Distributed PCFG Password Cracking

Radek Hranický, Lukáš Zobal, Ondřej Ryšavý, Dušan Kolář, and Dávid Mikuš

Faculty of Information Technology, Brno University of Technology, Czech Republic
{ihranicky,izobal,rysavy,kolar}@fit.vutbr.cz, d.mikus@gmail.com

Abstract. In digital forensics, investigators frequently face cryptographic protection that prevents access to potentially significant evidence. Since users prefer passwords that are easy to remember, they often unwittingly follow a series of common password-creation patterns. A probabilistic context-free grammar is a mathematical model that can describe such patterns and provide a smart alternative for traditional brute-force and dictionary password guessing methods. Because more complex tasks require dividing the workload among multiple nodes, in the paper, we propose a technique for distributed cracking with probabilistic grammars. The idea is to distribute partially-generated sentential forms, which reduces the amount of data necessary to transfer through the network. By performing a series of practical experiments, we compare the technique with a naive solution and show that the proposed method is superior in many use-cases.

Keywords: Distributed · Password · Cracking · Forensics · Grammar

1 Introduction

With the complexity of today's algorithms, it is often impossible to crack a hash of a stronger password in an acceptable time. For instance, verifying a single password for a document created in MS Office 2013 or newer requires 100,000 iterations of SHA-512 algorithm. Even with the use of the popular hashcat[1] tool and a machine with 11 NVIDIA GTX 1080 Ti[2] units, brute-forcing an 8-character alphanumeric password may take over 48 years. Even the most critical pieces of forensic evidence lose value over such a time.

Over the years, the use of probability and statistics showed the potential for a rapid improvement of attacks against human-created passwords [12,13,19]. Various leaks of credentials from websites and services provide an essential source of knowledge about user password creation habits [2,20], including the use of existing words [4] or reusing the same credentials between multiple services [3]. Therefore, the ever-present users' effort to simplify work is also their major weakness. People across the world unwittingly follow common password-creation patterns over and over.

[1] https://hashcat.net/

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

Weir et al. showed how *probabilistic context-free grammars* (PCFG) [19] could describe such patterns. They proposed a technique for automated creation of probabilistic grammars from existing password datasets that serve as *training dictionaries*. A grammar is a mathematical model that allows representing the structure of a password as a composition of fragments. Each fragment is a finite sequence of letters, digits, or special characters. Then, by derivation using rewriting rules of the grammar, one can not only generate all passwords from the original dictionary but produce many new ones that still respect password-creation patterns learned from the dictionary.

The rewriting rules of PCFG have probability values assigned accordingly to the occurrence of fragments in the training dictionary. The probability of each possible password equals the product of probabilities of all rewriting rules used to generate it. Generating password guesses from PCFGs is deterministic and is performed in an order defined by their probabilities. Therefore, more probable passwords are generated first, which helps with a precise targeting of an attack.

Since today's challenges in password cracking often require distributed computing [11], it is necessary to find appropriate mechanisms for PCFG-based password generators. From the entire process, the most complex part that needs distribution is the calculation of password hashes. It is always possible to generate all password guesses on a single node and distribute them to others that perform the hash calculation. However, as we detected, the generating node and the interconnecting network may quickly become a bottleneck. Inspired by Weir's work with preterminal structures [19] and our previously published parallel PCFG cracker [10], we present a distributed solution that only distributes "partially-generated" passwords, and the computing nodes themselves generate the final guesses.

1.1 Contribution

We propose a mechanism for distributed password cracking using probabilistic context-free grammars. The concept uses preterminal structures as basic units for creating work chunks. In the paper, we demonstrate the idea by designing a proof-of-concept tool that also natively supports the deployment of the hashcat tool. Our solution uses adaptive work scheduling to reflect the performance of available computing nodes. We evaluate the technique in a series of experiments by cracking different hash algorithms using different PCFGs, network speeds, and numbers of computing nodes. By comparison with the naive solution, we illustrate the advantages of our concept.

1.2 Structure of the paper

The paper is structured as follows. Section 2 provides a summary of related work. Section 3 describes the design of our distributed PCFG Manager. Section 4 shows experimental results of our work. Finally, Section 5 concludes the paper.

2 Background and related work

The use of probabilistic methods for computer-based password cracking dates back to 1980. Martin Hellman introduced a time-memory trade-off method for cracking DES cipher [6]. This chosen-plaintext attack used a precomputation of data stored in memory to reduce the time required to find the encryption key. With a method of distinguished points, Ron Rivest reduced the necessary amount of lookup operations [16]. Phillipe Oechslin improved the original concept and invented rainbow tables as a compromise between the brute-force attack and a simple lookup table. For the cost of space and precomputation, the rainbow table attack reduces the cracking time of non-salted hashes dramatically [14].

The origin of passwords provides another hint. Whereas a machine-generated password may be more or less random, human beings follow specific patterns we can describe mathematically. Markov chains are stochastic models frequently used in natural language processing [15]. Narayanan et al. showed the profit of using zero-order and first-order Markovian models based on the phonetical similarity of passwords to existing words [13]. Hashcat cracking tool utilizes this concept in the default configuration of a mask brute-force attack. The password generator uses a Markov model supplied in an external .hcstat file. Moreover, the authors provide a utility for the creation of new models by automated processing of character sequences from existing wordlists.

Weir et al. introduced password cracking using probabilistic context-free grammars (PCFG) [19]. The mathematical model is based on classic context-free grammars [5] with the only difference that each rewriting rule has a probability value assigned. Similarly to Markovian models, a grammar can be created automatically by training on an existing password dictionary. For generating passwords guesses from an existing grammar, Weir proposed the Next function together with a proof of its correctness. The idea profits from the existence of pre-terminal structures - sentential forms that produce password guesses with the same probability. By using PCFG on MySpace dataset (split to training and testing part), Weir et al. were able to crack 28% to 128% more passwords in comparison with the default ruleset from John the Ripper (JtR) tool[3] using the same number of guesses [19]. The original approach did not distinguish between lowercase and uppercase letter. Thus, Weir extended the original concept by adding special rules for capitalization of letter fragments. Due to high space complexity, the original Next function was later replaced by the Deadbeat dad algorithm [18].

