On Practical Aspects of PCFG Password Cracking

Radek Hranický, Filip Lištiak, Dávid Mikuš, Ondřej Ryšavý

On Practical Aspects of PCFG Password Cracking

Radek Hranický, Filip Lištiak, Dávid Mikuš, and Ondřej Ryšavý

Faculty of Information Technology, Brno University of Technology, Czech Republic
`ihranicky@fit.vutbr.cz`, {`xlisti00`,`xmikus15`}`@stud.fit.vutbr.cz`, `rysavy@fit.vutbr.cz`

Abstract. When users choose passwords to secure their computers, data, or Internet service accounts, they tend to create passwords that are easy to remember. Probabilistic methods for password cracking profit from this fact, and allow the attackers and forensic investigators to guess user passwords more precisely. In this paper, we present our additions to a technique based on probabilistic context-free grammars. By modification of existing principles, we show how to guess more passwords for the same time, and how to reduce the total number of guesses without significant impact on success rate.

Keywords: Password · Cracking · Security · Grammar

1 Introduction

Confidential data and user accounts for various systems and services are protected by passwords. Though a password is usually the only piece that separates a potential attacker from accessing the privileged data, users tend to choose weak passwords which are easy to remember [1]. In reaction, system administrators and software developers introduce mandatory rules for password composition, e.g., "use at least one special character." While password-creation policies force users to create stronger passwords [11,13], recent leaks of credentials from various websites showed the reality is much more bitter. People widely craft passwords from existing words [4] and often reuse the same password between multiple sites [3]. This fact may be utilized by both malicious attackers and forensic investigators who seek for evidence in password-protected data.

Traditional ways of password cracking contain a *brute-force* attack where one tries every possible sequence of characters upon a given alphabet, and a *dictionary attack* where one uses a list of existing passwords and tries each of them. The main drawback of the brute-force attack is the size of a *keyspace* (a set of all possible password candidates) which grows exponentially with the length of the password, and one does not need to "try everything" to crack the password. The dictionary attack, on the other hand, usually checks a limited number of commonly-used or previously-leaked passwords. It is possible, however, to combine both methods to perform a "smarter" cracking. The use of probability and statistics has been shown to bring substantially better results for cracking human-created passwords [9, 10, 15].

One approach is the use of *Markov chains* which consider probabilities that a certain character will follow after another one. The probabilities are learned from an existing password dictionary and then reused for generating password guesses [10]. The method, however, only works with individual characters and does not consider digraphs or trigraphs. To work with larger password fragments, Weir et al. proposed the use of *probabilistic context-free grammars* (PCFG) that can describe the structure of passwords in an existing (training) dictionary. Fragments described by PCFG represent finite sequences of letters, digits, and special characters. Then, by derivation using rewriting rules of the grammar, one can not only generate all passwords from the original dictionary, but produce many new ones that still respect password-creation patterns learned from the dictionary [15].

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. Using PCFGs, generating password guesses is deterministic and is performed in an order defined by their probabilities. Therefore, more probable passwords are generated first.

While the creation of such grammar is fast and straightforward, the application of rewriting rules takes a significant amount of processor time, and the number of generated passwords is overwhelming in comparison with the original dictionary. For example, using Weir's tool[1] with a PCFG trained on 6.5 kB *elite-hacker*[2] dataset (895 passwords) generates a 12 MB dictionary with 1.8 million passwords. However, using 73 kB *faithwriters* dataset (8,347 passwords) generates a 28 GB dictionary with over 3 billion passwords. Even more unpleasant is the time required to generate such datasets. The first 10 and first 100 passwords of *darkweb2017*[3] dataset and *darkweb2017-top100* can be both used for training and generating within 1 minute on Core(TM) i7-7700K CPU. Taking first 1000 passwords requires more than a day to generate guesses on the same processor.

1.1 Contribution

Our goal was to make the PCFG-based password cracking utilizable for practical use. We identified factors that influence the time of generating password guesses. Based on Weir's Python PCFG cracker, we created an implementation in Go[4] language, which enables to parallelize the generation of terminal structures making the password generation multiple times faster. Moreover, we proposed methods that remove specific rewriting rules from the grammar which leads to a massive speedup of password guessing and allows the process to end in a meaningful time without having a considerable impact on success rate.

