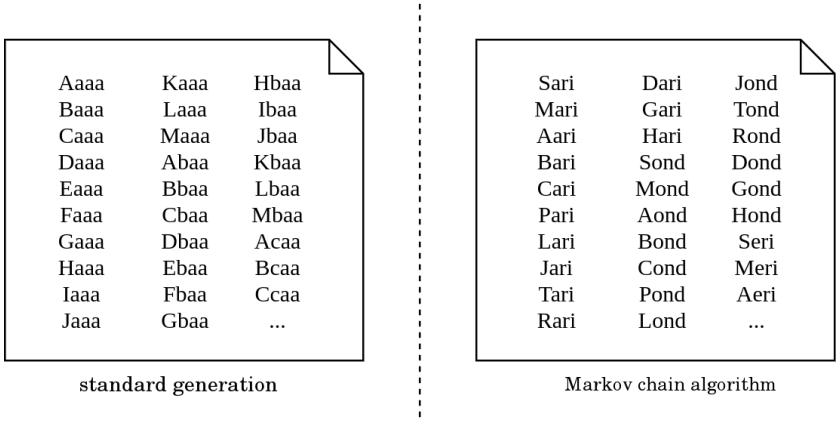

Fig. 8. Candidate password order using Markov chains

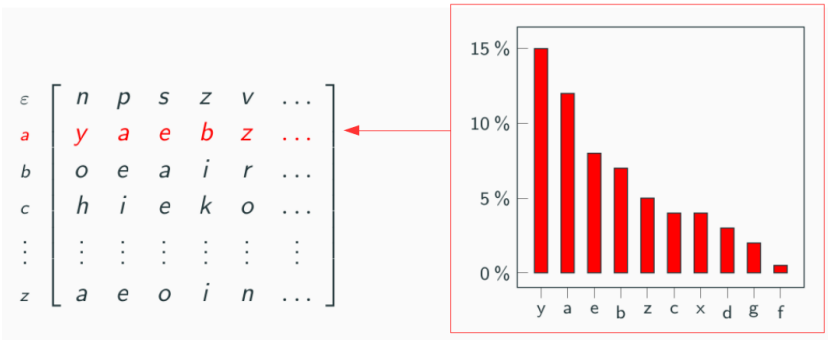

Fig. 9. Markov chain probability matrix

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.

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

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.

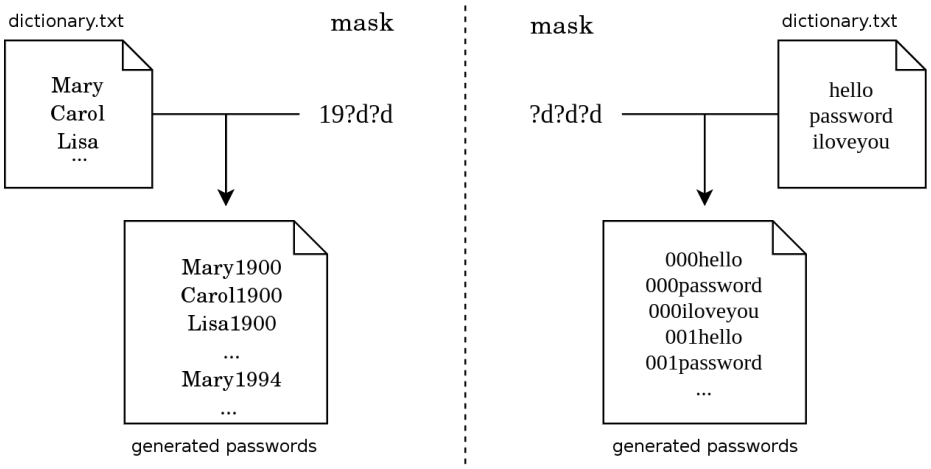

Fig. 11. The principle of hybrid attacks

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 keyspace of the left part only. When instructed to verify one password, hashcat, in fact, checks the combination of one string from the left side and all strings from the right side.

The early hashcat-based versions of Fitcrack used a compromise. With the high-performance *maskprocessor*[16] utility, the server created a dictionary of all possible strings from the given masks. Then, the attack was transformed into a combination attack, and the distribution followed the same strategy as proposed in Section 3.2. With the two dictionaries, it was possible to control the size of workunits relatively precisely. An advantage of the solution was that Fitcrack supported the use of left and right password-mangling rules (see Section 3.1.1.) in the same way as with the combination attack. The solution worked well with smaller masks. However, with complex high-keyspace masks, the initial overhead and the space requirements were not acceptable.

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

- For the **hybrid mask + wordlist** attack, the dictionary on the right side is fragmented in the same manner as in the combination attack. If necessary, the mask is limited using the –skip and –limit parameters using Algorithm 2. This is entirely safe since, for the hybrid attack mode, hashcat does not use the optimization described in Section 3.3. The limit parameter thus precisely specifies the exact number of strings generated from the mask. An example of the workunit distribution is shown in Figure 12.
- For the **hybrid wordlist + mask**, the mask on the right is transformed into multiple masks with lower keyspace using the newly-proposed Algorithm 4. To allow precise control of the keyspace, it uses creates custom character sets on-the-fly using the new `GetCharsetSlice()` function defined by Algorithm 3. If necessary, the dictionary on the left is limited using the –skip and –limit parameters using Algorithm 2. An example of workunit distribution with mask slicing is shown in Figure 12.

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

Algorithm 3: GetCharsetSlice() function for limiting character sets

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

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

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

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

```
1: adjustedStartIndex = startIndex
2: resultKeyspace = 1
3: maskSlice = []
4: forall symbol ∈ mask do
5:     if ¬IsCharset(symbol) then
6:         continue
7:     charset = symbol
8:     charIndex = adjustedStartIndex  mod |charset|
9:     adjustedStartIndex /= |charset|
10:    if charIndex > 0 then
11:        charset = charset[charIndex :] ;        // forced split here
12:    if desiredKeyspace ≤ resultKeyspace then
13:        maskSlice += charset[0] ;     // Desired keyspace reached,
               add single char
14:        continue
15:    remainingKeyspace = desiredKeyspace/resultKeyspace
16:    if charIndex == 0 && remainingKeyspace ≥ |charset| then
17:        maskSlice += symbol
18:        resultKeyspace *= |charset|
19:    else
20:        maskSlice += GetCharsetSlice(remainingKeyspace, charset)
21:        desiredKeyspace = resultKeyspace ; // Only added piece of
               charset do not add any more
22: return maskSlice
```

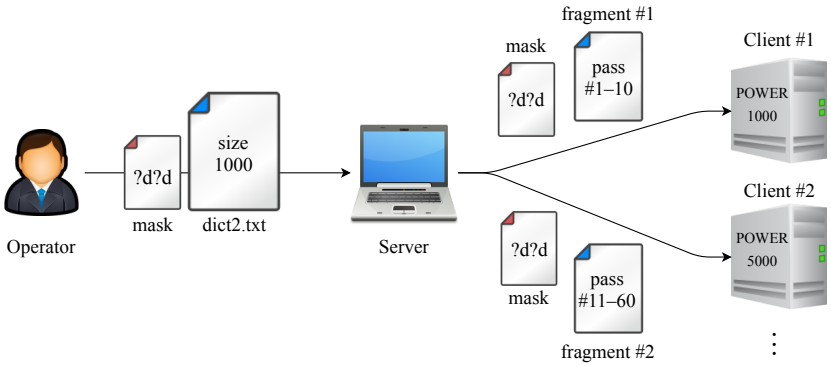

Fig. 12. Example of hybrid (mask+wordlist) attack distribution

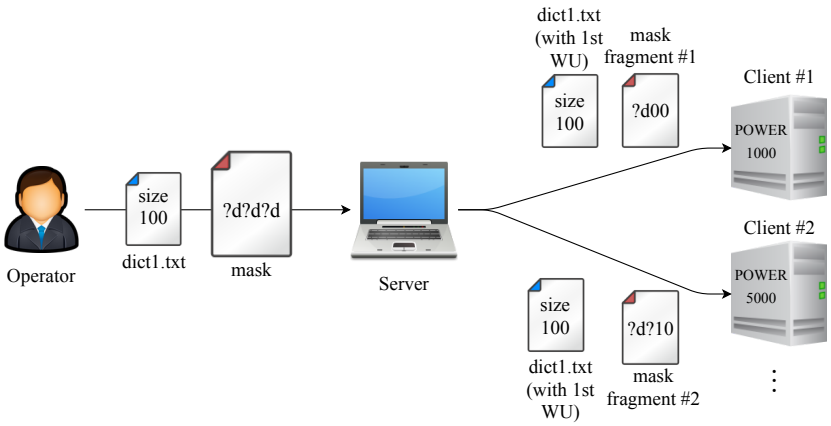

Fig. 13. Example of hybrid (wordlist+mask) attack distribution

The example from Figure 12 is relatively simple. The mask produces 100 different strings, and the dictionary has 1000 passwords. The first host needs 1000 passwords. Therefore, it receives a workunit with the mask ?d?d without limiting, and the first ten passwords from the right dictionary: $100 \times 10 = 1000$. The second host needs 5000 passwords, so it gets the same unlimited mask and 50 dictionary passwords: $100 \times 50 = 5000$.