Through the following years, Weir's work inspired other researchers as well. Veras et al. proposed a semantic-based extension of PCFGs that makes the provision of the actual meaning of words that create passwords [17]. Ma et al. performed a large-scale evaluation of different probabilistic password models and proposed the use of normalization and smoothing to improve the success rate [12]. Houshmand et al. extended Weir's concept by adding an extra set of rules that respect the position of keys on keyboards to reflect frequent patterns that

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

people use. The extension helped improve the success rate by up to 22%. Besides, they advised to use Laplace probability smoothing, and created guidelines for choosing appropriate attack dictionaries [8]. After that, Houshmand et al. also introduced targeted grammars that utilize information about a user who created the password [7]. Last but not least, Agarwall et al. published a well-arranged overview of new technologies in password cracking techniques, including the state-of-the-art improvements in the area of PCFGs and Markovian models [1].

In our previous research, we focused on the practical aspects of grammar-based attacks and identified factors that influence the time of making password guesses. We introduced parallel PCFG cracking and possibilities for additional filtering of an existing grammar. Especially, removing rules that rewrite the starting non-terminal to long base structures lead to a massive speedup of password guessing with nearly no impact on success rate [10]. In 2019, Weir released[4] a compiled PCFG password guesser written in pure C to achieve faster cracking and easier interconnection with existing tools like hashcat and JtR.

Making the password guessing faster, however, resolves only a part of the problem. Serious cracking tasks often require the use of distributed computing. But how to efficiently deliver the password guesses to different nodes? Weir et al. suggested the possible use of preterminal structures directly in a distributed password cracking trial [19]. To verify the idea, we decided to narrowly analyze the possibilities for distributed PCFG guessing, create a concrete design of intra-node communication mechanisms, and experimentally test its usability.

3 Distributed PCFG Manager

For distributed cracking, we assume a network consisting of a server and a set of clients, as illustrated in Figure 1. The server is responsible for handling client requests and assigning work. Clients represent the cracking stations equipped with one or more OpenCL-compatible devices like GPU, hardware coprocessors, etc. In our proposal, we talk about a client-server architecture since the clients are actively asking for work, whereas the server is offering a "work assignment service."

In PCFG-based attacks, a probabilistic context-free grammar represents the source of all password guesses, also referred to as candidate passwords. Each guess represents a string generated by the grammar, also known as a *terminal structure* [18,19]. In a distributed environment, we need to deliver the passwords to the cracking nodes somehow. A naive solution is to generate all password candidates on the server and distribute them to clients. However, such a method has high requirements on the network bandwidth, and as we detected in our previous research, also high memory requirements to the server [10]. Another drawback of the naive solution is limited scalability. Since all the passwords are generated on a single node, the server may easily become a bottleneck of the entire network.

[4] `https://github.com/lakiw/compiled-pcfg`

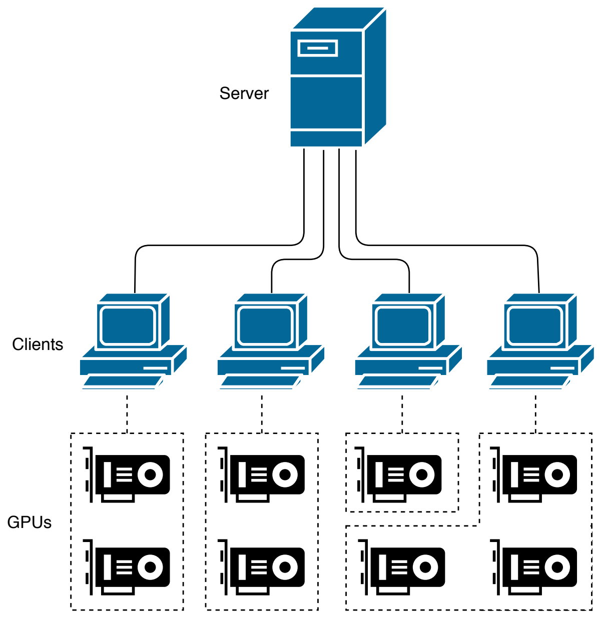

Fig. 1: An example of a cracking network

And thus, we propose a new distributed solution that is inspired by our parallel one [10]. In the following paragraphs, we describe the design and communication protocol of our distributed PCFG Manager. To verify the usability of our concept, we created a proof-of-concept tool that is freely available[5] on GitHub. For implementation, we chose Go[6] programming language, because of its speed, simplicity, and compilation to machine language. The tool can run either as a server or as a client. The PCFG Manager server should be deployed on the server node. It is responsible for processing the input grammar and distribution of work. The PCFG Manager client running on the client nodes generates the actual password guesses. It either prints the passwords directly to the standard output or passes them to an existing hash cracker. The behavior depends on the mode of operation specified by the user. Details are explained in the following paragraphs.

The general idea is to divide the password generation across the computing nodes. The server only generates the preterminal structures (PT), while the terminal structures (T) are produced by the cracking nodes. The work is assigned progressively in smaller pieces called chunks. Each *chunk* produced by the server contains one or more preterminal structures, from which the clients generate the password guesses. To every created chunk, the server assigns a unique identifier called the *sequence number*. The *keyspace*, i.e., the number of possible passwords, of each chunk is calculated adaptively to fit the computational capabilities of a node that will be processing it. Besides that, our design allows direct cracking with hashcat tool. We chose hashcat as a cracking engine for the same reasons

[5] `https://github.com/nesfit/pcfg-manager`
[6] `https://golang.org/`

as we did for the Fitcrack distributed password cracking system [11], mainly because of its speed and range of supported hash formats. The proposed tool supports two different modes of operation:

– **Generating mode** - the PCFG Manager client generates all possible password guesses and prints them to the standard output. A user can choose to save them into a password dictionary for later use or to pass them to another process on the client-side.
– **Cracking mode** - With each chunk, the PCFG Manager client runs hashcat in stdin mode with the specified hashlist and hash algoritm. By using a pipe, it feeds it with all password guesses generatable from the chunk. Once hashcat processes all possible guesses, the PCFG Manager client returns a result of the cracking process, specifying which hashes were cracked within the chunk and what passwords were found.

