Distributed password cracking with BOINC and hashcat

Radek Hranický[a,*], Lukáš Zobal[a], Ondřej Ryšavý[b], Dušan Kolář[b]

[a] *Brno University of Technology, Faculty of Information Technology, Department of Information Systems, Brno, Czech Republic*
[b] *Brno University of Technology, Faculty of Information Technology, Department of Information Systems, IT4Innovations Centre of Excellence, Brno, Czech Republic*

Abstract

Considering today's challenges in digital forensics, for password cracking, distributed computing is a necessity. If we limit the selection of password-cracking tools strictly to open-source software, hashcat tool unambiguously wins in speed, repertory of supported hash formats, updates, and community support. Though hashcat itself is by design a single-machine solution, its interface makes it possible to use the tool as a base construction block of a larger distributed system. Creating a "distributed hashcat" which supports the maximum of hashcat's original features requires a smart controller that employs different distribution strategies in different cases. In the paper, we show how to use BOINC framework to control a network of hashcat-equipped nodes and provide a working solution for performing different cracking attacks. We also provide experimental results of multiple cracking tasks to demonstrate the applicability of our approach. Last but not least, we compare our solution to an existing hashcat-based distributed tool - Hashtopolis.

Keywords: hashcat, BOINC, cracking, distributed computing, GPGPU

1. Introduction

With the escalating use of computers and mobile devices, forensic investigators frequently face encrypted data which could hide substantial evidence. Though General-purpose computing on graphics processing units (GPGPU) introduced a massive speedup to password cracking, the developers of software supporting data encryption tend to improve the password verification procedures, making the protection harder to be cracked [6]. For example, using hashcat and exhaustive search (brute-force) with a single NVIDIA GTX 1080 Ti GPU, cracking MS Office 2013 documents is 37840x harder[1] than cracking older MS Office ≤ 2003 using SHA1 + RC4 algorithms. Using 8 alphanumeric characters, one can create a total of $2.18 * 10^{14}$ different passwords. Using a GPU denoted above, a MS Office 2003 document can be cracked within 5 days, while cracking MS Office 2013 document may take up to 559 years. With the limited number of GPUs in a computer, cracking today's password protection in a meaningful time may require employing a massive network of multi-GPU nodes.

Our research aims at finding an open-source software solution for fast and efficient distributed password cracking, which could be easily deployed to any TCP/IP-based computer network made of commodity hardware. At first, we searched for an application which could serve as a "cracking engine" of computational nodes in a network. The key critieria were: a) speed, b) the range of supported formats, c) supported attack modes, and d) portability to different platforms. From existing open-source software, we chose *hashcat*[2], a self-proclaimed "World's fastest password cracker" which is distributed under MIT license. Considering speed, team hashcat won 5 of 7 years of *Crack me if you can* (CMIYC[3]) contest. Assessing features, hashcat supports over 200 different hash formats, and several different attack modes: brute-force attack (also reffered to as mask 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*[4] tool. Another reason for choosing hashcat was its support for different operating systems, and hardware platforms. Hashcat developers provide both sources, and pre-compiled binaries for 32-bit, and 64-bit Windows, and Linux. Cracking with hashcat can be performed on various OpenCL-compatible CPUs, GPUs, and even FPGAs, DSPs, and co-processors.

As a framework for distributed computing, we used *Berkeley Open Infrastructure for Network Computing* (BOINC[5]) which was initially designed [1] as a public-resource computing solution. However, in our previous research, we have shown its applicability in password cracking even in private networks [5]. In our use-case, BOINC handles the authentication of computing nodes, provides the distribution and automatic updates of executable binaries, OpenCL kernels, and the input/output data of each cracking task.

*I am corresponding author
Email addresses: ihranicky@fit.vutbr.cz (Radek Hranický), izobal@fit.vutbr.cz (Lukáš Zobal), rysavy@fit.vutbr.cz (Ondřej Ryšavý), kolar@fit.vutbr.cz (Dušan Kolář)

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

[2] https://hashcat.net/

[3] https://contest.korelogic.com/

[4] http://www.openwall.com/john/

[5] https://boinc.berkeley.edu/

1.1. Contribution

By redesigning Fitcrack[6] system [5], we created a solution for high-efficiency GPGPU password cracking using hashcat tool as a client-side computing engine. The offered solution employs the BOINC framework to handle host management, network communication, and work assignment. For each hashcat's attack mode, we propose a convenient strategy for task distribution to utilize the maximum of hardware resources. We show how dictionary segmentation, fine-grained adaptive scheduling, and batch assignment of workunits help to reduce the overhead of the entire cracking process. We experimentally proved that the new solution is capable of performing distributed attacks in a reliable and efficient way. Moreover, we compared our software with the Hashtopolis tool, underlying pros, and cons of each solution.

1.2. Structure of the paper

Section 2 introduces the reader to the current findings in distributed password cracking. Section 3 describes the architecture of the proposed distributed cracking system. In section 4 we describe how to use a distributed environment to perform the same attacks hashcat supports on a single machine. Section 5 shows experimental results supplemented by the comparison with our previous solution, and Hashtopolis tool. Our research is concluded by section 6 which brings together obtained experience and denotes possible future work.

2. Related work

Much of related work is based on the popular *John the Ripper* (JtR) tool. Up to this day, the John's wiki[7] enlists 15 different approaches on parallel and distributed processing with the tool, some of them were later abandoned. The first published academic work on the case was performed by Lim who modified the sources by adding MPI support for *incremental* (brute-force) cracking mode [11]. The solution used a master processor and a fixed number of slave processors. The master processor divided the *keyspace* (a set of all password candidates) to a pre-defined number of chunks, while each slave processor received an equal chunk to solve. The principle of keyspace division was adopted in many subsequent solutions, however, with various alterations since Lim's original technique is only feasible for use in a stable, homogenous environment.

Pippin et al. then proposed the parallel *dictionary* attack for cracking multiple hashes [16]. Instead of dividing keyspace, they assigned different hashes to each node while all nodes used the same password dictionary. We consider the approach to be efficient for large hashlists and simple hash algorithms only. In our previous work [6], we found the highest influence on the cracking time has the calculation of the hash from candidate passwords. The rest is a simple byte array comparison. Thus, if we crack multiple hashes of the same type, we can calculate the hash only once, and compare the result with all the hashes we are trying to crack.

Bengtsson showed the practical use of MPI-based brute-force and dictionary attack using Beowulf high-performance computing (HPC) cluster for cracking MD5-based Unix shadow files and DES-based passwd files [3]. Both attacks were based on simple password-by-password keyspace division.

Apostal et al. brought another enhancement to HPC-based password cracking. The *divided dictionary algorithm* evenly divided the dictionary words between MPI node equipped with GPUs. Using CUDA, GPUs on each MPI node locally calculated the hashes and compared them with the ones which should be cracked [2].