The hybrid wordlist + mask attacks, on the other hand, use the mask slicing. The principle is illustrated using two examples. Consider a hybrid attack with mask `?d?h?l` on the right side. Suppose that we want to send a mask with 20 passwords. The first substitute symbol C_1 is `?d` with 10 different possibilities for the first character: $|C_1| = 10$. The desired keyspace is two times higher, so we leave the first symbol intact and proceed to the second one. The second symbol is C_2 = `?h`, which stands for the character set containing `23456789abcdef`. Thus $|C_2| = 16$ options. Nevertheless, the current keyspace is 10, and to get 20, we only want to multiply the keyspace by 2. Therefore, we take the first two characters from `?h` and create a custom character set `?1` containing `01`. After that, we reached the desired keyspace of 20. Hence, no more multiplication is needed. Therefore, from the next substitute symbol C_3 = `?l` we take only the first character `a`. The slicing is completed, and we send the mask `?d?1a`.

For the next workunit, suppose that we want a mask with keyspace 150. We already processed 20 passwords from the mask because we used the first substitute symbol `?d` with keyspace 10 with the first two characters of `?h`. From the `?h`, 14 characters are remaining. To avoid unnecessary complexity, the heuristic from algorithm 3 forces Fitcrack to finish the fragmented character set first. Therefore, the resulting mask is `?d?1a` with the custom character set `?1` containing `23456789abcdef`. The workunit has a keyspace of 140, which is as close to 150 as allowed in the current case.

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

The second example follows the situation in Figure 13. Both the dictionary and mask have a keyspace of 100. In this case, there are two options: either fragmenting the left dictionary or slicing the mask. The illustration shows the slicing option. The first host needs a workunit of 1000 passwords. Therefore, it receives the entire dictionary of 100 words and the slice `?d00` of the mask that produces ten strings: $100 \times 10 = 1000$. The second host needs 5000 passwords. The sliced mask is `?d?10` where `?1` is the custom character set containing `12345`: $100 \times 50 = 5000$.

3.5 PCFG attack

The attack is based on the previous knowledge of user passwords whose structure is represented by a grammar. The use of *probabilistic context-free grammars* (PCFG) for password cracking was originally proposed by Weir et al. [25] and

implemented in the form of a Python-based tool[17]. The tool consists of two scripts: the *PCFG Trainer* that performs an automated analysis of an input dictionary and creates a PCFG; the second script is called *PCFG Manager* which generates password guesses from an existing PCFG. Using the same concept, we created a fast, parallel, and portable alternative for the original password generator. Our tool[18], written in Go[19] compiled language, is currently employed in Fitcrack on both server and client side, and enables to perform a PCFG attack. The details of the tool are described by a seperate paper [9] and a technical report in czech [11].

3.5.1 The use of probabilistic context-free grammars

The mathematical model is based on classic context-free grammars [5] with the only difference that each rewriting rule is assigned a probability value. The grammar is created by training on an existing password dictionary. Each password is divided into continuous fragments of letters - the apha symbols (A), digits (D), and other characters (O). For fragment of length n, a rewriting rule of the following form is created: $T_n \rightarrow f : p$, where T is a type of the character group (A, D, O), f is the fragment itself, and p is the probability obtained by dividing the number of occurrences of the fragment by the number of all fragments of the same type and length. In addition, we add rules that rewrite the starting symbol (S) to *base structures* which are non-terminal sentential forms describing the structure of the password [25]. For example, password "p@per73" is described by base structure $A_1S_1A_3D_2$ since it consist from a single letter followed by a single special character, three letters, and two digits. Table 4 shows rewriting rules of a PCFG generated by training on two passwords: "pass!word" and "love@love". There is only one rule that rewrites S since both passwords are described by the same base structure. By using PCFG on MySpace dataset (split to training and testing part), Weir et al. were able to crack 28% to 128% more passwords in comparison with the default ruleset from *John the Ripper* (JtR) tool[20] using the same number of guesses.

The proposed approach, however, does not distinguish between lowercase and uppercase letters. Weir extended the original generator by adding capitalization rules like "UULL" or "ULLL" where "U" means uppercase and "L" lowercase. The rules are applied to all letter fragments which increases the number of generated guesses [24]. The current version also contains numerous enhancements. Houshmand et al. introduced keyboard patterns represented by additional rewriting rules that helped improve the success rate by up to 22% and proposed the use of Laplace probability smoothing [6].

[17] https://github.com/lakiw/legacy-pcfg
[18] https://github.com/nesfit/pcfg-manager
[19] https://golang.org/
[20] https://www.openwall.com/john/

left	→	right	probability
S	→	$A_4 O_1 A_4$	1
A_4	→	pass	0.25
A_4	→	word	0.25
A_4	→	love	0.5
O_1	→	@	0.5
O_1	→	!	0.5

Table 4. An example of PCFG rewriting rules

3.5.2 Distributed PCFG attack

To perform a PCFG attack, it is necessary to have a probabilistic grammar in the format used by Weir's PCFG trainer. Fitcrack supports two ways of obtaining the grammar. The WebAdmin either allows the user to upload a ZIP archive containing the grammar, or to select a password dictionary while the WebAdmin will let the PCFG Trainer analyze it and create a grammar automatically.

The PCFG attack in Fitcrack is based on the distribution of preterminal structures, i.e., sentential forms representing partially generated passwords. The idea utilizes the fact that each preterminal structure produces passwords with the same probability. The server only generates the preterminal structures (PT), while the terminal structures, i.e., the candidate passwords, are produced by the cracking nodes.

For distributed computing, our PCFG Manager can be run either as a standalone server that generates PTs, or as a client that generates the final password guesses. Both sides can communicate via gRPC[21] and Protocol buffers[22], as described in the tool's documentation [11]. The concept is utilized in Fitcrack as well with slight modifications. Since the client-server communication in BOINC is based on passing input/output files, we added a new workunit input template `pcfg_in` (see table 7) with two extra input files:

- `preterminals` - the file contains one or more preterminal structures that are used for generating password guesses within the workunit.
- `grammar` - the PCFG grammar in a marshalled (serialized) form. The serialization is performed by the `Pcfg` endpoint in WebAdmin backend (see section 4.4) at the time the grammar is created. The file is intentionally marked as *sticky* which implies that BOINC will only send it once - with the first workunit.

When a user launches a PCFG attack, the PCFG Monitor daemon (See Section 4.9.) starts the PCFG Manager server to listen at a TCP port calculated as:

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

[21] https://grpc.io/
[22] https://developers.google.com/protocol-buffers

where *JobId* is the ID of the corresponding job. This allows to run multiple PCFG attack at the same time. Using the `Connect()` call, the *Generator* (see section 4.13) then connects to the running instance of the PCFG Manager server.

When creating a workunit, the *Generator* invokes `GetNextItems()` call to obtain one or more preterminal structures. With the call, the *Generator* also specifies a keyspace value that is necessary to enable the adaptive scheduling (see section 2.4). In response, the PCFG Manager server will then give the *Generator* a chunk of as many PTs as necessary to generate as least the desired amount of password, and no more. An example is illustrated in Figure 14. The exact match can not be guaranteed because different PTs may generate different number of password guesses. The generator creates a `preterminals` file with all obtained PTs. The `preterminals` file together with the `grammar` file represent the input data for the new workunit.

On the client side, the Runner (see section 5.3) launches an instance of PCFG Manager client and passes it the `grammar` and `preterminals` file. Using a pipe, the Runner then connects standard output of the PCFG Manager to the standard input in hashcat. The hashcat is started in wordlist attack mode without the specification of a concrete dictionary, so that it reads all the passwords guesses directly from the pipe. For cracking more complex hash algorithms, the PCFG Manager may generate guesses faster than the hashcat manager to verify them. Thus, the Runner sets the pipe as buffered and blocking to make the PCFG Manager wait if necessary.