3.1 Communication protocol

The proposed solution uses remote procedure calls with the gRPC[7] framework. For describing the structure of transferred data and automated serialization of payload, we use the Protocol buffers[8] technology.

The server listens on a specified port and handles requests from client nodes. The behavior is similar to the function of Gouroutine M from the parallel version [10] - it generates PT and tailors workunits for client nodes. Each workunit, called chunk, contains one or more PTs. As shown in listing 1.1, the server provides clients an API consisting of four methods. Listing 1.2 shows an overview of input/output messages that are transferred with the calls of API methods.

```
service PCFG {
    rpc Connect (Empty) returns (ConnectResponse) {}
    rpc Disconnect(Empty) returns (Empty);
    rpc GetNextItems(Empty) returns (Items) {}
    rpc SendResult(CrackingResponse) returns (↩
    ResultResponse);
}
```

Listing 1.1: Server API

When a client node starts, it connects to the server using the `Connect()` method. The server responds with the `ConnectResponse` message containing a PCFG in a compact serialized form. If the desire is to perform an attack on a concrete list of hashes using hashcat, the `ConnectResponse` message also contains a *hashlist* (the list of hashes intended to crack) and a number that defines the *hash mode*[9], i.e., cryptographic algorithms used.

[7] https://grpc.io/
[8] https://developers.google.com/protocol-buffers
[9] https://hashcat.net/wiki/doku.php?id=example_hashes

```
message ConnectResponse {
    Grammar grammar = 1;
    repeated string hashList = 2;
    string hashcatMode = 3;
}
message Items {
  repeated TreeItem preTerminals = 1;
}
message ResultResponse {
  bool end = 1;
}
message CrackingResponse {
  map<string, string> hashes = 1;
}
```

Listing 1.2: Messages transferred between the server and clients

Once connected, the client asks for a new chunk of preterminal structures using the `GetNextItems()` method. In response, the server assigns the client a chunk of 1 to N preterminal structures, represented by the `Items` message. After the client generates and processes all possible passwords from the chunk, using the `SendResult()` call, it submits the result in the `CrackingResponse` message. In cracking mode, the message contains a map (an associative array) of cracked hashes together with corresponding plaintext passwords. If no hash is cracked within the chunk or if the PCFG Manager runs in generating mode, the map is empty. With the `ResultResponse` message, the server then informs the client, if the cracking is over or if the client should ask for a new chunk by calling the `GetNextItems()` method.

The last message is `Disconnect()` that clients use to indicate the end of their participation, so that the server can react adequately. For instance, if a client had a chunk assigned, but disconnected without calling the `SendResult()` method, the server may reassign the chunk to a different client. The flow of messages between the server and a client is illustrated in Figure 2.

3.2 Server

The server represents the controlling point of the computation network. It maintains the following essential data structures:

- **Priority queue** - the queue is used by the *Deadbeat dad* algorithm [18] for generating preterminal structures.
- **Buffered channel** - the channel represents a memory buffer for storing already generated PTs from which the server creates chunks of work.
- **Client information** - for each connected client, the server maintains its IP address, current performance, the total number of password guesses performed by the client, and information about the last chunk that the client

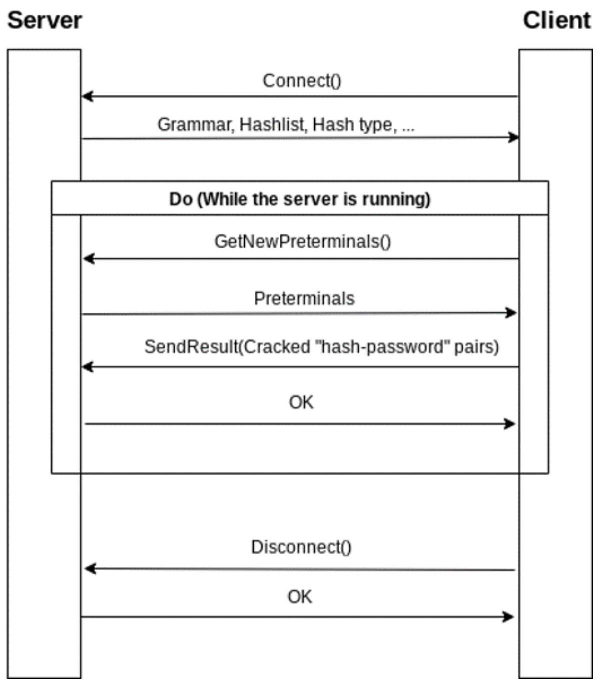
Server
Client
Connect()
Grammar, Hashlist, Hash type, ...
Do (While the server is running)
GetNewPreterminals()
Preterminals
SendResult(Cracked "hash-password" pairs)
OK
Disconnect()
OK

Fig. 2: The proposed communication protocol

completed: its keyspace and timestamps describing when the processing started and ended. If the client has a chunk assigned, the server also stores its sequence number, PTs, and the keyspace of the chunk.

- **List of incomplete chunks** - the structure is essential for a failure-recovery mechanism we added to the server. If any client with a chunk assigned disconnects before reporting its result, the chunk is added to the list to be reassigned to a different client.
- **List of non-cracked hashes** (cracking mode only) - the list contains all input hashes that have not been cracked yet.
- **List of cracked hashes** (cracking mode only) - The list contains all hashes that have already been cracked, together with corresponding passwords.

Once started, the server loads an input grammar in the format used by Weir's PCFG Trainer[10]. Next, it checks the desired mode of operation and other configuration options – the complete description is available via the tool's help. In cracking mode, the server loads all input hashes to the list of non-cracked hashes. In generating mode, all hashlists remain empty. The server then allocates memory for the buffered channel, where the channel size can be specified by the user.