[1] `https://github.com/lakiw/pcfg_cracker`
[2] `https://wiki.skullsecurity.org/index.php?title=Passwords`
[3] `https://github.com/danielmiessler/SecLists/tree/master/Passwords`
[4] `https://golang.org/`

1.2 Structure of the paper

The paper is structured as follows. Section 2 provides an introduction to PCFG and discusses related work. Section 3 describes the enhancements we made to PCFG-based techniques, while Section 4 shows experimental results of our work. Finally, Section 5 concludes the paper.

2 Background and related work

For a long time, probability and statistics have been applied to measure password strength [8, 11, 13] and generate guesses in password cracking [7, 9, 10, 15]. Major password leaks allowed to make a clearer image of how user create their passwords [2]. Such knowledge has been utilized in multiple password cracking principles and adopted to existing tools.

Narayanan et al. proposed the use of Markov chains for password guessing. The method uses conditional probability $P(A|B)$ that character A will follow after character B. The probabilities for all characters A, B are stored in a matrix obtained by the analysis of an existing password dictionary [10]. The technique was utilized in *Hashcat* tool which uses Markov chains for brute-force attacks by default. The probability matrix can be generated automatically using *Hcstatgen*[5] utility and is stored in a *.hcstat* file. Recent versions of Hashcat use LZMA compression which is indicated by *.hcstat2* file extension.

Weir et al. introduced password cracking using *probabilistic context-free grammars* (PCFG) [15]. The mathematical model is based on classic context-free grammars [5] with the only difference that each rewriting rule is assigned a probability value. The grammar is created by training on an existing password dictionary. Each password is divided into continuous fragments of letters (L), digits (D), and special characters (S). For fragment of length n, a rewriting rule of the following form is created: $T_n \rightarrow f : p$, where T is a type of the character group (L, D, S), f is the fragment itself, and p is the probability obtained by dividing the number of occurrences of the fragment by the number of all fragments of the same type and length. In addition, we add rules that rewrite the starting symbol (S) to *base structures* which are non-terminal sentential forms describing the structure of the password [15]. For example, password "p@per73" is described by base structure $L_1S_1L_3D_2$ since it consist from a single letter followed by a single special character, three letters, and two digits. Table 1 shows rewriting rules of a PCFG generated by training on two passwords: "pass!word" and "love@love". There is only one rule that rewrites S since both passwords are described by the same base structure. By using PCFG on MySpace dataset (split to training and testing part), Weir et al. were able to crack 28% to 128% more passwords in comparison with the default ruleset from *John the Ripper* (JtR) tool[6] using the same number of guesses.

[5] `https://hashcat.net/wiki/doku.php?id=hashcat_utils#hcstatgen`
[6] `https://www.openwall.com/john/`

left	$\rightarrow$	right	probability
S	$\rightarrow$	$L_4S_1L_4$	1
L_4	$\rightarrow$	pass	0.25
L_4	$\rightarrow$	word	0.25
L_4	$\rightarrow$	love	0.5
S_1	$\rightarrow$	@	0.5
S_1	$\rightarrow$	!	0.5

Table 1: An example of PCFG rewriting rules

The proposed approach, however, does not distinguish between lowercase and uppercase letters. Weir extended the original generator by adding capitalization rules like "UULL" or "ULLL" where "U" means uppercase and "L" lowercase. The rules are applied to all letter fragments which increases the number of generated guesses [14]. After adding capitalization, the notation for letter non-terminals were changed from L_n to A_n (as alphabetical) since L now stands for lowercase.

While the previous techniques consider only the syntax of passwords, Veras et al. designed a semantics-based approach which divides password fragments into categories by semantic topics like names, numbers, love, sports, etc. With JtR in *stdin* mode feeded by a semantic-based password generator, Veras achieved better success rates than using Weir's approach or the default JtR wordlist [12].

Ma et al. showed how normalization and smoothing can increase the success rate of Markov models. By training and testing on a huge number of datasets, Ma showed that the improved Markov-based guessing could bring better results than PCFGs [9].

Weir's PCFG-based technique encountered extensions as well. Houshmand et al. introduced keyboard patterns represented by additional rewriting rules that helped improve the success rate by up to 22%, proposed the use of Laplace probability smoothing, and created guidelines for choosing appropriate attack dictionaries [7]. After that, Houshmand also introduced targeted grammars that utilize information about a user who created the password [6].