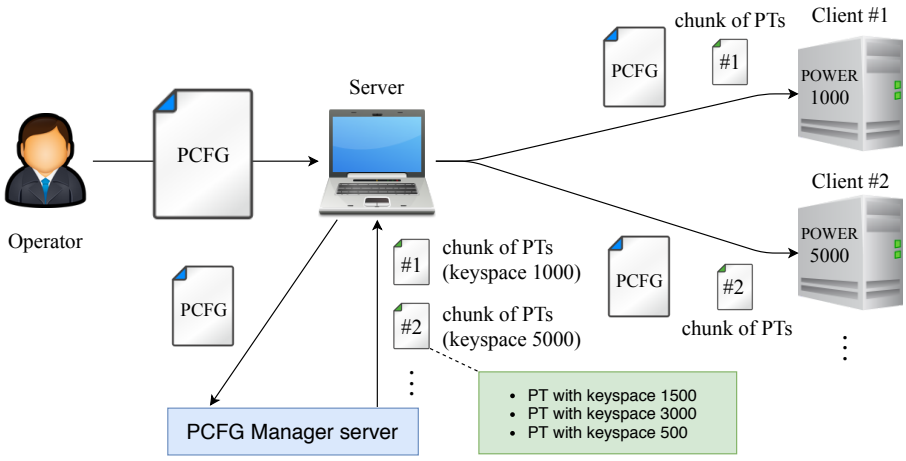

Fig. 14. Example of PCFG attack distribution

3.6 PRINCE attack

PRINCE (*PRobability INfinite Chained Elements*) is a modern password generation algorithm that can be used for advanced combination attacks. Jens Steube designed this algorithm to use only one vocabulary instead of two different dictionaries and then generate all possible combinations of words to form chains of combined words. These chains can contain one to N words derived from the input dictionary and concated together. The reference implementation of the algorithm - *princeprocessor* - is available[23] under MIT license.

3.6.1 Basic algorithm components

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

- **Element**
  An element is the smallest entity representing unmodified item (line, word) of the input dictionary. All elements should be sorted by their relevance / frequencies and group by the length into the database of elements. Some examples of elements are presented in table 5.

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

Table 5. Examples of elements

- **Chain**
  A chain with the length L is sorted sequence of element lengths and is equal to L. For example, a chain with its length 8 can be (1, 6, 1) or (8). For the length L there are $2^{(L-1)}$ different chains.
  Example: chain with length 4 can be split to following elements:
  - 4 letter word
  - 2 letter word + 2 letter word
  - 1 letter word + 3 letter word
  - 1 letter word + 1 letter word + 2 letter word
  - 1 letter word + 2 letter word + 1 letter word
  - 1 letter word + 1 letter word + 1 letter word + 1 letter word
  - ...

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

- **Keyspace**
  Keyspace of the chain is equal to number of all candidate passwords which are created by combination of all available elements according to sorted sequence of lengths.
  For example, if we have X elements of length 2 and Y elements of length 6 in our input dictionary, then keyspace of chain (6, 2) is $X * Y$. Keyspaces of some chains from the *rockyyou* dictionary are shown in table 6.

Chain	Elements	Keyspace
3 + 1	335 * 45	15 075
1 + 3	45 * 335	15 075
4	17 889	17 889
2 + 2	335 * 335	112 225
2 + 1 + 1	335 * 45 * 45	678 375
1 + 2 + 1	45 * 335 * 45	678 375
1 + 1 + 2	45 * 45 * 335	678 375
1 + 1 + 1 + 1	45 * 45 * 45 * 45	4 100 625

Table 6. Keyspace of chains with length 4 (passwords from *rockyyou*)

The total keyspace p is then computed as the sum of chain keyspaces:

$$p = \sum_{i=1}^{n} keyspace(chain_i)$$

where n is the number of generated chains.

Algorithm 5: Pseudocode of the PRINCE algorithm

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

Output: candidate passwords

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

3.6.2 Distributed PRINCE attack

The our idea behind PRINCE attack distribution is illustrated in Figure 15.

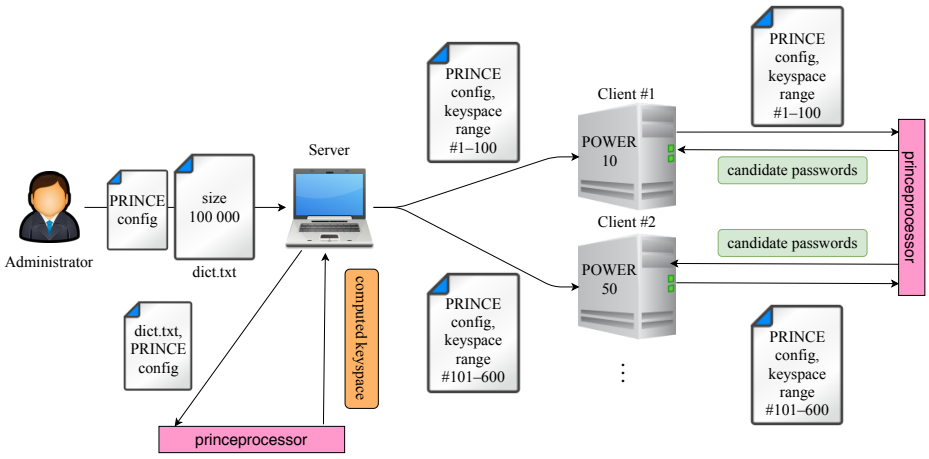

Fig. 15. Scheme of PRINCE attack distribution

An user creates a new PRINCE attack job. The server receives the configuration of created PRINCE attack job together with the user-selected dictionary. To

get a total keyspace of this job, the server needs to run princeprocessor with this dictionary and specific flags according to the configuration of this PRINCE attack job. Then, the *Generator* on the server side assigns a keyspace range to every active client according to the benchmarked "power" of the client. This range info is a part of the config of every workunit with PRINCE attack. An example of attack distribution is shown in Figure 16.

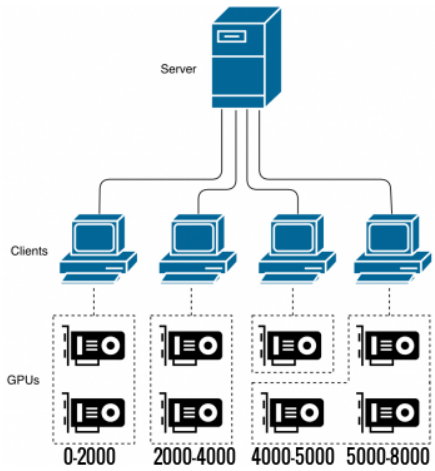

Fig. 16. An example of PRINCE attack distribution

PRINCE attack on the client side relies on two options of the princeprocessor as the external password generator:

- **--skip=X**
  princeprocessor will skip first X generated passwords from the start.
- **--limit=Y**
  princeprocessor will generate only Y passwords on the output.

Runner on the client side launches princeprocessor with options --skip and --limit to generate assigned keyspace range of candidate passwords using the PRINCE algorithm. Runner internally connects princeprocessor with hashcat using pipes, so hashcat can crack the candidate passwords as soon as they are generated by princeprocessor.

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.

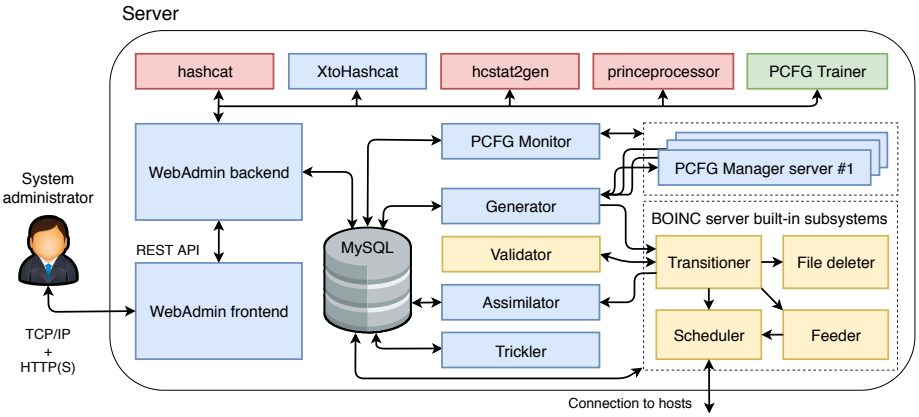

Fig. 17. The architecture of Fitcrack server

While for the client, we support both Windows, and Linux nodes, the server has a Linux-only implementation. As illustrated in figure 17, Fitcrack server consist of multiple subsystems: *Generator* (see section 4.13), *Validator* (see section 4.14), *Assimilator* (see section 4.15), and *Trickler* (see section 4.16) are the main scheduling-related daemons implemented within Fitcrack. They closely relate to BOINC built-in subsystems which are: *Transitioner* (see section 4.17), *Scheduler* (see section 4.18), *Feeder* (see section 4.19), and *File deleter* (see section 4.20). 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 17: *hashcat*, *Hashvalidator*, *XtoHashcat*, and *hcstat2gen*.

For storing all cracking-related information we use a MySQL[24] 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[25] 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.