As soon as all necessary data structures get initialized, the server starts to generate PTs using the *Deadbeat dad* algorithm, and with each PT, it calculates

[10] `https://github.com/lakiw/legacy-pcfg/blob/master/python_pcfg_cracker_version3/pcfg_trainer.py`

and stores its keyspace. Generated PTs are sent to the buffered channel. The process continues as long as there is free space in the channel, and the grammar allows new PTs to be created. If the buffer infills, generating new PTs is suspended until the positions in the channel get free again.

When a client connects, the server adds a new record to the client information structure. In the `ConnectResponse` message, the client receives the grammar that should be used for generating passwords guesses. In cracking mode, the server also sends the hashlist and hash mode identifying the algorithms that should be used, as illustrated in Figure 2.

Upon receiving the `GetNextItems()` call, the server pops one or more PTs from the buffered channel and sends them to the client as a new chunk. Besides, the server updates the client information structure to denote what chunk is currently assigned to the client. The number of PTs taken depends on their keyspace. Inspired by our adaptive scheduling algorithm that we Like in Fitcrack [9, 11], the system schedules work adaptively to the performance of each client.

In our previous research, we introduced an algorithm for adaptive task scheduling. We integrated the algorithm into our proof-of-concept tool, Fitcrack[11] - a distributed password cracking system, and showed its benefits [9, 11]. Therefore, we decided to use a similar technique and schedule work adaptively to each client's performance. In our distributed PCFG Manager, the performance of a client (p_c) in passwords per second is calculated from the keyspace (k_{last}) and computing time (Δt_{last}) of the last assigned chunk. The keyspace of a new chunk (k_{new}) assigned to the client depends on the client's performance and the `chunk_duration` parameter that the user can specify:

$$p_c = \frac{k_{last}}{\Delta t_{last}}, \tag{1}$$

$$k_{new} = p_c * chunk\_duration. \tag{2}$$

The server removes as many PTs from the channel as needed to make the total keyspace of the new chunk at least equal to k_{new}. If the client has not solved any chunk yet, we have no clue to find p_c. Therefore, for the very first chunk, the k_{new} is set to a pre-defined constant.

An exception occurs if a client with a chunk assigned disconnects before reporting its result. In such a case, the server saves the assignment to the list of incomplete chunks. If the list is not empty, chunks in it have an absolute priority over the newly created ones. And thus, upon the following `GetNextItems()` call from any client, the server uses the previously stored chunk.

Once a client submits a result via the `SendResult()` call, the server updates the information about the last completed chunk inside the client information structure. For each cracked hash, the server removes it from the list of non-cracked hashes and adds it to the list of cracked hashes together with the resulting password. If all hashes are cracked, the server prints each of them with the correct password and ends. In generating mode, the server continues as long as

[11] `https://fitcrack.fit.vutbr.cz`

new password guesses are possible. The same happens in the cracking mode in case there is a non-cracked hash.

Finally, if a client calls the `Disconnect()` method, the server removes its record from the client information structure. As described above, if the client had a chunk assigned, the server will save it for later use.

3.3 Client

After calling the `Connect()` method, a client receives the `ConnectResponse` message containing a grammar. In cracking mode, the message also include a hashlist and a hash mode. Then it calls the `GetNextItems()` method to obtain a chunk assigned by the server. Like in our parallel version, the client then subsequently takes one PT after another and uses the generating goroutines to create passwords from them [10]. In the generating mode, all password guesses are printed to the standard output.

For the cracking mode, it is necessary to have a compiled executable of hashcat on the client node. The user can define the path using the program parameters. The PCFG Manager then starts hashcat in the *dictionary attack* mode with the hashlist and hash mode parameters based on the information obtained from the `ConnectResponse` message. Since no dictionary is specified, hashcat automatically reads passwords from the standard input. And thus, the client creates a *pipe* with the end connected to the hashcat's input. All password guesses are sent to the pipe. From each password, hashcat calculates a cryptographic hash and compares it to the hashes in the hashlist. After generating all password guesses within the chunk, the client closes the pipe, waits for hashcat to end, and reads its return value. On success, the client loads the cracked hashes.

In the end, the client informs the server using the `SendResult()` call. If any hashes are cracked, the client adds them to the `CrackingResponse` message that is sent with the call. The architecture of the PCFG Manager client is displayed in Figure 3.

4 Experimental results

We conduct a number of experiments in order to validate several assumptions. First, we want to show the proposed solution results in a higher cracking performance and lower network usage. We also show that while the naive terminal distribution quickly reaches the speed limit by filling the network bandwidth, our solution scales well across multiple nodes. We discuss the differences among different grammars and the impact of scrambling the chunks during the computation. In our experiments, we use up to 16 computing nodes for the cracking tasks and one server node distributing the chunks. All nodes have the following configuration:

- CentOS 7.7 operating system,
- NVIDIA GeForce GTX1050 Ti GPU,
- Intel(R) Core(TM) i5-3570K CPU, and 8GB RAM.

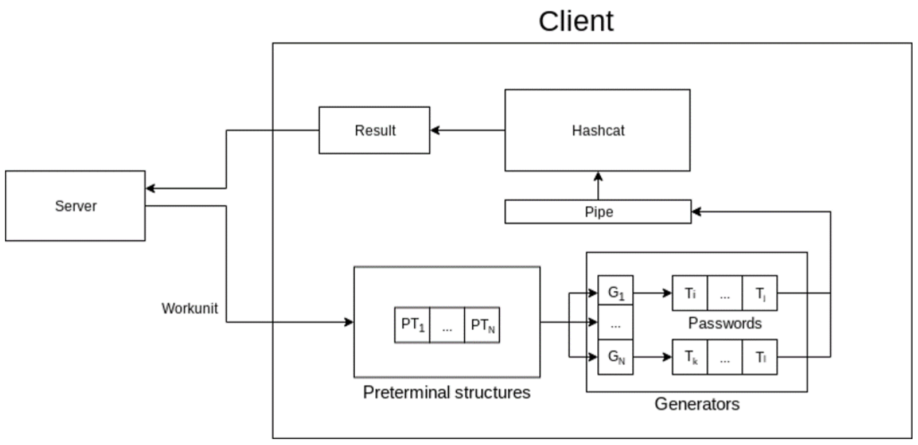

Fig. 3: The architecture of the client side

The nodes are in a local area network connected with links of 10, 100, and 1000 Mbps bandwidth. During the experiments, we incrementally change the network speed to observe the changes. Furthermore, we limit the number of generated passwords to 1, 10, and 100 million.