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

At the time of writing this paper, both tools include the support for letter capitalization rules [14], keyboard patterns [7], as well as the ability to generate new password segments using Markov chains [10]. In the training phase, a user can set a *coverage* value which defines the portion of guesses to be generated using rewriting rules only while the rest is generated using Markov-based brute-force. A *smoothing* parameter allows the user to apply probability smoothing as described in [7]. Moreover, the tools contain the support for context-sensitive character sequences like "<3" or "#1" that, if present in the training data, form a separate set of rewriting rules. Such replacements can be used to describe special strings like smileys, arrows, and others.

Despite numerous improvements made by Houshmand [7], users still have to face slow password guessing speed which is currently the bottleneck of the entire process. Besides, the generating of password guesses gets progressively slower as the time goes on and, as we detected, has high memory requirements. Creating a complete wordlist of possible password candidates using PCFGs trained on leaked datasets may take many hours or even days. Moreover, current tools do not provide information about the size of the *keyspace*, i.e., the number of possible password candidates, and thus the user has no clue about how long will the process take. This obstacle has already been reported as a GitHub issue[7]. Weir, however, does not plan to resolve the issue "anytime soon."

3 Enhancements to PCFG

We focus on making PCFG-based password cracking suitable for practical use - i.e., allow the user to create a PCFG, generate a wordlist of password guesses in a short time, and start cracking immediately. To achieve this, we decided to:

– Create a faster "password generator" that could produce more guesses at the same time using the same hardware.
– Make a tool to calculate the number of possible password guesses from a PCFG. The number can help estimate the size of an output dictionary as well as the time required to generate all password candidates.
– Analyze if modification of an existing grammar can provide any help to the password guessing process. Concretely, if it accelerates the password guessing, or makes it end in a meaningful time.

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

3.1 Key observations

By analyzing the behavior of Weir's Python PCFG Cracker on various leaked datasets, we observed the following:

– The Python implementation of PCFG Manager uses a priority queue and three processes: one that fills the queue with pre-terminal structures [15], one that creates terminal structures (password guesses), and one for storage backup. No other parallelization is supported. Thus, the processor cores are not utilized well.

[7] `https://github.com/lakiw/pcfg_cracker/issues/9`

- Processing long base structures like $A_1D_1A_2D_2A_3D_3A_4D_4A_5D_5$ is computationally complex and wastes a lot of time even if their probabilities are insignificant.
- Rewriting rules for alpha characters (A), digits (D), and other symbols (O) have all similar probability, while rewriting rules for base structures differ more between each other.
- For capitalization of letter fragments, a grammar usually contains few (1 to 4) rules with higher probabilities while the rest have probability below 0.1 and only little impact on success rate.

3.2 Long base structures

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

In PCFG Manager, every base structure is processed by *Deadbeat dad* algorithm [14]. The goal of this algorithm is to create new children from the current node and ensure that these child nodes are inserted into the priority queue in the correct order. Deadbeat dad replaced the original *Next* function [15] and significantly reduced the size of the priority queue at the expense of computing operations [14].

We analyzed the algorithm and observed that the most expensive task is to find every possible parent of every node which is being inserted into the priority queue. In Weir's PCFG Manager, the task is resolved by a function called `dd_is_my_parent` that runs in iterations whose count is potentially increased by every non-terminal present in the processed base structure. The deciding factor is the number of different probabilities assigned to the rewriting rules applicable to the non-terminal. If all usable rules have the same probability value, the number of iterations is not increased. The more different probabilities are present, the more rapidly the iteration count grows, if the non-terminal is added to the base structure.

Table 2 shows the number of `dd_is_my_parent` iterations under different settings. For D_3 non-terminal, all rules have the same probability, and thus D_3 has no impact on the iteration count. For A_1, rewriting rules have 26 to 29 different probability values (A_1^p). As a capitalization rule for A_1, only "L" is used. One can see, the number of iterations grows almost exponentially each time A_1 is added to the base structure.

In PCFGs trained on leaked password datasets, the variedness between rule probabilities is usually high, especially for shorter character fragments. For long base structures, the `dd_is_my_parent` function may iterate millions of times which significantly slows the password guessing process. Such structures usually have low probability values since they are in most cases created from randomly generated strings, not created by users. We assume, removing such structures

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

Table 2: The number of iterations of `dd_is_my_parent` function

from the grammar speeds up password generation several times and does not noticeably decrease success rate at cracking sessions.