[24] https://www.mysql.com/
[25] 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-5.1.0` - hashcat binaries,
    - `hashcat-utils` - hashcat-related utilities,
    - `hashvalidator` - the *Hashvalidator* tool,
    - `princeprocessor` - the *princeprocessor* tool,
    - `pcfg_manager` - the *pcfg_manager* tool,
    - `pcfg_trainer` - the *pcfg_trainer* tool,
    - `pcfg_mower` - the *pcfg_mower* 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 front end is a web application built on the *Vue*[26] web framework and using the *Vuetify*[27] user interface component framework. As the user-facing part of WebAdmin, it is made to be a user-friendly way of controlling most aspects of the Fitcrack system.

The application supports multiple users with different roles, allowing for granular control over permissions. Remote access is available by simply exposing the server to the internet. Users also receive notifications about events in the system.

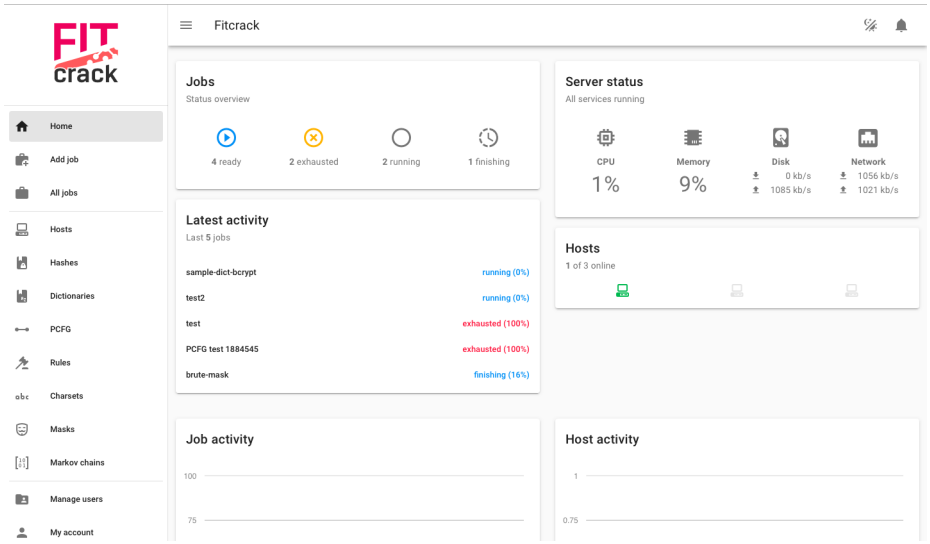

Fig. 18. The dashboard of Fitcrack WebAdmin

A dashboard, pictured in fig. 18, serves as the homepage, aggregating information and statistics from multiple areas for easy overview of the system. All other functionality is separated into distinct views, accessible at their own URIs via a navigation menu, thanks to the *Vue router* and the use of components. These sections can be categorized into three groups:

4.3.1 Job creation and management

The cracking jobs and their creation, management, and monitoring are the main focus of WebAdmin and a crucial part of the system as a whole. To help create new jobs easily, WebAdmin offers a four step interactive input form, allowing the

[26] https://vuejs.org
[27] https://vuetifyjs.com

user to set up the job in detail, while also making it possible to save the settings into named templates. The form guides the user through entering the job input, setting up an attack by way of a graphical interface mirroring Hashcat's attack settings, assigning the job to host nodes, and more.

All jobs added to the system can then be controlled and monitored via various listings, including user defined bins. The listing view implements a search and filter function as well as quick access to most used functions for each job and an overview of the most important information.

The job detail view shows all information about a job and affords further control over it. It also shows various charts with statistics, graphing the job's progress and showing how it's being distributed. From here, the user can even see detail about each workunit and access logs from its run.

4.3.2 Asset library management

Many different assets used by the system in creating or running jobs can be managed via their WebAdmin pages. The system comes with some by default, but most anything can be changed or deleted to suit the user's preference. Some of the assets can also be edited in place. The asset management pages include:

- **hash cache** – stores every hash and its associated password, if one was found, for future reference by the system as well as by the user
- **dictionaries** – allows adding or removing password dictionaries, upload via HTTP or import from server directory
- **charsets** – stores custom non-ASCII character set files
- **rules** – stores text files with rule sets for password mangling
- **masks** – stores text files with masks, which can be loaded into a brute force attack setup
- **markov chains** – stores *hcstat2* files with character statistics, and provides a way to generate them from an existing dictionary
- **PCFG** – stores PCFG assets in directories and accepts premade ones in zip archives or generates from dictionaries

4.3.3 System and user administration

The host nodes connected to the system can be viewed on the hosts page. While adding and removing hosts is not in WebAdmin's scope, as that is handled by the BOINC framework, the user can still access information about the hosts' hardware, work done, and their status.

Each user can see their own information and change their password on the account page. With the permission to manage users, another view becomes available, in which the user can create and manage accounts, assign them a role, and also create or remove roles and grant or revoke permissions they possess.

The server monitor shows hardware usage of the main server over time. It allows the user to specify a time frame to examine. It also shows which of the server services are running.

On the settings page, the user can configure various aspects of WebAdmin as well as change global system settings that are taken into account by other modules.

4.4 WebAdmin backend

The backend, written in Python 3, is based on Flask[28] 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[29], 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 verifying hash format and calculating the keyspace of masks (see section 5.4),
- **XtoHashcat** - for hash extraction (see section 4.6),
- **hcstat2gen** - for generating Markov statistics (see section 4.7),
- **princeprocessor** - for calculating PRINCE keyspace (see section 4.11),
- **pwd_dist** - for calculating password length distribution (see section 4.12),
- **PCFG Trainer** - for creating probabilistic grammars (see section 3.5).

For handling frontend requests, the backend exposes the following API:

Charset

Endpoints for work with charset files.

- `GET /charset` – Ruturns collection of HcStats files
- `POST /charset/add` – Uploads charset file on server
- `GET /charset/{id}` – Returns information about charset with data
- `DELETE /charset/{id}` – Deletes charset
- `GET /charset/{id}/download` – Downloads charset

Dictionary

Endpoints for work with charset dictionary.

- `GET /dictionary` – Returns collection of dictionaries
- `POST /dictionary/add` – Uploads dictionary on the server
- `POST /dictionary/fromFile` – Makes dictionary from file
- `GET /dictionary/{id}` – Returns information about dictionary
- `DELETE /dictionary/{id}` – Deletes dictionary
- `GET /dictionary/{id}/data` – Returns first 25 rows from dictionary
- `GET /dictionary/{id}/download` – Sends zipped PCFG as attachment

[28] http://flask.pocoo.org/
[29] https://www.sqlalchemy.org/

Directory

Endpoinsts for browsing filesystem of server.

- `GET /directory` – Returns list of files in directory

Graph

Endpoints for graph presentation

- `GET /graph/hostPercentage/{id}` – Returns 2D graph representing ratio of host's computing power
- `GET /graph/hostsComputing` – Returns 2D graph representing computing power of active hosts
- `GET /graph/hostsComputing/{id}` – Returns 2D graph representing computing power of active host
- `GET /graph/jobsProgress` – Returns 2D graph representing progress of started jobs
- `GET /graph/jobsProgress/{id}` – Returns 2D graph representing progress of started job

Hashcat

Endpoints for hashcat usage.

- `GET /hashcat/attackModes` – Returns list of supported attacks
- `GET /hashcat/hashTypes` – Returns list of supported hashes

Hashes

Operations with hashes.

- `GET /hashes` – Returns list of hashes

Hosts

Operations with hosts.

- `GET /hosts` – Returns list of hosts
- `GET /hosts/info` – Returns information about hosts
- `GET /hosts/{id}` – Returns exact host
- `DELETE /hosts/{id}` – Removes host from table

Job

Operations with jobs.

- `GET /job` – Returns list of jobs
- `POST /job` – Creates new job
- `GET /job/crackingTime` – Calculates cracking time
- `GET /job/info` – Returns information about jobs
- `GET /job/lastJobs` – Returns the last jobs at which there were changes of the state
- `POST /job/verifyHash` – Verifies format of uploaded hash
- `GET /job/{id}` – Returns job
- `DELETE /job/{id}` – Deletes job
- `PUT /job/{id}` – Changes created job
- `GET /job/{id}/action` – Operations with job(restart, start, stop)
- `GET /job/{id}/host` – Returns list of hosts that are working on job
- `POST /job/{id}/host` – Mapping of hosts to job
- `GET /job/{id}/job` – Returns workunits to which job was devides

Log

Endpoints for log operations.

- `GET /log` – Returns logs
- `GET /log/new` – Returns new logs

MarkovChains

Endpoints for work with HcStats files.

- `GET /markovChains` – Returns collection of HcStats files
- `POST /markovChains/add` – Uploads HcStats files on server
- `POST /markovChains/makeFromDictionary` – HcStat from dictionary
- `GET /markovChains/{id}` – Downloads HcStat file
- `DELETE /markovChains/{id}` – Deletes HcStat file

Masks

Endpoints for work with mask files.