With this setup, we conduct a number of cracking tasks on different hash types and grammars. As the hash cracking speed has a significant impact on results, we chose bcrypt with five iterations, a computationally difficult hash type, and SHA3-512, an easier, yet modern hash algorithm. Table 1 displays all chosen grammars with description. The columns cover statistics of the source dictionary: password count (pw-cnt) and average password length (avg-len), as well as statistics of the generated grammar: the number of possible passwords guesses (pw-cnt), the number of base structures (base-cnt), their average length (avg-base-len) and the maximum length of base structures (max-base-len), in nonterminals. One can also notice the enormous number of generated passwords, especially with the myspace grammar. Such a high number is caused only by few base structures with many nonterminals. We discussed the complexity added by long base structure in our previous research [10]. If not stated otherwise, the grammars are generated from password lists found on SkullSecurity wiki page[12]. For each combination of described parameters, we run two experiments – first, with the naive terminal distribution (terminal), and second using our solution with the preterminal distribution (preterminals).

4.1 Computation Speedup and Scaling

The primary goal is to show that the proposed solution provides faster password cracking using PCFG. In Figure 4, one can see the average cracking speed of

[12] `https://wiki.skullsecurity.org/Passwords`
[13] `https://github.com/danielmiessler/SecLists/blob/master/Passwords/darkweb2017-top10000.txt`

Dictionary Statistics			PCFG Statistics			
name	**pw-cnt**	**avg-len**	**pw-cnt**	**base-cnt**	**avg-base-len**	**max-base-len**
myspace	37,145	8.59	6E+1874	1,788	4.50	600
cain	306,707	9.27	3.17E+15	167	2.59	8
john	3,108	6.06	1.32E+09	72	2.14	8
phpbb	184,390	7.54	2.84E+37	3,131	4.11	16
singles	12,235	7.74	6.67E+11	227	3.07	8
dw17[13]	10,000	7.26	2.92E+15	106	2.40	12

Table 1: Grammars used in the experiments

SHA3-512 hash with *myspace* grammar, with different task sizes and network bandwidths.

Apart from the proposed solution being generally faster, we see a significant difference in speeds with the lower network bandwidths. This is well seen in the detailed graph in Figure 5 which shows the cracking speed of SHA3-512 in a 10Mbps network with different task sizes. The impact of the network bandwidth limit is expected as the naive terminal distribution requires a significant amount of data in the form of a dictionary to be transmitted. In the naive solution, network links become the main bottleneck that prevents achieving higher cracking performance. In our solution, the preterminal distribution reduces data transfers dramatically, which removes the obstacle and allow for achieving higher cracking speeds.

Apart from SHA3-512, we conducted the same set of experiments with SHA1 and MD5 hashes. Results from cracking these hashes are not present in the paper as they are very similar to SHA3. The reason for this is the cracking performance of SHA1 and MD5 is very high, similar to the former.

Figure 6 illustrates the network activity using both solutions. The graph compares the data transfered in time in SHA3-512 cracking task with the lowest limit on network bandwidth. With the naive terminal distribution one can notice the transfer speed is limited for the entire experiment with small pauses for generating the terminals. As a result, the total amount of data transferred is more than 15 times bigger than with the proposed preterminal distribution. The maximum speed has not reached the 10 Mbit limit due to auto-negotiation problems on the local network.

The difference between the two solutions disappears if we crack very complex hash algorithms. Figure 7 shows the results of cracking bcrypt hashes with *myspace* grammar. The average cracking speeds are multiple times lower than with SHA3. In this case, the two solutions do not differ because the transferred chunks have much lower keyspace since clients can not verify as many hashes as was possible for SHA3. Most of the experiment time is used by hashcat itself, cracking the hashes.

In the previous graphs, one could also notice the cracking speed increases with more hashes. This happens since smaller tasks cannot fully leverage the whole distributed network as the smallest task took only several seconds to crack.

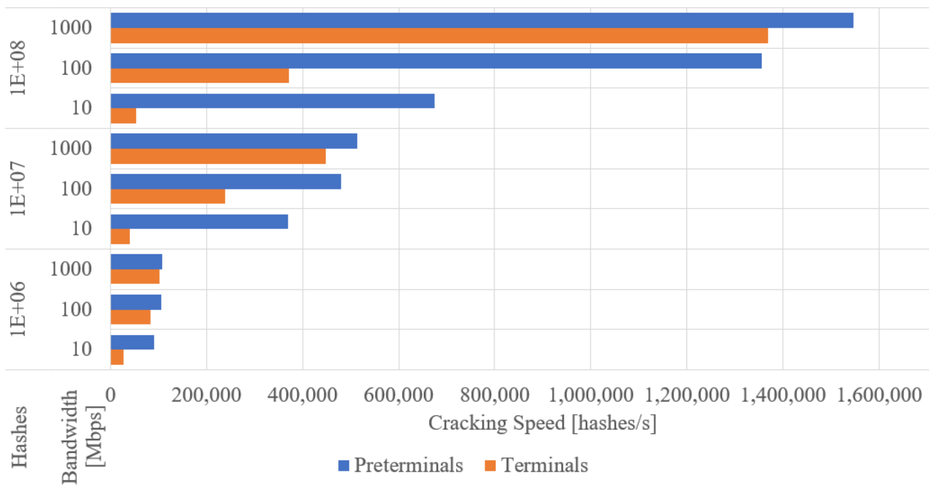

Fig. 4: Average cracking speed with different bandwidths and password count (SHA3-512 / *myspace* grammar / 4 nodes)