3.3 Calculating the number of password candidates

The calculation of possible password guesses from a PCFG is a currently missing[7] feature that is, however, essential for tools presented in this paper. Let $size(N)$ be the number of terminal structures that can be created by applying rewriting rules on non-terminal N. For base structure $B = N_1N_2 \dots N_n$, the number of possible password candidates can be calculated as:

$$cnt\_base(B) = \prod_{i=1}^{n} size(N_n). \qquad (1)$$

For grammar G, the total number of possible password candidates is the sum of $cnt\_base(B)$ for all base structures $B \in G$:

$$cnt\_total(G) = \sum_{B \in G} cnt\_base(B). \qquad (2)$$

The file and directory structure of Weir's PCFG considers a single rewriting rule per line. All rewriting rules have non-zero probability, and thus, all are used. Therefore, $size(N)$ for non-terminal $N = T_n$ (see Section 2) is, in most cases, the number of lines in `n.txt` file located in a directory for fragments of type T. For example, $size(D_3)$ equals the number of lines in `Digits/3.txt` file. Since letter capitalization rules have been introduced, it is necessary to take them into consideration. Thus, $size(A_n)$ is the number of lines in `Alpha/n.txt` file multiplied by the number of lines in `Capitalization/n.txt` file.

The calculation shown above is usable for classical PCFG-based approach only, i.e., with the `--coverage` parameter of PCFG Trainer set to 1. Otherwise, Weir's PCFG Manager would create additional character fragments using brute-force and Markov chains which is out of the scope of this paper.

3.4 The new PCFG Manager

To improve the use of resources, we created an alternative[8] to Weir's PCFG Manager. We started with a simple transcription of Python sources to Go programming language that we chose because of its speed, simplicity, and compilation to machine language. Early experiments showed that our Go-based alternative using the same algorithms was about four times faster than the original solution. However, there was still enough space for optimization.

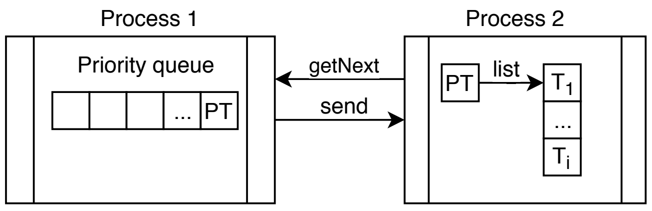
Within all steps performed by the PCFG Manager, generating password guesses from pre-terminal structures [14,15] was the most computationally complex part. Since there is no mutual dependence between the pre-terminals, we decided to modify the program and parallelize this part of the process. Our new design uses a single *goroutine* (a lightweight thread) for filling the priority queue [15] with pre-terminal structures, and one to n goroutines for generating terminals in parallel. The n can be set by a user to reflect the processor's capabilities. Moreover, we added a parameter which allows the user to limit the number of generated password guesses. We illustrate both approaches by simplified schematics that display goroutines and data transfer operations. While Figure 1a shows the original design of Weir's PCFG Manager, the parallel version is depicted in Figure 1b.

For synchronization and mutual communication, goroutines use a mechanism called *channels* that act as FIFO queues. A goroutine can send values to a channel or receive values from it. By default, channels are not buffered and both send and receive operations are blocking. In our solution, we use a buffered channel of size n where the sender is blocked only if the channel contains n values in the queue. Each value represents a pre-terminal structure. The main goroutine (M) implements the Deadbeat dad algorithm [14] filling the priority queue with pre-terminals. Every time a pre-terminal is created, it is sent to the buffered channel if there is enough space. Every time the channel is full, the main goroutine is suspended automatically by the send operation. There is no need to generate more pre-terminals at the time they cannot be processed. In contrast to the original version, the proposed design allows to process multiple pre-terminals and generate passwords in parallel if $n > 1$. In that case, the only apparent drawback is the possible slight change of the password order at the output. This behavior could be resolved by adding a supplementary synchronization mechanism at the output, however, at the cost of performance loss. For practical use, we do not consider this as a large obstacle since for millions of password, the changes are insignificant because the order of larger password blocks is preserved. Moreover, if the user does not set the guess limit explicitly, or if the limit is set in the PCFG Mower (see Section 3.5) instead of PCFG Manager, the output dictionary contains the same passwords, and the success rate would be intact.