- `GET /masks` – Returns collection mask files
- `POST /masks/add` – Uploads mask file on server
- `GET /masks/{id}` – Returns information about maskset with data
- `DELETE /masks/{id}` – Deletes mask
- `GET /masks/{id}/download` – Downloads maskset
- `POST /masks/{id}/update` – Exchanges maskset with new string

Notifications

Endpoints for graph representation.

- `GET /notifications` – Returns user's notifications
- `GET /notifications/count` – Returns number of unreaded notifications

Pcfg

Endpoint for pcfg operations

- `GET /pcfg` – Returns collection of pcfg
- `POST /pcfg/add` – Upload pcfg on server
- `POST /pcfg/makeFromDictionary` – Creates pcfg from the dictionary
- `GET /pcfg/{id}` – Sends zipped PCFG as attachment
- `DELETE /pcfg/{id}` – Deletes pcfg

ProtectedFiles

Endpoints for operations with files with passwords.

- `GET /protectedFiles/` – Returns collection of hashed files
- `POST /protectedFiles/add` – Uploads hashed files on server
- `GET /protectedFiles/{id}` – Downloads hashed file

Rule

Endpoints for work with rule files.

- `GET /rule` – Returns collection of HcStats files
- `POST /rule` – Uploads rule file on server
- `GET /rule/{id}` – Returns information about rule file
- `DELETE /rule/{id}` – Deletes rule file
- `GET /rule/{id}/data` – Returns first 25 rows from dictionary
- `GET /rule/{id}/download` – Downloads rule
- `POST /rule/{id}/update` – Replaces rule with new string

ServerInfo

Operation with server.

- `GET /serverInfo/actualUsageData` – Returns last record from table fc_server_usage
- `GET /serverInfo/control` – Operations with server(restart, start, stop)
- `GET /serverInfo/getUsageData` – Returns data from table fc_server_usage according to a given time
- `GET /serverInfo/info` – Information about server
- `GET /serverInfo/saveData` – Function for saving of new data into the table fc_server_usage

Settings

Endpoints for manipulating system settings.

- `GET /settings` – Returns all system settings
- `POST /settings` – Sets all system settings

Status

Endpoints for reading changes in job status.

- `GET /status` – Returns collection of all job statuses
- `GET /status/{id}` – Returns collection of all job statuses

Template

Endpoints for job templates.

- `GET /template` – Returns collection of job templates
- `POST /template` – Add a job template
- `GET /template/{id}` – Returns a template with its job settings
- `DELETE /template/{id}` – Deletes a template

User

Endpoints for authorization.

- `GET /user/` – Returns list of users
- `POST /user/` – Adds new user
- `GET /user/isLoggedIn` – Finds out if user is logged in and returns him
- `POST /user/login` – User login
- `GET /user/logout` – User logout
- `POST /user/password/change_my_password` – Changes user's password
- `GET /user/role` – Returns list of roles
- `POST /user/role` – Changes user's role
- `POST /user/role/new` – Adds new role into DB
- `DELETE /user/role/{id}` – Deletes role
- `POST /user/role/{id}` – Changes one property in user's role
- `DELETE /user/{id}` – Deletes user

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. Another use for hashcat is to verify the format of input hashes. For that purpose, WebAdmin runs hashcat with the `--show` argument. For more about hashcat see section 5.4.

4.6 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[30,31] 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`) [26, 27],
- **PDF documents** (`-f 10400-10700`) [1],
- **RAR archives** (`-f 12500/13000`) [20, 16],
- **ZIP archives** (`-f 1300`) [3],
- **7z archives** (`-f 11600`) [23].

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.

4.7 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**[32]. This tool generates a custom Markov statistics file from selected dictionary. The usage is:

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

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

Starting from version 4.x, hashcat requires the file to be LZMA-compressed[33] compression. . The compression can be done in the following way:

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

4.8 Monitoring System Usage

We use a stand-alone script, *measureUsage*, to monitor server resources. This script checks periodically for CPU and memory usage on the server and reports the results every minute to REST API. The network usage is monitored as well. The results are stored in the database and visualized to user through WebAdmin frontend.

4.9 PCFG Monitor

PCFG monitor is our custom daemon that periodically checks for new running PCFG jobs. When it detects such a job with no *PCFG Manager* assigned to generate preterminal structure, the daemon launches it. Checking for running PCFG jobs is done using periodic database polling every 2 seconds. The existence of assigned *PCFG Manager* is realized by checking for a process with a correct name, listening on a correct port computed with given job ID as described in subsection 3.5.2

While this functionality was governed by WebAdmin before, because of the new features like automatic starting of batch jobs, we created a simple stand-alone daemon for this.

4.10 PCFG Manager server

The PCFG attack is the only mode where the server actively contributes to the password guessing process. For each job, the PCFG Monitor runs a single instance of the PCFG Manager[34] in server mode. The running instance generates preterminal structures (PT) from a given grammar and communicates with the Generator via gRPC and Protocol buffers technologies [12, 9].

When the Generator creates a workunit for a host within a PCFG attack job, it asks the PCFG Manager for a chunk containing one or more preterminal structures. The newly created chunk is transferred with the workunit to the host machine with a running PCFG Manager client that uses it for generating password guesses.

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

[34] https://github.com/nesfit/pcfg-manager

4.11 Princeprocessor

The PRINCE attack requires *princeprocessor* tool on the server side to compute a job based on an input dictionary and configuration options of the PRINCE algorithm. Configurations options are converted to *princeprocessor* arguments. Fitcrack server runs *princeprocessor* with an input dictionary, converted arguments and option `--keyspace`. The *princeprocessor* tool then computes the keyspace and prints it to the standard output stream.

4.12 pwd_dist

The tool calculates password length distribution for dictionary file passed as first argument. When user uploads new dictionary, Fitcrack server uses this tool to compute password length distribution of upload dictionary. For example, if we have three passwords with length 8, two passwords of length 6 and one password of length 12, tool prints: `6:2;8:3;12:1`. Computed password length distribution is stored as *password_distribution* in the table *fc_dictionary*.

4.13 Generator

The *Generator* is a server daemon responsible for creating new workunits for hosts. To achieve an efficient use of network's resources, it employs our *Adaptive scheduling algorithm* [7] described in Section 1. The algorithm tailors each workunit to fit the client's computational capabilities based on the current cracking speed, which could change over time. To get the initial speed, at the beginning of each cracking job, the clients receive a *benchmark* job which measures their cracking speed for a given hash type. For the attack modes, the Generator implements the distribution strategies described in Section 12. For example, it performs the fragmentation of dictionaries, calculates the appropriate password index boundaries, loads the preterminal structures for the PCFG attack, etc.

The Generator daemon runs in a loop illustrated by Algorithm 6. It takes care that for each running job, there is always at least a single workunit assigned. And if possible, all participating hosts have a workunit. 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 supported hash 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. Each job in Fitcrack goes through a series of states. All possible states are enlisted in Table 13, while each has a unique number from 0 to 12. Numbers above 10 mean the job is

not running. Historically, not all are numbers are used; some are reserved for future use. The lifetime of a job is illustrated in Figure 20. Once created, the job is in the Ready state. Clicking the start button changes the state to Running. The following transitions depend on the conditions. If there is at least one non-cracked hash and no error or user action occurs, the job eventually proceeds to the Finishing state. In this state, the entire keyspace is distributed, but some hosts are still processing. Once all hosts are done, the job switches to either Finished or Exhausted state, depending on the results. A user may optionally specify a deadline for the job. If exceeded, the job ends in the Timeout state. In case of a non-recoverable error (e.g., the database gets corrupted), the job switches to the Malformed state. Three subsystems of Fitcrack are allowed to change the state of the job: the Generator, when a host asks for a new workunit, the Assimilator if a workunit result is received, and the Webadmin at an event of user's action. Every job starts in the *ready* state, created by a user and added to the database by the WebAdmin backend. Once the user launches it, the job switches to the *running* state. All hosts assigned to a job also have status codes defining the stage of their participation. The host codes are shown in Table 14.

If possible, there are always two cracking workunits ready for each host within the job. One is sent to the host. The second one is generated beforehand, so the host can start working on it right after the first one is completed. 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.

Another feature is job purging. When the user wants to purge the job, revert all the work done and delete the progress, Generator sends signal to all connected clients to abort the current task. When the purged job is running, Generator also reverts its state back to *Ready*. Generator is notified about this duty by a `kill` flag in the *fc_job* table, which is set by WebAdmin, after the user clicks at the purge button.

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, containing *hcstat2* file is created.