Marks et al. designed a hybrid CPU/GPU cluster formed by devices from different vendors (Intel, AMD, NVIDIA) [12]. The design includes both hardware, and software solution. The distributed network consisted of *Management/storage nodes* which control the calculation and handle user inputs, and *Computation nodes* responsible for the work itself. For interconnection, three different lines were used: 10 Gb/s Ethernet for data transfer, 1 Gb/s Ethernet and Infiband for controlling the computation process. Marks proposed a software framework called *Hybrid GPU/CPU Cluster* (HGPC) utilizing a master-slave communication model using an XML-based protocol over TCP/IP network. A proof-of-concept implementation was able to crack MD5, SHA-1, and four versions of SHA-2 hashes. Experimental results of cracking on up to 24 nodes showed great power, and scalability. We, however, suppose that using a time-proven tool of computation nodes would increase the performance even more. While cracking MD5 hashes on NVIDIA Tesla M2050, Marks achieved the speed around 800 Mh/s, while hashcat users report[8] cracking over 1200 Mh/s using the same GPU.

Previous solutions work well for a "classic HPC" system using a homogenous cluster with a static set of nodes which is, however, not our use-case. Using a simple text-based protocol, loosely modeled on HTTP, Zonenberg created a distributed solution for cracking MD5 hashes using a brute-force attack [19]. The architecture consisted of a master server and a set of compute nodes, which were either CPU-based, or used GPU acceleration based on CUDA.

Crumpacker came with the idea of using BOINC [1] to distribute work, and implemented a proof-of-concept tool for distributed cracking with JtR [4]. Similarly as in our previous prototype [5], he decided to standardize the size of the workunit (chunk in terms of BOINC) by counting out the number of passwords that one computer could check in a reasonable amount of time. Crumpacker, however, reports he was unable to properly distribute the *incremental crack mode*[9] since JtR did not track the starting and ending position of the generated password segments. This was fixed by the modification of JtR database, however at the cost of efficiency. Crumpacker later introduced a *batch concept* which divides passwords to groups

[6] `https://fitcrack.fit.vutbr.cz/`
[7] `https://openwall.info/wiki/john/parallelization`
[8] `https://hashcat.net/forum/thread-2084.html`
[9] `https://www.openwall.com/john/doc/MODES.shtml`

called batches, and tracks them during the entire cracking process, possibly using different hash types, and attack modes [4].

Despite hashcat's speed, JtR still offers some advantages over hashcat, for instance, supports some formats which hashcat does not – e.g. encrypted RAR3 archives with unprotected header. Such formats requires a large piece of work performed on the host machine while hashcat is a pure-OpenCL solution. Thus, we studied the possibilites of JtR integration to Fitcrack, and encountered the similar problems with distibuting incremental mode as Crumpacker reported [4]. We postponed the efforts to future work, and decided to focus on hashcat at the time.

Kasabov et al. performed a research on password cracking methods in distributed environment resulting in a technical report describing different architectures and technologies for work distribution [9]. Kasabov considers MPI combined with OpenCL as the best practical approach for setting up a password cracking GPU cluster, underlying the posibility to use a combination of MPI and OpenMP[10] to achieve fine-grained paralellism [18]. The research is, however, merely theoretical and provides no proof-of-concept tool or experimental results to support the conclusions. Kasabov mentions Zonenberg's Distributed Hash Cracker [19], however Crumpacker's BOINC-based solution [4] is not discussed. Still, the report includes a brief study of BOINC, emphasizing its advantages in automation, which include integrity checks, workunit replication, checkpointing, and others – which we take advantage from in Fitcrack system [5, 8], and Crumpacker in his distributed JtR-based solution [4]. We agree with Kasabov, BOINC is not an "out-of-the-box" solution for password cracking, however we do not consider the actual creation[11] of a password cracking project to be as difficult task as Kasabov describes. Moreover, the statement "BOINC API lacks functions for managing projects" [9] is, at the time of writing this paper, not entirely true. Every BOINC project contains a project management website, allowing the administrator to observe and control project's tasks, users, and other, in a general way. The rest can be added by writing a custom API, like we did for Fitcrack WebAdmin – see section 3.1.1.

Veerman et al. aimed to create a scalable, modular and extensible solution for password cracking using existing cracking tools [15]. The initial research compares the existing password cracking tools, and discusses the possible use of BOINC, and MPI for task distribution. Veerman does not consider BOINC to be an optimal choice due to "large deployment overhead and complexity", refering to Kasabov's research [9]. The use of MPI, as Veerman states [15], requires the cracking tool to either support MPI, or be modifiable for adding the MPI support. Since Veerman wants the solution to support both closed-sourcce tools, such modification is not always possible. The work proposes an architecture consisting of three parts: the *Node Controller* which handles user requests, stores cracking-related data, and schedules work; the *Worker Node* responsible for cracking; and the *Website* serving as a user interface. Generally, the design is similar to the architecture of our Fitcrack system, described in section 3. The assignements entered by user are divited into *subjobs* analogous to *workunits* in Fitcrack. The software output of Veerman's research represents a proof-of-concept system based on PHP, MySQL, Apache and SQLite. From all discussed cracking tools, the proposed PHP-based system could only use JtR with the default cracking configuration (first dictionary, then brute-force attack), and the cracking is limited to MD5 hashes only [15]. From our perspective, the proposed architecture is clear and well-designed, however the software solution is not ready for real deployment due to its limited functionality.

Kim et al. proposed a protocol for distributed password cracking, based on the distribution of password indexes – i.e. starting point and keyspaces of each workunit [10]. The paper clarifies the principle used in various existing tools, including our Fitcrack system [5, 8]. Kim's description is, however, very general, and limited to a brute-force attack on a single hash.

Since we focus our efforts on distributing hashcat, it is necessary to mention existing work, despite being out of the academic sphere. Hashstack[12] is an enterprise solution from Sagitta HPC, a subsidiary of Terahash LLC founded by J. Gosney, a core member of Hashcat development team. Hashstack is described to provide extreme scalability, however, the solution is closed-source and tailored for Sagitta's specialized hardware.

McAtee et al. [13] presented Cracklord[13], a system for hardware resource management which supports creatig job queus and contains a simple hashcat plugin. The plugin allows to remotely run a dictionary or a brute-force attack with a limited set of options. The project, however, seems be updated very rarely, and the last supported version is hashcat 3.

In 2014, a Github user cURLy bOi created Hashtopus[14], an open-source distributed wrapper around oclHashcat, a predecessor of the current hashcat tool. In 2015, it was used by Samek to create a virtual GPU cluster for the purpose of bachelor's thesis [17]. The Hashtopus project was abandoned in 2017, however S. Coray created[15] a fork called Hashtopussy which was rebranded to Hashtopolis in 2018.