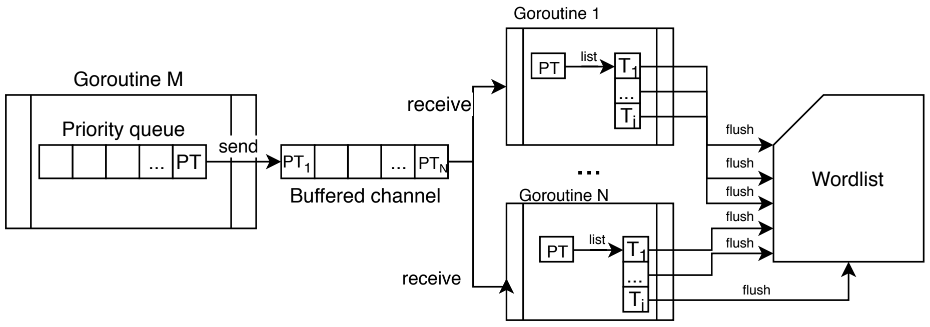
By profiling, we later detected that even though we accelerated generating terminal structures, the new bottleneck was at the output, where simple I/O text operations slowed down the entire process. We overcame this obstacle by

[8] `https://github.com/Dasio/pcfg-manager`

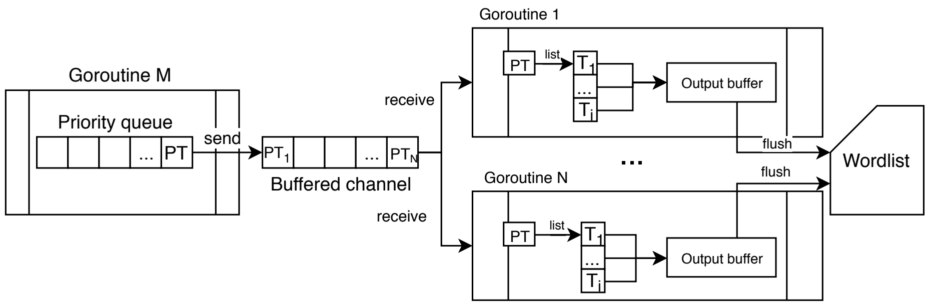
adding extra output buffers to goroutines that generate terminal structures. The buffers store the terminal structures and are flushed to output after the entire pre-terminal is processed. The final design is illustrated in Figure 1c and the experimental results in Section 4.



(a) Python PCFG Manager



(b) Go PCFG Manager



(c) Go PCFG Manager with buffered output

Fig. 1: The architecture of PCFG Manager in Python and Go (PT - pre-terminal structure, T - terminal)

3.5 Grammar filtering

To increase speed even more, we experimented with various modifications of already-trained grammars. We noticed that removing rules which rewrite the starting symbol into long base structures brings a significant speedup without higher impact on a success rate. The motivation for such filtering was discussed in Section 3.2. We automated the process by creating a simple script that automatically filters out all base structures longer than a user-defined maximum.

At this point, we were able to generate much more passwords per time unit. However, without a manually-defined limit for password guesses, the total amount of time required for generating was still extensive. From a practical perspective, any limit to guess count means that there is always a part of the grammar that is never used and unnecessarily wastes memory during the guess generation. Such consideration led us to speculate about reducing the size of the grammar instead of limiting guesses in PCFG Manager.

We came with an idea to remove the least significant rewriting rules from the grammar. We are aware of the fact that any removal of rules from already-created PCFG without adjusting probability values results in a mathematically incorrect grammar where the total probability of rules that rewrite some non-terminals may be lower than one. For practical use with the PCFG Manager, it does not matter. The goal of the filtering is to make the output dictionary more compact and to ensure that generating passwords will end in a meaningful time. Besides, having a reduced grammar that can be processed entirely, ensures that even the parallel run of PCFG Manager generates the same passwords every time. Nevertheless, the strongest motivation for grammar filtering is a potentially massive saving of processor time. Putting a limit before the guessing even starts prevents the Deadbeat dad algorithm from performing many useless derivation steps on trees that never form terminal passwords due to a low probability.