4.14 Validator

The *Validator*[35] is a tool implemented within BOINC. It validates the syntax of all incoming workunit results from clients before they are passed to the Assimi-

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

Algorithm 6: Generator daemon algorithm

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

lator. The subsystem also checks if each result contains all necessary output files. If the job *replication* is active, i.e., a single workunit is assigned to more than one host, the Validator verifies if the replicated results match. TThe technique helps in an untrusted network where we expect hosts may be compromised produce intentionally incorrect results. In Fitcrack, the replication is by default disabled since it reduces the computational power by 50% or more, as discussed in [8]. In volunteer-based computing, the Validator also grants credit to hosts, but this feature is not interesting for our use-case.

4.15 Assimilator

The Assimilator[36] is a server daemon which parses the results supplied by hosts. It decides what to do when a host completes a workunit. As mentioned in section 4.13, there are three types of workunits – the benchmark for one format, complete benchmark, and a normal cracking job – each having a specific type of result. Depending on the result, the Assimilator can modify the job's state in the database or even 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 7. There are three options, how a workunit could end:

- **Successful benchmark** of a node - The Assimilator saves the node's cracking performance to the database.
- **Finished regular job** - The Assimilator updates the job progress. If the host cracked one or more hashes, the passwords are saved to the database. If all hashes are cracked, the entire job is considered done, and all ongoing workunits are terminated. If the whole keyspace is processed, the Assimilator changes the job's status to either finished or exhausted.
- **Computation error** - The Assimilator sets the *retry* flag to the workunit to perform the failure-recovery process, as described in [8].

4.16 Trickler

With the Generator, Validator, and Assimilator, the server knows what host has which workunit assigned. However, the only information about the workunit's progress the server obtains when the host finishes its work and sends a report. To provide the administrator with a more detailed overview, the Runner (See Section 5.3.) uses BOINC Trickle message API[37]. Via this API, each host periodically sends *Trickle messages* that inform the server about partial progress. The goal of the *Trickler* daemon is to process these messages and updates the information in the database. The administrator may then visualize it in the *WebAdmin* application.

This way, 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, Trickler daemon reads them and updates the Fitcrack table *fc_ workunit* with progress. The old entries in the database are periodically removed.

[36] https://boinc.berkeley.edu/trac/wiki/AssimilateIntro
[37] https://boinc.berkeley.edu/trac/wiki/TrickleApi

Algorithm 7: Assimilator daemon algorithm

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

4.17 Transitioner

Transitioner 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.

4.18 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.

4.19 Feeder

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

4.20 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 19.

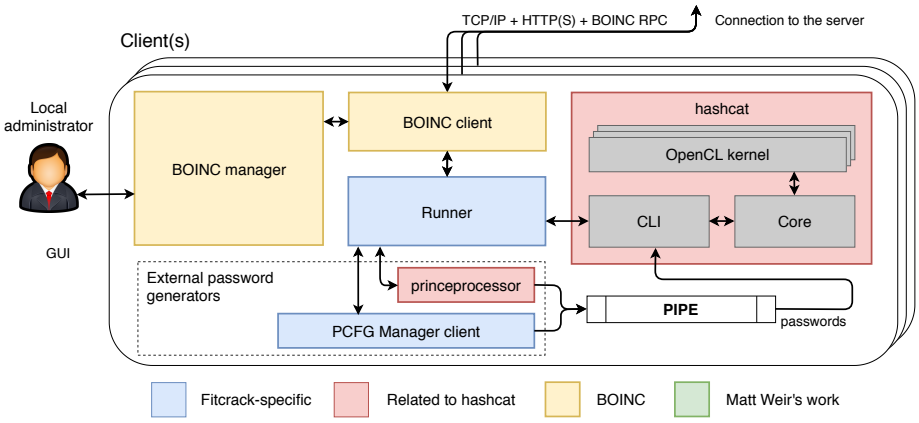

Fig. 19. The architecture of Fitcrack server

5.1 BOINC Client

BOINC Client[38], 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.

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

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.
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[39] 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

The 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 Pre-compilation

In order to maintain maximum compatibility with most systems, the Runner is distributed as a static binary pre-compiled to different architectures. However, in case an experienced user decides to modify it, it is possible to re-compile the application. To compile runner, it is necessary to run `make linux` in the Runner directory to build Runner binary for Linux. Similarly, use `make windows` to produce a binary for Windows. The Makefile requires a MinGW G++ to be installed on the Linux OS to be able cross compile the Runner for Windows.

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

To make development more easy, there are two scripts that help update the binaries in Fitcrack. The *update_client_libs.sh* compiles the Runner for both Linux and Windows and copies the binaries to the `server/client_bin` inside the directory with Fitcrack's sources. If the Fitcrack server is later installed from the directory, it automatically uses the updated binaries. The second script *update_binaries.py* is helpful for live debugging. It can be run on a machine with an existing installation of the Fitcrack server. The script compiles the Runner and updates the currently installed system. BOINC then automatically updates the Runner binaries on client nodes.

5.3.2 Basic operation

Once runner is started, it:

1. reads the `config` file (see section 6.1), and converts the options to *hashcat* parameters,
2. if an attack requires an external password generator, Runner launches it
3. launches hashcat,
4. monitors the cracking progress of cracking,
5. 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:

- required files:
  - `hashcat_files_v510_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:
  - `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),
  - `pretermnals` - a file with *PCFG preterminals*,
  - `grammar` - a file with *PCFG grammar*.
- 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.

Runner launches the hashcat as a cracking engine for all attack modes. In case of PCFG and PRINCE attacks, an external password generator is required to produce candidate passwords and Runner needs to launch it. A light-weight abstraction of processes for Linux and Windows is implemented in *ProcessLinux* and *ProcessWindows*.

An external generator is connected with the hashcat via pipes for the best possible performance. As soon as a generator is able to produce new candidate passwords, the hashcat starts cracking them. A light-weight abstraction of Linux and Windows pipes is implemented in classes *PipeLinux* and *PipeWindows*.

Given that concurrently running multiple instances of hashcat generally leads to a decrease in overall cracking speed as well as increased usage of system resources such as GPU memory, Runner also ensures (see classes *NamedMutexLinux* and *NamedMutexWindows*) that while one instance of Runner currently has a running hashcat process, all other instances will wait for it to finish before launching their own.

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.

5.3.3 Attack specific configuration

Runner maps options from the `config` file to an arguments for hashcat and for external external generators like PCFG manager or princeprocessor. This mapping is implemented in the classes with name starting with "Attack" - eg. AttackDictionary, AttackPCFG, etc.

5.3.4 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.

5.5 PCFG Manager client

In contrast with the classic attack modes, An instance of the PCFG Manager runs in the client mode and generates candidate passwords from the preterminals created by the PCFG Manager server [12, 9], as described in Section 4.10. Through a pipe, the passwords are sent to the standard input of hashcat. Hashcat runs in a dictionary attack mode without a concrete wordlist specified. It reads all the passwords directly from the pipe.

5.6 Princeprocessor

The PRINCE attack also uses an external password generator on the client-side. An instance of the princeprocessor tool runs with options --skip and --limit to generate assigned range of candidate passwords using the PRINCE algorithm. The generated passwords are sent to the standard input of hashcat through a pipe. Hashcat runs in a dictionary attack mode and reads all the generated passwords directly from the pipe.

6 Client-server communication

While the underlaying communication is handled by BOINC using *BOINC scheduling server protocol*[40], 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*[41] 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 then used to bootstrap 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. It is also used to initialize the `power` column in the `fc_host` table, after being converted to the appropriate units. 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 set 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 submodes (represented by the *attack number* - see 3):

- **bench_in** - used for *bench_all* workunits (no attack),
- **dict_in** - used for *dictionary* attacks (00),
- **rule_in** - used for *dictionary* attacks with *rules* (01),

[40] https://boinc.berkeley.edu/trac/wiki/RpcProtocol
[41] https://boinc.berkeley.edu/trac/wiki/JobTemplates

- **combinator_in** - used for *combination* attacks (10, 11, 12, 13),
- **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),
- **hybrid_dict_mask_in** - used for *hybrid (wordlist+mask)* attacks (60),
- **hybrid_mask_dict_in** - used for *hybrid (mask+wordlist)* attacks (70),
- **prince_in** - used for *PRINCE* attacks (80),
- **prince_rules_in** - used for *PCFG* attacks with *rules* (81),
- **pcfg_in** - used for *PCFG* attacks (90),
- **pcfg_rules_in** - used for *PCFG* attacks with *rules* (91).

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),
- **preterminals** - a file with *PCFG preterminal structures*,
- **grammar** - a file with serialized *PCFG grammar*.

Table 7 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	preterminals	grammar
bench_in		X							
dict_in	00	X	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		
prince_in	80	X	X	X					
prince_rules_in	81	X	X	X		X			
pcfg_in	90	X	X					X	X
pcfg_rules_in	91	X	X			X		X	X

Table 7. 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*[42] (TLV) representation. Each line of the config file has the following syntax:

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

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

The *name* identifies the configration parameter. Allowed names are listed in table 9. The *type* matches one of the data types defined in table 8. 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.