Hashtopolis uses a network with a *server*, and one or more *agents* – machines used as cracking nodes. The server provides a user-friendly PHP-based administration GUI, and agent connection point. Agents contain a Hashtopolis client which exist in C#, and Python versions. The communication is based on HTTP(S) and human-readable JSON messages. Hashtopolis allows the user to create and manage cracking tasks. While Fitcrack distinguishes between various attack modes (see section 4) for which it employs different distribution strategies, Hashtopolis is more low-level and does not provide such abstraction. For each task, the user selects a hashlist, one or more files (e.g. password dictionaries) to be transferred to the client, and an *attack command* in form of hashcat starting options.

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

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

[12] https://sagitta.pw/

[13] http://jmmcatee.github.io/cracklord/

[14] https://github.com/curlyboi/hashtopus

[15] https://github.com/s3inlc/hashtopolis

While the attack-based options have to be crafted manually, Hashtopolis handles benchmarking, keyspace distribution into chunks, and automated download of hashcat binaries and other files necessary for cracking. Being the only well-known maintained open-source solution for distributed computing with the current version of hashcat, we consider Hashtopolis as a state-of-the-art tool in our area of research. Thus in this paper, we compare our solution with Hashtopolis using different attack options.

3. Architecture

In this section, we show how we modified [8] the architecture of the original Fitrack system [5] to replace our custom password cracking software with hashcat. Similarly as in other related projects [19, 4, 10, 15], our system is divided into a *server* and a *client* part. In Fitcrack, the server and clients are interconnected by a TCP/IP network, not necessarily only LAN which makes it possible to run a cracking task over-the-Internet on nodes in geographically distant locations. Clients communicate with the server using an RPC-based *BOINC scheduling server protocol*[16] over HTTP(S). The current architecture of Fitcrack is shown in Figure 1, and is fairly different from the original one described in [5]. In the following sections, we will describe the subsystems on both server and client sides.

3.1. Server

The server is responsible for the management of cracking jobs, and assigning work to clients. In our terminology, a *job* represents a single cracking task added by the *administrator*. Each job is defined by an attack mode (see section 4), attack settings (e.g. which dictionary should be used), and one or more password hashes of the same type (e.g. SHA-1). Once the job is running, the keyspace is continuously split into smaller chunks called *workunits*. In terms of the *client-server* architecture, the service offered by the server is a *workunit* assignment.

3.1.1. WebAdmin

We created a completely new solution for remote management of Fitcrack. The application is called *WebAdmin* and consist of two separate parts: *frontend* and *backend* connected with each other using a REST API.

The frontend is written in *Vue.js* and allows the administrator to manage different parts of the system as depicted in Figure 2. Under *Jobs* tab, the administrator can add, modify and manage all cracking jobs. *Hosts* section provides an overview of connected clients, their software and hardware specification, jobs the clients were participating on, and workunits assigned to them. Every hash, cracked or not, can be viewed in a summary within *Hashes* tab. *Dictionaries* tab can be used to manage and add password dictionaries. Fitcrack supports three ways of adding new dictionaries: a) importing directly from the server; b) uploading new via web using HTTP; c) upload using SFTP/SCP, if configured. Using *Rules* tab, the administrator can manage **.rule* files contaning the password-mangling rules for hashcat. *Charsets* and *Masks* tabs allows to manage character sets and password masks used for mask attack. Since for mask attack hashcat generates passwords using *Markov chains* [14], it is necessary to provide a **.hcstat* / **.hcstat2* (for hashcat 4+) file with per-position character statistics. In *Markov chains* tab, Fitcrack supports adding *hcstat* files either by uploading an existing file, or by generating a new one. The second option stands for an automated training on a password dictionary using *hcstatgen* tool. Least but not last, in *Users* tab, WebAdmin allows to manage user accounts and permissions.

The backend, written in Python 3, is based on Flask[17] 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[18], the backend operates a MySQL database which serves as a storage facility for all cracking-related data. For selected operations, the WebAdmin uses a set of external utilities.

3.1.2. Utilities used by WebAdmin

We wanted to save the administrator's time and extend the usability by an automation of frequent tasks like adding new cracking jobs. For that purpose, Fitcrack WebAdmin uses the following external utilities:

- **hashcat** is used for calculating keyspace. This is important since hashcat's keyspace may **not** correspond with the actual number of possible passwords.
- **Hashvalidator** is our custom tool adopting a vast portion of hashcat's source code. It is used for validating the format of password hashes.
- **maskprocessor** implemented by Jens Steube, the author of hashcat, is used for generating dictionaries from masks, and is used for hybrid attacks (see section 4).
- **XtoHashcat** is our tool used for automated detection of format, and hash extraction from the input data. For some formats, where it is possible (e.g. ZIP or Office documents), it detects the signature and contents of the file and calls one of the existing scripts which extracts the hash.
- **hcstatgen** from *hashcat-utils*[19] repository is used for generating **.hcstat* files from dictionaries.

3.1.3. Generator

Generator is a server deamon responsible for creating new workunits and assigning created workunits to clients. To achieve an efficient use of network's resources, it employs our *Adaptive scheduling algorithm* described in [5]. The algorithm