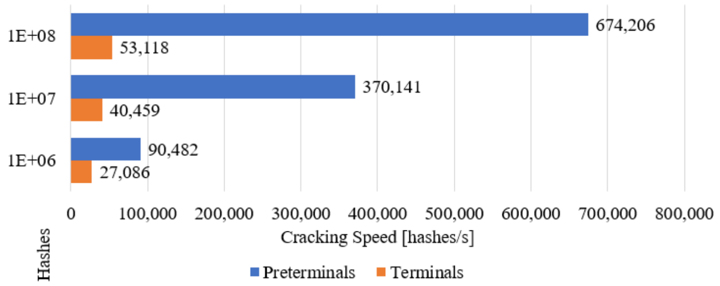

Fig. 5: Detail of 10Mbps network bandwidth experiment (SHA3-512 / *myspace* grammar / 4 nodes)

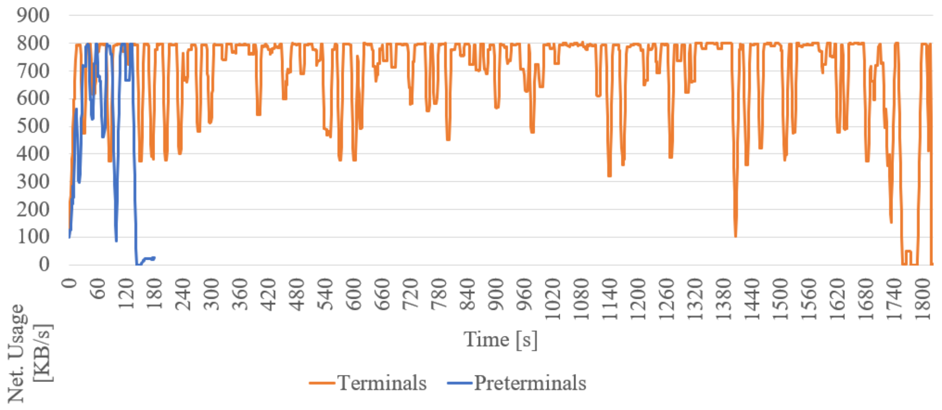

Fig. 6: Comparison of network activity (SHA3-512 / *myspace* grammar / 100 million hashes / 10 Mbps network bandwidth / 16 nodes)

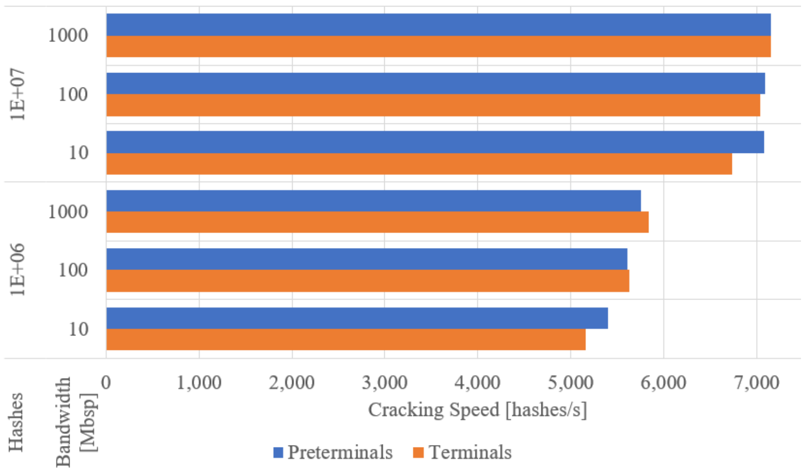

Fig. 7: Average cracking speed with different bandwidths and password count (bcrypt / *myspace* grammar)

Figure 8 shows that the average cracking speed is influenced by the number of connected cracking nodes. While for the smallest task, there is almost no difference with the increasing node count, for the largest task, the speed rises even between 8 and 16 nodes. As this task only takes several minutes, we expect larger tasks would visualize this even better. One can also observe the naive solution using terminal distribution does not scale well. Even though we notice a slight speedup up to 4 nodes. The speed is capped after that even in the largest task because of the network bottleneck mentioned above.

Figure 9 shows a similar picture but now with the network bandwidth cap increased from 100 Mbps to 1000 Mbps. While the described patterns remain visible, the difference between our and the naive solutions narrows. The reason for this is with the increased bandwidth, it is possible to send greater chunks of pre-generated dictionaries, as described above. With the increasing number of nodes and cracking task length, advantages of the proposed solution become more clear, as seen at the top of the described graph.

4.2 Grammar Differences

Next, we measure how the choice of a grammar influences the cracking speed. In Figure 10, we can see the differences are significant. While the cracking speed of the naive solution is capped by the network bandwidth, results from the proposed solution show generating passwords using some grammars is slower than with others – a phenomenon that is connected with the base structures lengths [10].

Generating passwords from the *Darkweb2017* (dw17) grammar is also very memory demanding because of the long base structures at the beginning of the grammar, and 8 GB RAM is not enough for the largest cracking task using the naive solution. With the proposed preterminal-based solution, we encounter no such problem.

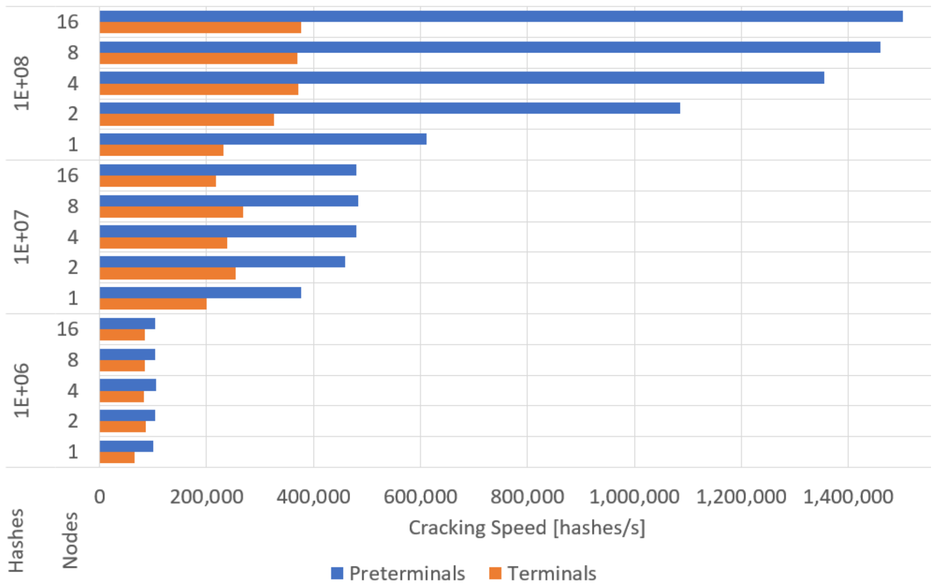

Fig. 8: Scaling across multiple cracking nodes (SHA3-512 / *myspace* grammar / 100 Mbps network bandwidth)