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 9 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.

type	description
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 8. Data types supported in Fitcrack config

Not all config parameters are used in every workunit, e.g. in dictionary attack, we have no mask, etc. Table 10 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. If a parameter may be used but is optional, "O" is displayed in the corresponding column.

For *benchmark* workunits, the workunit config file is identical to the one created for a normal workunit for a given job, aside from parameters limiting the number of cracked passwords, which are currently `start_index`, `hc_keyspace`, `dict_hc_keyspace` and `skip_from_start`. The only exception is for jobs with dictionaries. Their size may be enormous, and they are often fragmented to smaller parts before use. Hence, sending entire dictionaries to all nodes would be a costly and useless effort. Moreover, the exact content of a dictionary is not necessary for a benchmark. As we experimentally detected, the measured performance is influenced mainly by the distribution of password lengths. Therefore, we use an alternative solution. The server sends an empty file instead, plus

name	describtion	hashcat arg.	princeprocessor 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	
mask	password mask for brute-force attack		
start_index	starting password index (see 2.3)	--skip	
hc_keyspace	keyspace of the workunit	--limit	
mask_hc_keyspace	keyspace of the entire mask		
mode	computation mode (b|n|a)	-b for bench	
markov_threshold	threshold for Markov model (see3.3)	--markov-threshold	
dict_hc_keyspace	keyspace of the wordlist		--limit
case_permute	case permutation		--case-permute
check_duplicates	check and ignore duplicated passwords		--dupe-check-disable
max_password_len	maximal length of passwords		--pw-max
min_password_len	minimal length of passwords		--pw-min
max_elem_in_chain	maximal number of elements in chain		--elem-cnt-max
min_elem_in_chain	minimal number of elements in chain		--elem-cnt-max
skip_from_start	number of passwords to skip		--skip
generate_random_password	number of generated random passwords	-g	
hw_temp_abort	temperature threshold to abort cracking	-hwmon-temp-abort	
benchmark_dict1	Password length distribution of dict1		
benchmark_dict2	Password length distribution of dict2		

Table 9. Parameters used in the config file

information about password length distribution using special config parameters `benchmark_dict1` and `benchmark_dict2`. Password length distribution for a dictionary is computed once using the **pwd_dist** tool (see 4.12) and then stored to the table `fc_dictionary`. The format of those parameters is a string of concatenated items, where one item is formatted as length followed by a colon, the number of passwords of this length, and a semicolon. For example three 6 character passwords followed by four 8 character passwords would result in the string `3:6;4:8;`. Based on this information, the Runner on the host machine generates a temporary dummy dictionary with similar properties and uses it for the benchmark.

The config file is created by Generator together with the workunit. 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 mode		0		1				3			6	7	8		9	
attack submode		0	1	0	1	2	3	0	1	2	0	0	0	1	0	1
name	**type**															
attack_mode	UInt	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X
attack_submode	UInt	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X
hash_type	UInt	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X
name	UInt	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X
hw_temp_abort	UInt	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X
charset1	String							X	X	X	O	O				
charset2	String							X	X	X	O	O				
charset3	String							X	X	X	O	O				
charset4	String							X	X	X	O	O				
rule_left	String				X		X				O					
rule_right	String					X	X					O				
mask	String							X	X	X	X	X				
start_index	BigUInt			X	X	X	X	X	X	X	X	X				
hc_keyspace	BigUInt			X	X	X	X	X	X	X	X	X			X	X
mask_hc_keyspace	BigUInt							X	X	X						
dict_hc_keyspace	BigUInt												X	X		
mode	String	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X
markov_threshold	UInt										X	X				
max_password_len	UInt												X	X		
min_password_len	UInt												X	X		
max_elem_in_chain	UInt												X	X		
min_elem_in_chain	UInt												X	X		
case_permute	UInt												X	X		
check_duplicates	UInt												X	X		
skip_from_start	BigUInt												X	X		
generate_random_rules	UInt												X	X		

Table 10. Use of config parameters within different attacks

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 11.