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

[17] `http://flask.pocoo.org/`

[18] `https://www.sqlalchemy.org/`

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

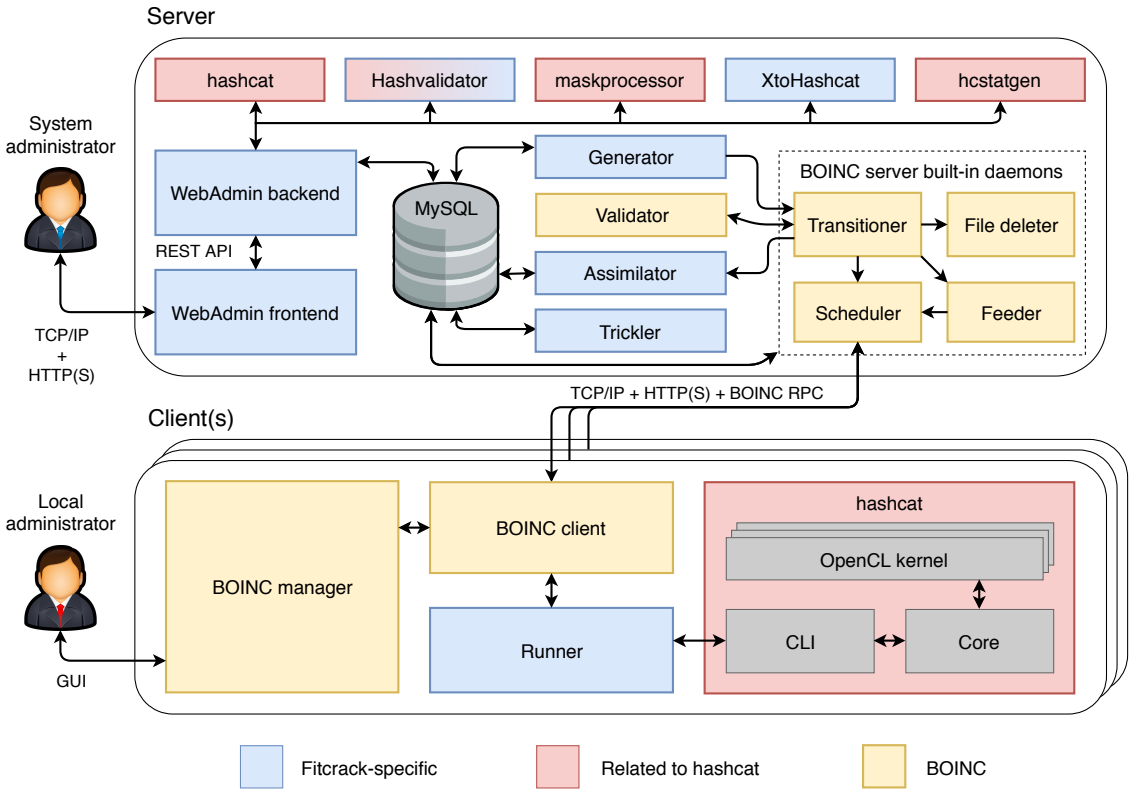

Figure 1: The architecture of Fitcrack server and client

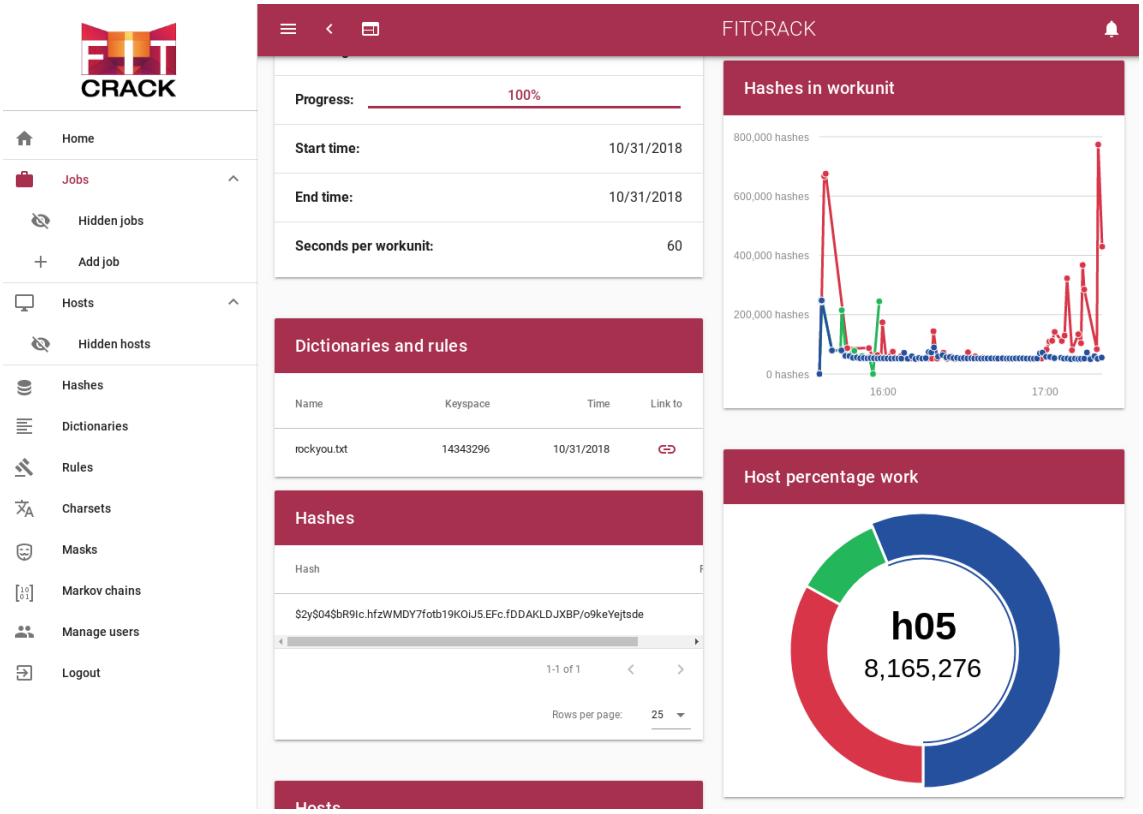

Figure 2: The interface of Fitcrack WebAdmin

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.

To support all hashcat's attacks, we had to redesign the original Fitcrack Generator [5] completely. The new version of Generator is able to work with password *masks*, which are processed in a sequential order if multiple masks are set for a job. For mask attack, we also added support for assigning Markov **.hcstat* files to clients. Moreover, we had to take the password-mangling rules into account since they have a strong influence on the attack's complexity. If the rules are used within a *dictionary* attack, the actual keyspace is multiplied by a number of rules applied to each password. Last but not least, we added support for *combinator attack* which, in our case, includes *hybrid attacks* as well. The exact strategies for keyspace distribution within each attack mode are described in section 4.

3.1.4. Validator

Validator is a tool implemented within BOINC. It validates the syntax of all incoming workunit results from clients, before they are passed to Assimilator. If the job *replication* [5] is active, the Validator verifies if the replicated results match. The replication may be helpful in an untrusted network where we expect hosts may be compromised an produce intentionally incorrect results. In our system, the replication is by default disabled since we consider the network to be trusted. Moreover, the replication reduced the total computational power by 50% or more, as discussed in [5].

3.1.5. Assimilator

The assimilator processes each result received from clients. Basically, there are three options, how a workunit could end:

- **successful benchmark** of a node - the assimilator saves the node's cracking speed to a database;
- **finished regular job** - if one or more hashes are cracked, the assimilator updates the database. If all hashes are cracked, the entire job is considered done and all ongoing workunits are terminated. The assimilator also updates the progress of the entire cracking job. If the whole keyspace is processed, the job is considered finished;
- **computation error** - the assimilator performs the failure-recovery process as described in [5].

3.1.6. Trickler

In the original version of Fitcrack, the only information about computation progress the server had was from the result of each workunit. Since we wanted the administrator to have better overview of the system, we implemented a *Trickler* daemon, which handles BOINC *trickle messages* sent by a client. Using this technique, the clients periodically report their progress on each workunit which is then visualized in the *WebAdmin*.

3.1.7. BOINC daemons

Besides the previously denoted subsystems, we use the following deamons which are part of the BOINC:

- **Transitioner** - controls the state updates of workunits, either newly created, or finished.
- **Scheduler** - handles the requests of each client.
- **Feeder** - allocates blocks of shared memory for saving all workunit-related data.
- **File deleter** - deletes all unnecessary files remaning from previous workunits.

3.2. Client

While the architecture of the server is rather complex and it may take some effort to properly configure all subsystems, adding a new client to the cracking network is relatively easy.