As denoted above, rules for alpha characters, digits, and special symbols usually have similar probabilities, thus removing them leads to a considerable loss of information which decreases the success rate. Rulesets for base structures and capitalization, on the other hand, contain many insignificant rewriting rules that can be removed safely. We created a tool called *PCFG Mower*[9] which can:

- Calculate the total number of possible password guesses from a PCFG and inform the user about achievable keyspace. Moreover, if the user knows an average speed of password guessing, it is possible to estimate the total time required for generating all password candidates.
- Filter a PCFG by performing an automatic removal of rewriting rules based on a set of options entered by the user.

To verify our assumptions, we created a simple *PCFG reduction algorithm* that is implemented in PCFG Mower and shown in Figure 2. The goal of the algorithm is not to provide a universal solution, but to validate or disprove that systematic PCFG filtering brings a possible benefit to password cracking. Besides the

[9] `https://github.com/findo11/pcfg_mower`

```
Input: original grammar, $limit$, $b_s$, $c_s$
Output: reduced grammar
 1: $reduce$ = true, $i = 0$
 2: repeat
 3:     $i$++
 4:     $count$ = password_count()
 5:     if $count \leq limit$ then
 6:         $reduce$ = false
 7:     end if
 8:     if $reduce$ then
 9:         Remove as many base structures as required to reduce their total probability
            by $b_s$.
10:         Remove all capitalization rules that have probability lower than $i \times c_s$.
11:     end if
12: until not $reduce$
```

Fig. 2: PCFG reduction algorithm

original grammar, it takes the following input parameters: *limit* defining the maximum number of password guesses to be generated, and probability values b_s, c_s. While b_s allows to set how rapidly should the algorithm remove base structures, c_s sets the same for capitalization rules. The output of the algorithm represents a PCFG which generates the maximum of *limit* password guesses.

4 Experimental results

In this section, we demonstrate the practical benefits of our enhancements to PCFGs. For experimental purposes, we work with both original and modified datasets from real password leaks. As data sources, we used SkullSecurity[2] pages and SecLists[3] repository. All employed datasets are enlisted in table 3. For shorter notation, we assign each a unique identifier (ID). The last row (def) represents the default PCFG from Weir's PCFG Cracker[1], which is said to be trained on a random sample of million passwords from RockYou dataset.

The table shows the number of passwords in the dataset (pw count), its size, and the average password length (avg). The other columns illustrate how a PCFG trained on the dataset looks like. We show the number of rewriting rules for alpha characters (A), digits (D), other characters (O) as well as the number of rewriting rules for base structures (base) and capitalization (cap).

4.1 The performance of PCFG Manager

At first, we measured the acceleration that can be achieved using our new PCFG Manager in contrast with the original one from Weir et al. [15]. Table 4 shows experimental results of generating password guesses using PCFG trained on *Darkweb2017-10000* dataset (dw), *RockYou-75* dataset (ru75), and the default PCFG (def) used in Weir's cracker. All three experiments were performed using

dataset					PCFG				
ID	name	pw count	size	avg	A	D	O	base	cap
dw	Darkweb2017-10000.txt	10,000	82.6 kB	7	5,244	947	30	323	83
r65	rockyou-65.txt	30,290	344.5 kB	7	17,845	4,213	35	256	39
r75	rockyou-75.txt	59,187	478.9 kB	7	30,670	10,601	51	351	51
ms	myspace.txt	37,126	354.2 kB	8	22,587	4,273	133	1,574	179
tl	tuscl.txt	38,820	324.7 kB	7	26,806	6,518	71	1,290	242
pr	probab-v2-top12000.txt	12,645	100.2 kB	6	11,117	534	1	125	23
def	Random million passwords from RockYou				330,343	145,510	906	84,307	950

Table 3: Password datasets used for experiments

a computer with Intel(R) Core(TM) i7-4700HQ CPU with 8 GB RAM. We also decided to study the influence of disk I/O speed, so that we measured everything using HDD and then using SSD. In all cases, we measured how many password guesses we can generate within 3 minutes.