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

Table 11. 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
    ```

  - **3** - 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
3
<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.
  - **3** - 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
    3
    <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. 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
```

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*[43] which is used to send *trickle messages* to the server. The messages are processed by the *Trickler* daemon (see section 4.16). 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.

[43] 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[44] server 4.0.9 or higher, respectively MariaDB[45] 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[46]. 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[47].

[44] https://www.mysql.com/
[45] https://mariadb.org/
[46] https://boinc.berkeley.edu/trac/wiki/DataBase
[47] https://boinc.berkeley.edu/

7.2 The overview of Fitcrack tables

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

- **fc_batch** - batch job runs,
- **fc_benchmark** - benchmarking results of hosts,
- **fc_bin** - job bins (folders),
- **fc_bin_job** - bins and jobs junction table,
- **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_job_status** - job status history,
- **fc_mask** - password masks,
- **fc_masks_set** - sets of password masks,
- **fc_notification** - various notifications,
- **fc_pcfg_grammar** - grammars for PCFG attacks,
- **fc_pcfg_preterminals** - PCFG preterminals for given workunit,
- **fc_protected_file** - input files to XtoHashcat,
- **fc_role** - user roles in WebAdmin,
- **fc_rule** - files with password-mangling rules,
- **fc_server_usage** - server resources usage,
- **fc_settings** - global server sertings,
- **fc_template** - job templates,
- **fc_user** - user accounts in WebAdmin,
- **fc_user_permissions** - per-job user permissions,
- **fc_workunit** - workunits (chunks of keyspace).

7.3 fc_batch

The table keeps track of batch job runs. Batches have a name and rememeber who created them. Jobs are assigned to the batch via a foreign key in the fc_job table, which also stores their position in queue. The structure of the table is defined as follows:

- **id** - primary key,
- **name** - name of the batch,
- **creator_id** - user id (linked to fc_user).

7.4 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[48] of hash algorithm,
- **power** - measured cracking speed in hashes per second,
- **last_update** - time of last update of the record.

7.5 fc_bin

The table stores job bins, which group together multiple jobs for listing and managing. Many jobs can be assigned to a bin and many bins can list the same job, so the relationship has a junction table in fc_bin_job (see below). The structure of the table is defined as follows:

- **id** - primary key,
- **name** - name of the bin,
- **position** - position of the bin in the menu.

7.6 fc_bin_job

This is a junction table linking fc_bin and fc_job. The structure of the table is defined as follows:

- **id** - primary key,
- **job_id** - id from the fc_job table,
- **bin_id** - id from the fc_bin table.

7.7 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,
- **keyspace** - number of characters in the charset,
- **time** - time the charset file was added to the system,
- **deleted** - flag (0/1) saying if the charset was deleted.

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

7.8 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 (displayed in WebAdmin),
- **path** - the real name of the dictionary file,
- **password_distribution** - password distribution in the dictionary,
- **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.

7.9 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 `fc_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.

7.10 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.

7.11 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 reported by the workunit most recently completed by the host. It should be expressed in candidate passwords attempted per second, not considering rules.
- **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,
  - **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.

7.12 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.

7.13 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.

7.14 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,

- **attack_mode** - attack mode (see section 3), a value from table 12,
- **attack_submode** - attack submode (see section 3), a value from table 12,
- **hash_type** - number of hashcat's hash type,
- **status** - job status code, a value from table 13, possible transitions can be seen in Figure 20,
- **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), or 0 if fragmenting of the left dictionary is not necessary,
- **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,
- **workunit_sum_time** - total sum of host cracking times,
- **seconds_per_workunit** - time period for a workunit used in the adaptive scheduling algorithm (see 2.4),
- **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"),
- **grammar_id** - ID of grammar from `fc_pcfg_grammar`,
- **case_permute** - flag (0/1) defining if case permutation is enabled or not,
- **check_duplicates** - flag (0/1) defining whether to check for duplicated passwords or not,
- **max_password_len** - maximal length of passwords,
- **min_password_len** - minimal length of passwords,
- **max_elem_in_chain** - maximal number of elements in chain,
- **min_elem_in_chain** - minimal number of elements in chain,
- **generate_random_rules** - number of generated random rules,
- **deleted** - flag (0/1) defining if the job was hidden, or not,
- **kill** - flag (0/1) defining if the job was purged and BOINC workunits should be stopped,
- **batch_id** - ID of the batch the job is assigned in, if any, from `fc_batch`,
- **queue_position** - position in queue when part of a batch, lesser numbers go first

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
6	0	Hybrid attack: wordlist + mask
6	1	Hybrid attack: wordlist + *left rule* + mask
7	0	Hybrid attack: mask + wordlist
7	2	Hybrid attack: mask + wordlist + *right rule*
8	0	PRINCE attack
8	1	PRINCE attack with *password-mangling rules*
9	0	PCFG attack
9	1	PCFG attack with *password-mangling rules*

Table 12. 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 to exceeded `time_end`.
10	running	Computation is in progress.
12	finishing	All keyspace assigned, some hosts still compute.

Table 13. Job *status codes* in Fitcrack

status	name	description
0	benchmark	The host is waiting for, or working on a benchmark.
1	normal	The host is working on a cracking job.
2	-	Currently unused and reserved for future use.
3	done	The host has finished all work on the given job.
4	error	The host encountered an error during the computation.

Table 14. Host *status codes* within a job

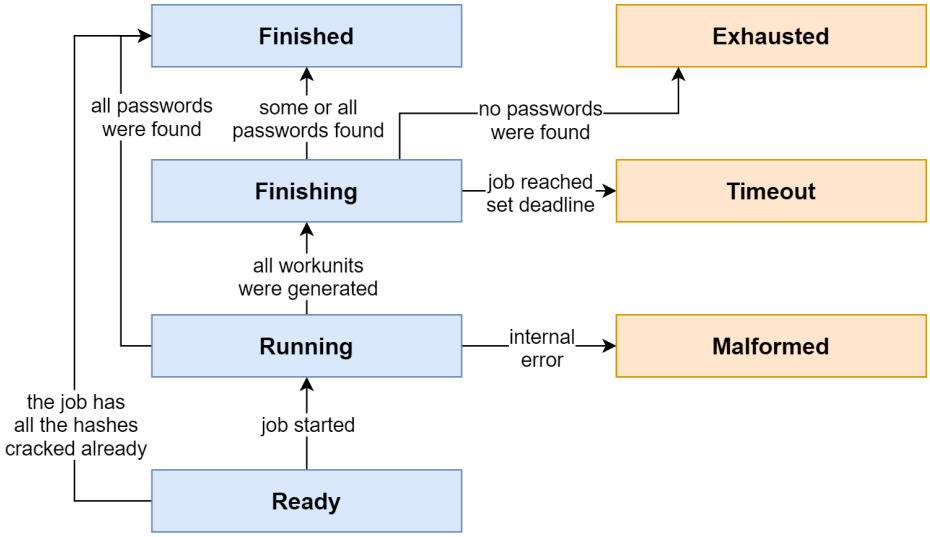

Fig. 20. State diagram of a job in the Fitcrack system

7.15 fc_job_dictionary

This allows Fitcrack to use multiple dictionaries within a *dictionary* or a *combination* attack. *Generator* subsystem (see section 4.13) 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.

7.16 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.

7.17 fc_job_status

The table is used to keep track of status changes for each job. The structure of the table is defined as follows:

- **id** - primary key,
- **job_id** - id of the job, from fc_job,
- **status** - the new status,
- **time** - the time the status change was recorded.

7.18 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.

7.19 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.

7.20 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.

7.21 fc_pcfg_grammar

This table servers as a storage for grammars in PCFG attacks (see section 3.5). Each such grammar has an entry in this table with its essential information.

- **id** - primary key,
- **name** - name of the PCFG grammar file (displayed in WebAdmin),
- **path** - path to the PCFG grammar,
- **keyspace** - number of possible passwords that can be generated from this grammar,
- **time_added** - timestamp of adding a grammar to the system,
- **deleted** - flag (0/1) saying if the grammar was deleted.

7.22 fc_pcfg_preterminals

The preterminals in PCFG attack are generated by the PCFG Manager (see section 3.5) on-the-fly. However, when a workunit is not finished by a certain host, we need to duplicate such a workunit to be done by a different host. PCFG Manager is not capable of returning to a state of previous workunit. Therefore, we have to store the preterminals in a separate table to be used to duplicate such unfinished workunit.

- **id** - primary key,
- **job_id** - ID of the job which these preterminals belong to,
- **workunit_id** - ID of the original workunit,
- **preterminals** - preterminal payload in the format specified by PCFG Manager.

7.23 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 4.6). 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.

7.24 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*!).

7.25 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,
- **count** - number of rules in the file,
- **time** - time the ruleset was added to the system,
- **deleted** - flag (0/1) saying if the ruleset was deleted.

7.26 fc_server_usage

The table is used to store server resource monitoring data. The structure of the table is defined as follows:

- **id** - primary key,
- **time** - the time the data was recorded,
- **cpu** - CPU usage (%),
- **ram** - memory usage (%),
- **net_recv** - network card downlink rate (kbps),
- **net_sent** - network card uplink rate (kbps),
- **hdd_read** - disk reads (kbps),
- **hdd_write** - disk writes (kbps).

7.27 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 structure of the table is defined as follows:

- **id** - primary key,
- **default_seconds_per_workunit** - number of seconds (default: 3600) specifying the default value for `seconds_per_workunit` job parameter,
- **workunit_timeout_factor** - the number (default: 1) multiplying the workunit timeout - the time after a workunit is considered failed (and re-assigned), if no result is received.
- **hw_temp_abort** - temperature threshold (default: 90) when to abort cracking,
- **bench_all** - flag (0/1, default: 1) - defining if the newly-connected hosts should perform the complete benchmark, as described in section 6,
- **distribution_coefficient_alpha** - maximum percentage (default: 0,1) of the remaining keyspace that can be assigned with a single workunit unless it would be below the minimum,
- **t_pmin** - absolute minimum seconds (default: 20) per workunit (including the start of hashcat, etc.), prevents creation of extremely small workunits,
- **ramp_up_workunits** - flag (0/1, default: 0) defining if smaller workunits should be created at start - the size increases until the solving time hits the "Time per workunit" value,
- **ramp_down_coefficient** - minimum fraction of "Time per workunit" that can be created; influences the size of workunits at the end of the job; the lower the value, the smaller the size workunits at the end: 1.0 means no ramp down, 0.0 ramp down is limited only by t_{pmin},
- **verify_hash_format** - flag (0/1, default: 1) defining if WebAdmin should verify the format of input hashes using *HashValidator* tool (see section **??**),

7.28 fc_template

The table is used to store job templates saved in Fitcrack WebAdmin. Each record represents a named template with saved state from the *Add Job* form. The structure of the table is defined as follows:

- **id** - primary key,
- **name** - name of the template,
- **created** - timestamp of when the template was created,
- **template** - serialized form data stored as JSON.

7.29 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-mail, 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.

7.30 fc_user_permissions

The table is designed to store per-job (non-global) user permissions. Each record represents a permission of a user to do specific operation with a specific job. The owner flag also denotes the user who created the job. This user is the only one allowed to assign permissions in this table for their 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`),
- **owner** - flag (0/1) denoting the user created this job,
- **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.

7.31 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 workunit was duplicated due to some problems during computation,
- **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[49]. The most recent version of Fitcrack is located on NES@FIT GitHub page[50]

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] Seymour Ginsburg. *The Mathematical Theory of Context Free Languages*. McGraw-Hill Book Company, 1966.

[6] Shiva Houshmand, Sudhir Aggarwal, and Randy Flood. "Next Gen PCFG Password Cracking." In: *IEEE Trans. Information Forensics and Security* 10.8 (2015), pp. 1776–1791.

[7] 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.

[8] 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?id=11276.

[9] Radek Hranický, Filip Lištiak, Dávid Mikuš, and Ondřej Ryšavý. "On Practical Aspects of PCFG Password Cracking". In: *Data and Applications Security and Privacy XXXIII*. Ed. by Simon N. Foley. Cham: Springer International Publishing, 2019, pp. 43–60. ISBN: 978-3-030-22479-0.

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

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

[10] 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.

[11] Radek Hranický, Dávid Mikuš, and Lukáš Zobal. *Lámání hesel pomocí pravděpodobnostních gramatik*. Czech. Tech. rep. FIT-TR-2019-03, Brno, CZ, 2019, p. 19. URL: https://www.fit.vut.cz/research/publication/12140.

[12] Radek Hranický, Dávid Mikuš, and Lukáš Zobal. *Lámání hesel pomocí pravděpodobnostních gramatik*. Czech. Tech. rep. FIT-TR-2019-03, Brno, CZ, 2019, p. 19. URL: https://www.fit.vut.cz/research/publication/12140.

[13] 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.

[14] Radek Hranický, Lukáš Zobal, Vojtěch Večeřa, and Matúš Múčka. *The architecture of Fitcrack distributed password cracking system*. Tech. rep. FIT-TR-2018-03, Brno, CZ, 2018, p. 61. URL: https://www.fit.vut.cz/research/publication/11887.

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

[16] 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.

[17] 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.

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

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

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

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

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

[23] 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.

[24] Charles Matthew Weir. “Using probabilistic techniques to aid in password cracking attacks”. PhD thesis. Florida State University, 2010.

[25] M. Weir, S. Aggarwal, B. d. Medeiros, and B. Glodek. “Password Cracking Using Probabilistic Context-Free Grammars”. In: *2009 30th IEEE Symposium on Security and Privacy*. May 2009, pp. 391–405. DOI: 10.1109/SP.2009.8.

[26] 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.

[27] 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.