Basically, the administrator only needs to install an application called *BOINC client* and connect to the server. The rest of client-side software is downloaded automatically from the server.

3.2.1. BOINC Client

BOINC Client also referred to as a *core client* is an application which handles the communication between the client and the server. The client actively asks the server for work and once a workunit assignment is received, it downloads all necessary input and output data. Besides that, the client also handles downloading and updating of all executable binaries required: *Runner* (see below), *hashcat*, and all hashcat's OpenCL kernels and files needed. Depending on how the BOINC client is installed, it can either run in the background like a daemon, or start when an individual user logs in; and is stopped when the user logs out.

3.2.2. BOINC Manager

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

3.2.3. Runner

Runner is a wrapper of hashcat, which we implemented to make hashcat usable with a BOINC-based system. The Runner is used for processing workunit assignments, controlling hashcat, and creating reports for the server. It provides abstraction of hashcat's paramaters for both regular, and benchmarking tasks. The runner optionally uses a local configuration file, where the user can specify, which OpenCL devices should be used for computation, and specify a workload profile for the devices.

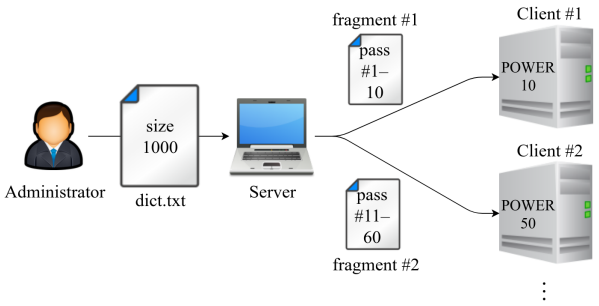

Figure 3: Example of dictionary attack distribution

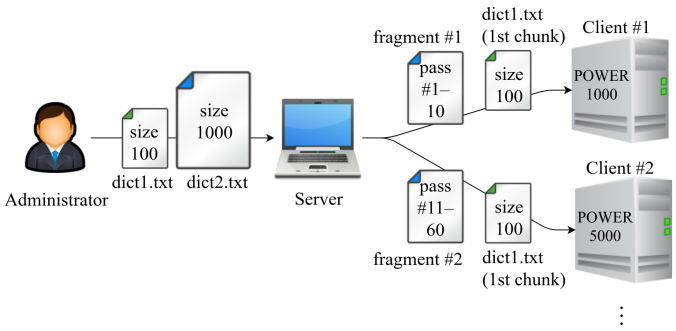

Figure 4: Example of combinator attack distribution

3.2.4. *hashcat*

As a cracking engine of the system, we use *hashcat* tool. It employs various OpenCL kernels that implement a GPGPU-based cracking of various hash algorithms. The reasons for choosing hashcat were described in section 1.

4. Attacks modes

Hashcat supports the following basic attack modes: dictionary attack, combinator attack, mask attack, and hybrid attacks. In the following subsections, we show how we perform these attacks in the distributed environment of BOINC.

4.1. Dictionary attack

The most straightforward way of cracking a password is a dictionary attack where a list of possible candidates is verified. While there are many ways of obtaining such lists, we always have to distribute these candidates from the server to our computing nodes. This makes distributed dictionary attacks inefficient when dealing with large dictionary files [7].

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

Therefore, a fragment of the original dictionary is created for each host with each workunit, whose 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 3.

4.2. Combinator attack

In combinator attack, the goal is to verify combinations of all passwords in two input dictionaries, using concatenation. Therefore, the number of possible candidates equals to $m \times n$, where m and n is the number of passwords in the first and second dictionary.

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

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

To deal with this issue, we came up with the following solution. The first dictionary is distributed as a whole to all computing nodes in the first workunit, also referred to as a chunk. Then, with each workunit, only a small portion of the second dictionary is being sent. This way, we can control the number of passwords we send in the second dictionary – n, while we can still limit the number of verified passwords in the first dictionary – m, using the hashcat mechanism. Also, we keep the linear complexity of the whole attack.

You can see a scheme of such an attack in Figure 4.

4.3. Mask 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 hashcat index range to verify is no longer a problem.

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

4.4. Hybrid attack

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 a mask on the left and a dictionary on the right side.

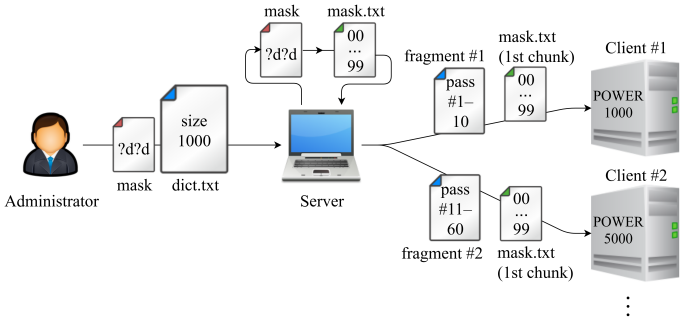

Figure 5: Example of hybrid attack distribution

When we look at the hybrid attack keyspace, it equals to $m \times n$, where m represents the size of the dictionary while n is the number of passwords generated by the mask. Similar to the combinator attack, hashcat does not provide us with the keyspace of the whole attack but with the size of the dictionary only. This means, when instructed to verify one password, in fact, hashcat checks one dictionary password combined with the whole mask.

The same solution as in combinator attack cannot be used, as there is no way to send just a portion of the mask to each host. To avoid generating all possible variants beforehand, which would cause the same problems described in the combinator attack above, we use the following technique. Dictionary is generated from the mask using high performance *maskprocessor*[20]. This means, we have two dictionaries on input and we can proceed with performing a combinator attack, as described above. The mask transformation process can be seen in Figure 5.

5. Results

To analyze the impact of the distribution strategies we use for different attack modes, we performed a series of experiments with hash cracking under various settings. In all cases, we measured the total time of cracking using Fitcrack, the total time of cracking using Hashtopolis tool, and the number of chunks assigned within each job. The first set of experiments analyzes the behavior of both systems during the dictionary attack using big wordlists. Other experiments aim at the brute-force attack where we studied the impact of chunk size and keyspace to overall cracking time. The goal is to determine how the systems behave under different conditions and identify the factors that influence the efficiency of an attack. During the measurements, we discovered several shortcomings of our design that slowed down the cracking process. Thus, we propose few additional improvements that help reduce the overhead.