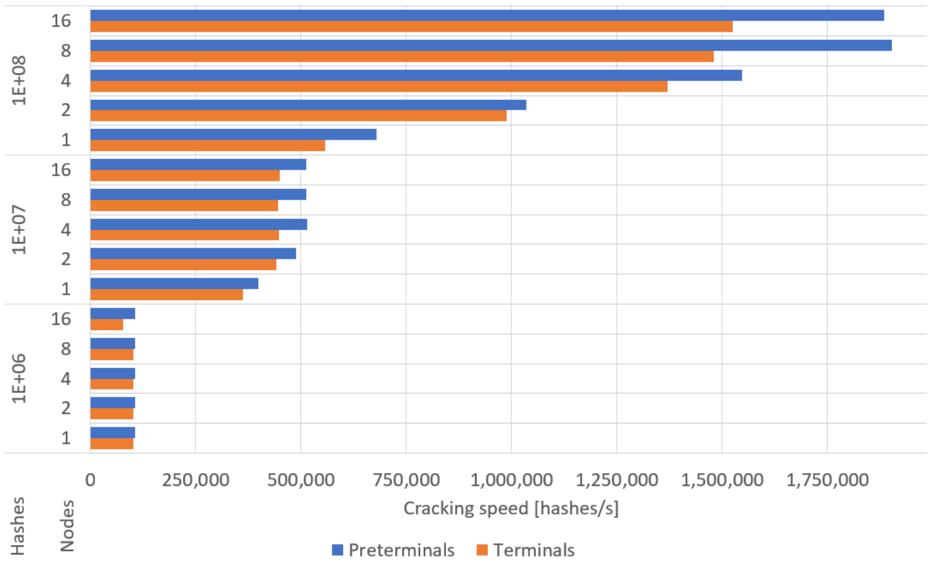

Fig. 9: Scaling across multiple cracking nodes (SHA3-512 / *myspace* grammar / 1000 Mbps network bandwidth)

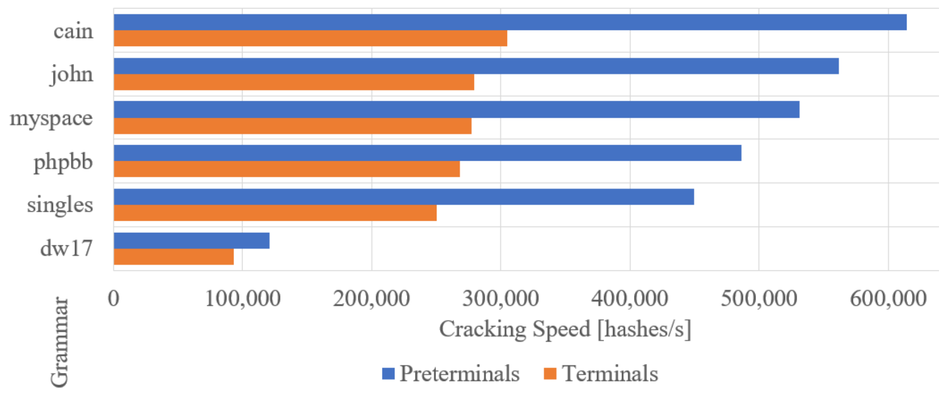

Fig. 10: Differences in cracking speed among grammars

4.3 Workunit Scrambling

The Deadbeat dad algorithm [18] ran on the server ensures the preterminal structures are generated in a probability order. The same holds for passwords, if generated sequentially. However, in a distributed environment, the order of outgoing chunks and incoming results may scramble due to the non-deterministic behavior of the network. Such a property could be removed by adding an extra intra-node synchronization to the proposed protocol. However, we do not consider that necessary if the goal is to verify all generated passwords in the shortest possible time.

Moreover, the scrambling does not affect the result as a whole. The goal of generating and verifying n most probable passwords is fulfilled. For example, for an assignment of generating and verifying 1 million most probable passwords from a PCFG, our solution generates and verifies 1 million most probable passwords, despite the incoming results might be received in a different order.

Though the scrambling has no impact on the final results, we study the extent of chunk scrambling in our setup. We observe the average difference between the expected and real order of chunks arriving at the server, calling it a *scramble factor* S_f. In the following equation, n is the number of chunks, and r_k is the index where k-th chunk was received:

$$S_f = \frac{1}{n}\sum_{k=1}^{n} |(k - r_k)|. \tag{3}$$

We identified two key features affecting the scramble factor – the number of the computing nodes in the system and the number of chunks distributed. In Figure 11, one can see that the scramble factor is increasing with the increasing number of chunks and computing nodes. This particular graph represents experiments with bcrypt algorithm, *myspace* grammar, capped at 10 million hashes. Other experiments resulted in a similar pattern. We conclude that the scrambling is relatively low in comparison with the number of chunks and nodes.