5.1. Adaptive scheduling

At first, it is necessary to explain how our *adaptive scheduling algorithm* [5] affects the generation of workunits. Both Fitcrack, and Hashtopolis allow the user to specify the *chunk size* as the desired number of seconds required to solve each workunit. Based on the benchmark of hosts, performed at the start of each cracking job, both systems are trying to create fine-tailored workunits to fit the defined chunk size. While Hashtopolis strictly respects the user-entered chunk size, and uses the same tailoring mechanism from the very start till the end, Fitcrack utilizes another adjustments. The workunit Generator (see section 3.1.3) modifies the size of the next chunk by another two variables: the *elapsed time from start* (t_J), and the *remaining keyspace* (s_R). Based on the idea that the initial benchmark may not be accurate enough, we start with smaller workunits, and the full-sized workunits are not created before the elapsed time reaches the user-defined chunk size. Since we cannot predict the exact moments of workunit calculation, it is possible that in the end, some hosts would still compute while the rest would be idle. To avoid this negative phenomenon, we progressively shrink the size of workunits at the end of the job. This ensures that all hosts actively participate till the very end.

The practical impact of the algorithm is illustrated in Figure 6 which shows how keyspace of workunits changes over time. The chart was generated using a real brute-force attack on SHA-1 hash using 8 nodes with NVIDIA GTX 1050Ti, and password mask made of 10 lowercase letters (10x `?l`). The chunk size was set to 10 minutes.

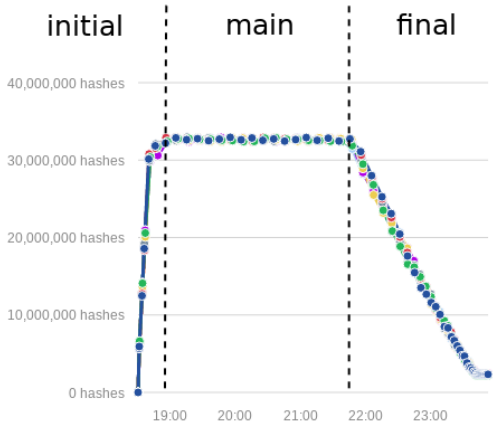

Figure 6: Illustration of adaptive workunit scheduling

The chart displays 8 different progresses with a different color for each node, however, we can see they precisely overlap. This is an anticipated result since all nodes had the same GPUs and no link outage, or computation error occured. The cracking took the interval of 5 hours and 35 minutes, and we can see the job progress chart is divided into three distinguishable parts. In the *initial* phase which took approximately first 10 minutes (which corresponds to the chunk size), the Generator did not create full-sized workunits. Next, the *main* phase displays workunits of the same size being assigned in 10-minute intervals, which correlates with the user-entered chunk size. Then, in the *final* phase, workunit sizes shrink progressively till the end of the job. The adaptive scheduling does not have a noticeable negative impact on the cracking process, as illustrated in Figure 7 which displays the job progress in time.

[20] `https://github.com/hashcat/maskprocessor`

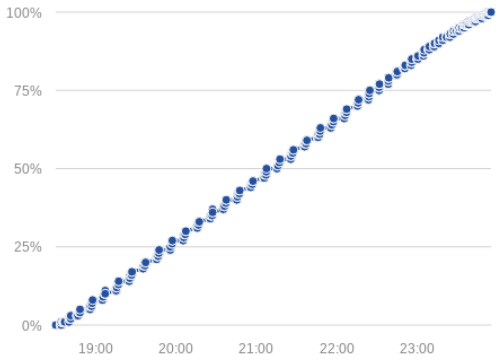

Figure 7: Job progress with adaptive scheduling

5.2. Used metrics

Since we compare our implementation with Hashtopolis, it was necessary to get equal conditions and fair metrics. To obtain comparable results, we measured the total time including the a) the benchmarking, b) data transfer, and c) the actual cracking. Fitcrack's total cracking time displayed in *WebAdmin* includes all these phases, while Hashtopolis only the last one. Another obstacle we had to solve was job starting.

Whereas Fitcrack allows to map all hosts to a job and start the process in a desired moment, Hashtopolis does not have a "start button". The cracking in Hashtopolis starts immediately once a *task* (equivalent to job) has any *agents* (equivalent to hosts) assigned to it, and the agents are *active*. In addition, Fitcrack supports sharing of hosts between jobs. If multiple jobs with the same host run simultaneously, the Generator uses a round-robin technique. In Hashtopolis, an agent may be assigned to only one task at a time. For experimental purposes, we created custom PHP-based scripts, which assign agents to the desired task, activate all agents at once, and remember the *activation time*. Once the task ends, it extracts the time of reporting the last chunk to the server, in Hashtopolis, displayed as a *last activity*. The total time is calculated as the interval between the activation time, and the last activity.

Another metric used is the number of *chunks* generated and assigned to hosts. While Hashtopolis performs the benchmarking separately from the actutal chunks, in Fitcrack, zero-keyspace workunits are used for that purpose. Thus, for comparison, we define that a *chunk* in Fitcrack equals to a non-benchmarking workunit.

5.3. Distributed dictionary attack

We wanted to verify that the segmentation of the dictionary described in section 4 brings the anticipated speedup over the naive solution where the entire dictionary is sent to all nodes. Thus, we performed a distributed dictionary attack using Fitcrack and Hashtopolis, where Hashtopolis distributes the entire dictionary to all nodes before the actual cracking starts. For the experiments, we used a network consisting of a server, and 8 nodes with NVIDIA GTX 1050 Ti, interconnected using a switch and 1 Gb/s Ethernet links. The distributed dictionary attack was done using four dictionaries with sizes from 1.1 GB to 8.3 GB, and the chunk size set to 60 seconds. The correct password was, in all cases, located at the end of each dictionary. The experimental results are shown in table 1, where for each attack, we show the total time, and the number of chunks generated. The hash algoritms used were SHA-1 which is easier-to-calculate, and Whirlpool which is more complex.

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

Table 1: Dictionary attack using 8 nodes, chunk size = 60s

With the first approach, marked as *Fitcrack-1*, the Generator created a single big chunk, therefore only one node from the entire network was used for the actual cracking. We searched for the cause of this phenomenon, and detected the classic benchmark of hashcat using `-b` option calculates much higher speeds than are possibly achieveable with a real attack.

To make a clear image, we did another series of tests using different GPUs (NVIDIA, AMD) and hash algorithms (MD5, SHA-1, SHA-512, and Whirlpool). We compared the benchmarked speed with the real speed of cracking with a brute-force attack using `7x?a` mask, and a dictionary attack using a 1.1 GB wordlist. The measured speeds are shown in table 2. Empty columns stand for OpenCL "CL_OUT_OF_RESOURCES" error which occured due to insufficient memory on the given GPU. We can see, even for a brute-force attack, the actual cracking is significantly slower than the benchmark reports. For dictionary attack, the measured speed is only a small fraction of the benchmark result. This makes sense since hashcat needs to load and cache dictionary passwords, making the cracking operations much slower.

We also decided to check why Hashtopolis' benchmarking is far more accurate, and searched through the sources. Instead of the classic benchmark mode of hashcat, Hashtopolis uses the `--speed-only` option which returns more accurate speed value, however, it requires to further specify the attack options, e.g. select a concrete dictionary for dictionary attack. This is possible since Hashtopolis transfers the entire dictionary to all nodes where it can be passed over to hashcat. Fitcrack, on the other hand, distributes dynamically crafted fragments of the original dictionary, and thus this technique is not usable.