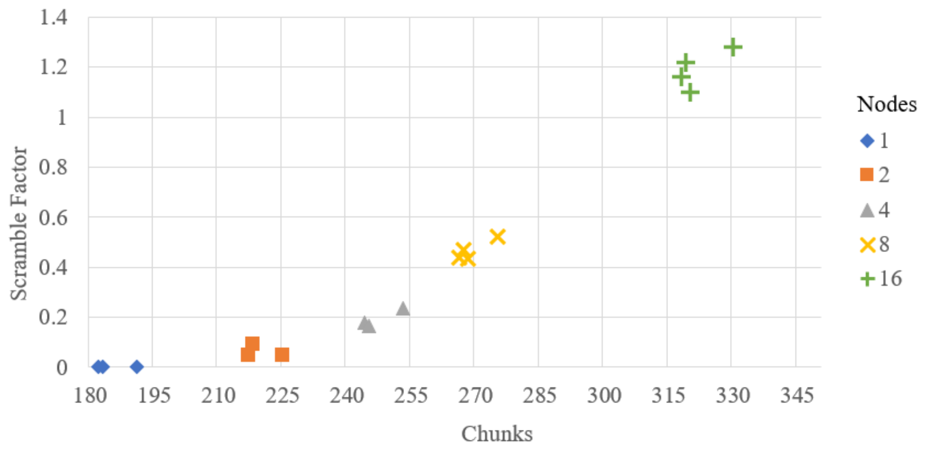

Fig. 11: Average scramble factor, bcrypt *myspace* grammar capped at 10 milion hashes

5 Conclusion

We proposed a method and a protocol for distributed password cracking using probabilistic context-free grammars. The design was experimentally verified using our proof-of-concept tool with a native hashcat support. We showed that distributing the preterminal structures instead of the final passwords reduces the bandwith requirements dramatically, and in many cases, brings a significant speedup. This fact confirms the hypothesis of Weir et al., who suggested preterminal distribution may be helpful in a distributed password cracking trial [19]. Moreover, we showed how different parameters, such as the hash algorithm, network bandwidth, or the choice of concrete grammar, affect the cracking process.

In the future, we would like to redesign the tool to become an envelope over Weir's compiled version of the password cracker in order to stay up to date with the new features that are added over time. The C-based version could figure as a module for our solution.

Acknowledgements The research presented in this paper is 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.

References

1. Aggarwal, S., Houshmand, S., Weir, M.: New technologies in password cracking techniques. In: Cyber Security: Power and Technology, pp. 179–198. Springer (2018)

2. Bonneau, J.: The science of guessing: Analyzing an anonymized corpus of 70 million passwords. In: 2012 IEEE Symposium on Security and Privacy. pp. 538–552 (May 2012). https://doi.org/10.1109/SP.2012.49
3. Das, A., Bonneau, J., Caesar, M., Borisov, N., Wang, X.: The tangled web of password reuse. In: NDSS. vol. 14, pp. 23–26 (2014)
4. Florencio, D., Herley, C.: A large-scale study of web password habits. In: Proceedings of the 16th International Conference on World Wide Web. pp. 657–666. WWW '07, ACM, New York, NY, USA (2007). https://doi.org/10.1145/1242572.1242661
5. Ginsburg, S.: The Mathematical Theory of Context Free Languages. McGraw-Hill Book Company (1966)
6. Hellman, M.: A cryptanalytic time-memory trade-off. IEEE transactions on Information Theory **26**(4), 401–406 (1980)
7. Houshmand, S., Aggarwal, S.: Using personal information in targeted grammar-based probabilistic password attacks. In: IFIP International Conference on Digital Forensics. pp. 285–303. Springer (2017)
8. Houshmand, S., Aggarwal, S., Flood, R.: Next gen pcfg password cracking. IEEE Trans. Information Forensics and Security **10**(8), 1776–1791 (2015)
9. Hranický, R., Holkovič, M., Matoušek, P., Ryšavý, O.: On efficiency of distributed password recovery. The Journal of Digital Forensics, Security and Law **11**(2), 79–96 (2016), `http://www.fit.vutbr.cz/research/view_pub.php.cs?id=11276`
10. Hranický, R., Lištiak, F., Mikuš, D., Ryšavý, O.: On practical aspects of pcfg password cracking. In: Foley, S.N. (ed.) Data and Applications Security and Privacy XXXIII. pp. 43–60. Springer International Publishing, Cham (2019)
11. Hranický, R., Zobal, L., Ryšavý, O., Kolář, D.: Distributed password cracking with boinc and hashcat. Digital Investigation **2019**(30), 161–172 (2019). https://doi.org/10.1016/j.diin.2019.08.001, `https://www.fit.vut.cz/research/publication/11961`
12. Ma, J., Yang, W., Luo, M., Li, N.: A study of probabilistic password models. In: 2014 IEEE Symposium on Security and Privacy. pp. 689–704 (May 2014). https://doi.org/10.1109/SP.2014.50
13. Narayanan, A., Shmatikov, V.: Fast dictionary attacks on passwords using time-space tradeoff. In: Proceedings of the 12th ACM Conference on Computer and Communications Security. pp. 364–372. CCS '05, ACM, New York, NY, USA (2005). https://doi.org/10.1145/1102120.1102168
14. Oechslin, P.: Making a faster cryptanalytic time-memory trade-off. In: Boneh, D. (ed.) Advances in Cryptology - CRYPTO 2003. pp. 617–630. Springer Berlin Heidelberg, Berlin, Heidelberg (2003)
15. Rabiner, L.R.: A tutorial on hidden markov models and selected applications in speech recognition. Proceedings of the IEEE **77**(2), 257–286 (Feb 1989). https://doi.org/10.1109/5.18626
16. Robling Denning, D.E.: Cryptography and data security. Addison-Wesley Longman Publishing Co., Inc. (1982)
17. Veras, R., Collins, C., Thorpe, J.: On semantic patterns of passwords and their security impact. In: NDSS (2014)
18. Weir, C.M.: Using probabilistic techniques to aid in password cracking attacks. Ph.D. thesis, Florida State University (2010)
19. Weir, M., Aggarwal, S., d. Medeiros, B., Glodek, B.: Password cracking using probabilistic context-free grammars. In: 2009 30th IEEE Symposium on Security and Privacy. pp. 391–405 (May 2009). https://doi.org/10.1109/SP.2009.8

20. Weir, M., Aggarwal, S., Collins, M., Stern, H.: Testing metrics for password creation policies by attacking large sets of revealed passwords. In: Proceedings of the 17th ACM conference on Computer and communications security. pp. 162–175 (2010)