Since the actual cracking speed with a dictionary attack is much lower, we added a simple heuristic: In a dictionary attack, if benchmarked speed is higher than 1 Mh/s, it is set to 1 Mh/s. This allows to craft initial chunks more precisely and still respects complex ciphers like BCrypt. Moreover, if the actual cracking speed is higher, it will be modified accordingly using the adaptive scheduling algorithm in the following workunits.

The results after adding the heuristic are displayed in table 1,

marked as *Fitcrack-2*. We can see, more chunks were created, and for bigger dictionaries, we achieved a significant speedup.

During the experiments, we detected Fitcrack's Generator (see section 3) is slowed down by the fragmentation of the original dictionary. The implementation forced Generator to open the dictionary file again in each iteration of the main loop [8], and sequentially skip initial passwords which were already processed. With the increasing job progress, the operation took still more and more processor time. Thus, we modified the Generator to remember the current position in the file, which reduced the processing overhead dramatically. Moreover, we came with an idea to decrease the communication overhead as well: During the cracking on GPUs, the CPU and the TCP/IP stack are more or less idle. Thus, we reconfigured BOINC server to assign up to two workunits to a host, and BOINC client to accept two workunits while only one is solved. The batch assignment allows each host to compute one workunit and download another at the same time. The results of the final implementation are displayed in table 1 as *Fitcrack-3*.

We depicted the results to graphs displaying the total time based on the dictionary size. Figure 8 shows the graph for SHA-1 algorithm, and we can see the course is similar to the Whirlpool case, displayed in Figure 9.

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

GPU	algorithm	bench [Mh/s]	brute [Mh/s]	dict [Mh/s]
NVIDIA GTX 1050 Ti	MD5	6310	2425	21
	SHA-1	2022	1540	9.1
	SHA-512	302	47	14
	Whirlpool	66	58	16
NVIDIA GTX 1080 Ti	MD5	35401.5	11267.3	29.4
	SHA-1	11872.3	7263.8	29.1
	SHA-512	1416.9	192.3	25.5
	Whirlpool	338.9	306.3	26.7
AMD Radeon RX 460	MD5	4186	1277	7
	SHA-1	1400	800	7.7
	SHA-512	155	41	4.56
	Whirlpool	-	-	-
AMD Radeon RX Vega 64	MD5	26479	8134	41.46
	SHA-1	9260	5664	45.92
	SHA-512	1212	751	40.49
	Whirlpool	699	332	39.57
AMD Radeon R9 Fury X	MD5	17752	5548	9.4
	SHA-1	17754	3785	9.3
	SHA-512	534	75	7.9
	Whirlpool	527	257	-

Table 2: Difference between benchmark and real attacks

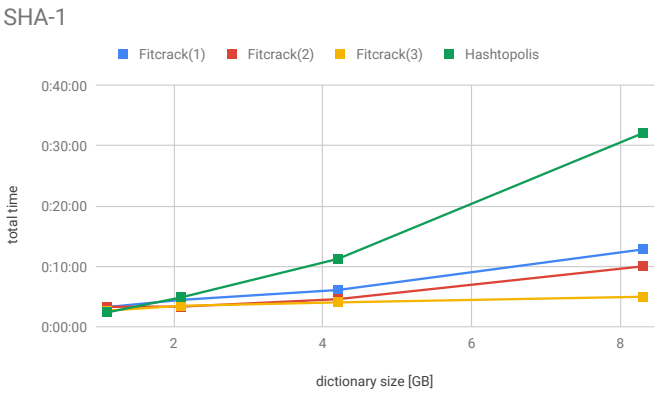

Figure 8: Total time of dictionary attack on SHA-1

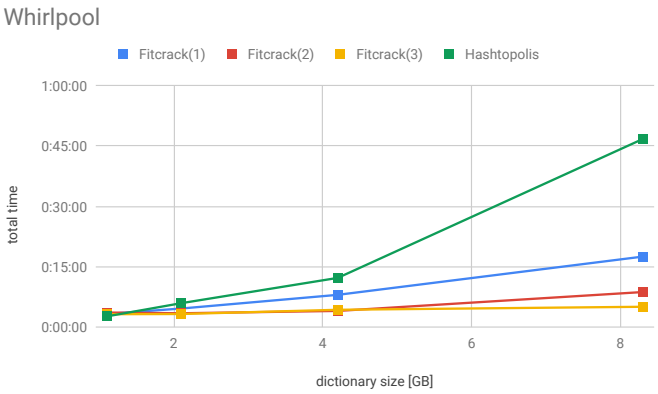

Figure 9: Total time of dictionary attack on Whirlpool

5.4. Distributed brute-force attack

While dictionary attacks may have high requirements for data transfers between nodes, the brute-force attack is not traffic-extensive at all. With each workunit, we only need to send hosts an attack configuration [8], and the range of password candidate indexes. To analyze, how our scheduling strategy behaves in comparison with Hashtopolis, we performed a series of brute-force attacks on SHA-1 using masks from 8 lowercase Latin letters (`?l?l?l?l?l?l?l?l`) to 10 letters (`?l?l?l?l?l?l?l?l?l?l`). The chunk size was set to the following values: 60s, 600s, 1200s, and 1800s. To get comparable results, we let both tools get through the entire keyspace. This was ensured by using two input hashes – one which was crackable using the mask, and other which was uncrackable. Table 3 shows the total time and the number of generated chunks, for all attacks.

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

Table 3: Brute-force attack on SHA-1 using 8 nodes, different chunk sizes

We can see again the behavior of our scheduling algorithm described in section 5.1. Due to the workunit shrinking in the initial and final phases, Fitcrack generated significantly more chunks than Hashtopolis. Obviously, this behavior is not contributive if we use very small chunks – for 60s, Hashtopolis was always faster. However, with higher chunk sizes, Fitcrack achieved better distribution. The workunit distribution progress for the attack with 600s chunks and 10-letter mask is displayed in Figure 6. In the main phase, the workunit assignment is standard, however, in the final phase, the count is increased, and the sizes are smaller – which ensures all nodes are utilized in every moment. Especially for chunk sizes of 1200s, and 1800s, the cracking times using Fitcrack were much lower. What also helped to make Fitcrack faster was the upgrade described in section 5.3 which almost eliminated the communication overhead, and the hosts were able to start next workunit immediately after the previous was finished.

5.5. Summary

When transferring big amounts of data, the distribution network and bandwidth of its links play an essential role in the overall efficiency of the cracking process. In dictionary attacks, we have to distribute the passwords to hosts somehow. Sending an entire dictionary to all hosts at the start is the most straightforward solution. In Hashtopolis, it allows estimating the cracking speed more precisely using the `--speed-only` option of hashcat. Since there is no broadcast in HTTP, the entire transfer has to be performed using unicast connections. As shown in the results, this works well for small wordlists but is not usable in general. The bigger the dictionary is, the higher is the initial overhead which is also multiplied by the number of hosts in the network. Dictionary segmentation used in Fitcrack requires additional logic on the server, prevents accurate estimation of cracking speed, but reduces the overhead rapidly since each password is transferred only once. Reducing the initial overhead shortens the dictionary transfer and allows to start cracking sooner. Using batch workunit assignment, we can eliminate the communication overhead of all attack modes almost entirely.

While in dictionary attacks, the password inputs of hashcat are processed, cached, and loaded to GPU, in a brute-force attack such operations are not required. The passwords are generated directly on the GPU, and thus it is possible to achieve much higher cracking speed. The network bandwidth is not limiting since we only transfer a range of password indexes together with additional options. The communication overhead can be, again, reduced by the batch workunit assignment. To utilize hardware resources well, it is, however, required to choose the keyspace of workunits wisely. Both Fitcrack and Hashtopolis calculate the keyspace from a cracking speed obtained by the benchmark and a user-defined chunk size. Hashtopolis preserves the similar keyspace to all workunits which, as we detected, leads to shorter cracking times of less-complex jobs, if the chunk size is set to a smaller value. Fitcrack, on the other hand, employs the adaptive scheduling algorithm which modifies the keyspace of workunits depending on the current progress. If the user-defined chunk size is big enough, the strategy used in Fitcrack helps reduce the total cracking time even if the total number of chunks is higher than in Hashtopolis.

6. Conclusion

We used BOINC framework to design an open-source hashcat-based distributed password cracking solution [8]. For each attack mode, we proposed a strategy which can be used for task distribution in different cracking networks including the ones which are non-homogenous, or dynamically changing. While Hashtopolis is by design more low-level and closer to hashcat, allowing the user to craft attack commands directly, Fitcrack provides higher level of abstraction and automation for the attacks.

The actual efficiency of an attack relies on many factors, where some can be controlled by the user. Such factors include the number of nodes, chunk size, and other attack options. We performed a modification which allowed nodes to work and download new workunits at the same time. This upgrade dramatically reduced BOINC's communication overhead, allowing the nodes to switch to new tasks almost instantly. For dictionary attack, we showed how the fragmentation strategy saves network bandwidth, and can lead to far better cracking times than in Hashtopolis. For brute-force attack, Fitcrack can bring great results if the user-defined chunk size is set wisely. We also proved that our adaptive scheduling algorithm [5] can, in many cases, provide better work distribution than the fixed chunk creation.

In the future, we want to add support for recently released hashcat 5.x and study the possibility of integrating John the Ripper. We would like to focus our research on enhancement of the distribution strategies we use, and add more automation to generating password-mangling rules[21] which can be used for modifying dictionary passwords.

References

[1] Anderson, D. P. (2004). Boinc: a system for public-resource computing and storage. In *Fifth IEEE/ACM International Workshop on Grid Computing* (pp. 4–10).

[2] Apostal, D., Foerster, K., Chatterjee, A., & Desell, T. (2012). Password recovery using MPI and CUDA. In *Proc. of HiPS 2012* (pp. 1–9).

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

[3] Bengtsson, J. (2007). Parallel password cracker: A feasibility study of using linux clustering technique in computer forensics. In *Workshop on Digital Forensics and Incident Analysis, International(WDFIA)* (pp. 75–82). volume 00.

[4] Crumpacker, J. R. (2009). *Distributed password cracking*. Ph.D. thesis Monterey, California. Naval Postgraduate School.

[5] Hranický, R., Holkovič, M., Matoušek, P., & Ryšavý, O. (2016). On efficiency of distributed password recovery. *The Journal of Digital Forensics, Security and Law*, *11*, 79–96.

[6] Hranický, R., Matoušek, P., Ryšavý, O., & Veselý, V. (2016). Experimental evaluation of password recovery in encrypted documents. In *Proceedings of ICISSP 2016* (pp. 299–306). SciTePress - Science and Technology Publications.

[7] Hranický, R., Zobal, L., Večeřa, V., & Matoušek, P. (2017). Distributed password cracking in a hybrid environment. In *Proceedings of SPI 2017* (pp. 75–90). University of defence in Brno.

[8] Hranický, R., Zobal, L., Večeřa, V., & Múčka, M. (2018). *The architecture of Fitcrack distributed password cracking system*. Technical Report.

[9] Kasabov, A., & van Kerkwijk, J. (2011). Distributed gpu password cracking. *Universiteit Van Amsterdam*, .

[10] Kim, K. (2012). Distributed password cracking on gpu nodes. In *2012 7th International Conference on Computing and Convergence Technology (ICCCT)* (pp. 647–650).

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

[12] Marks, M., & Niewiadomska-Szynkiewicz, E. (2014). Hybrid cpu/gpu platform for high performance computing. In *Proc. of the 28th ECMS* (pp. 523–537).

[13] McAtee, M., & Morris, L. (2015). CrackLord: Maximizing Computing Resources. In *BlackHat USA 2015*. Mandalay Bay, Las Vegas NV, USA: Crowe Horwath LLP.

[14] Narayanan, A., & Shmatikov, V. (2005). Fast dictionary attacks on passwords using time-space tradeoff. In *Proceedings of the 12th ACM Conference on Computer and Communications Security* CCS '05 (pp. 364–372). New York, NY, USA: ACM.

[15] Pavlov, D., & Veerman, G. (2012). *Distributed Password Cracking Platform*. Technical Report.

[16] Pippin, A., Hall, B., & Chen, W. (2006). *Parallelization of John the Ripper Using MPI (Final Report)*. Technical Report.

[17] Samek, J. (2016). *Virtul GPU cluster*. Bachelor's thesis. Faculty of Information Technology. Czech technical university in Prague.

[18] Yang, C.-T., Huang, C.-L., & Lin, C.-F. (2011). Hybrid cuda, openmp, and mpi parallel programming on multicore gpu clusters. *Computer Physics Communications*, *182*, 266–269.

[19] Zonenberg, A. (2009). Distributed hash cracker: A cross-platform gpu-accelerated password recovery system, .

Acknowledgements

Research presented in this paper is supported by "Integrated platform for analysis of digital data from security incidents" project, no. VI20172020062 granted by Ministry of the Interior of the Czech Republic and "ICT tools, methods and technologies for smart cities" project, no. FIT-S-17-3964 granted by Brno University of Technology.

The work is also supported by Ministry of Education, Youth and Sports of the Czech Republic from the National Programme of Sustainability (NPU II) project "IT4Innovations excellence in science" LQ1602.

We would like to thank our fellow developers Vojtěch Večeřa and Matúš Múčka for all the work on Fitcrack system, and Petr Matoušek for supervising our research project.

The final publication is available at Elsevier via `https://doi.org/10.1016/j.diin.2019.08.001`.