Our solution was from 8 to 40 times faster than the original one. Using *Darkweb* dataset (see Figure 3) resulted in lowest acceleration since it contains long and complex base structures. With the default PCFG (see Figure 4) and *Rockyou-75* dataset (see Figure 5), we were able to generate much more password guesses, and the difference between HDD and SSD is more noticeable.

training	manager	HDD	SSD
dw	Python	3,022,923	2,948,532
	Go	24,592,908	24,609,579
	acceleration	8.14 x	8.35 x
def	Python	29,613,726	32,402,490
	Go	405,819,926	485,244,534
	acceleration	13.70 x	14.98 x
r75	Python	18,418,684	20,843,491
	Go	490,635,443	842,695,475
	acceleration	26.64 x	40.43 x

Table 4: No. of guesses and acceleration of PCFG manager

4.2 The impact of PCFG filtering

The second set of experiments aim to examine the effects of our attempts to reduce the grammar. Table 5 shows the results of training, modification, generating password guesses, and checking success rate using multiple datasets. The experiments were performed using Intel(R) Core(TM) i7-7700K CPU with 32 GB RAM and an SSD. Since generating password guesses using non-modified

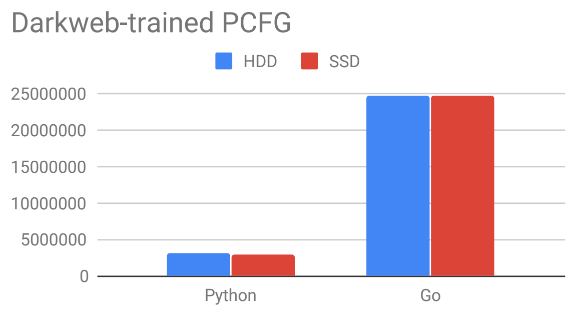


Fig. 3: No. of guesses within 3 minutes using Darkweb-trained PCFG

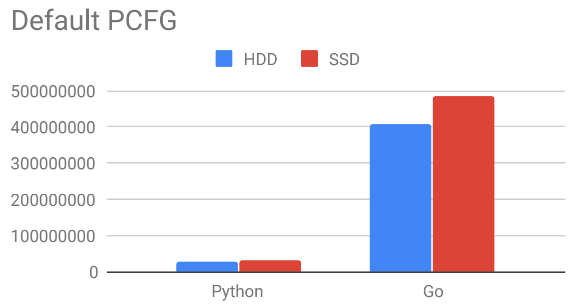


Fig. 4: No. of guesses within 3 minutes using Default PCFG

PCFGs would take hours and days, we set a time limit of 10 minutes to all measurements - every time the PCFG manager exceeded the 10-minute interval, it was stopped.

The first column (tr) shows which dataset we used for training to create the PCFG. For all training datasets, the first line represents generating password guesses using the original grammar - i.e., without any modification. The *longbase* modification stands for the grammar where we removed base structures longer than 10 characters (5 non-terminals). In other measurements, we used a grammar with already-removed long base structures and reduced it using *PCFG Mower* described in Section 3. The *mow-n* modification means that we performed *longbase* first and then we set the *limit* of the PCFG reduction algorithm to n passwords. We experimented with the following *limit* values: 1,000,000,000 (1000M), 500,000,000 (500M), and 20,000,000 (20M) passwords. In all cases, the b_s and c_s constants were set to 0.001 to achieve fine-grained filtering. Since the algorithm removes selected rules, we illustrate the changes done to the grammars in each step. For every modification, we display the preserved number of rewriting rules for base structures (base) and capitalization (cap).

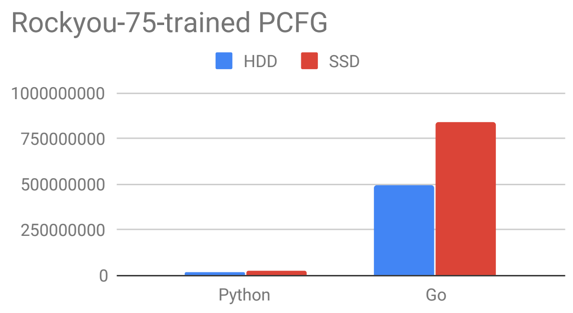


Fig. 5: No. of guesses within 3 minutes using Rockyou-75-trained PCFG

Next columns inform about password guessing. We display the amount of time required to generate the output dictionary (time), (or $10m^*$ if we reached the time 10-minute limit), the size of the output dictionary (out size) and the number of its passwords in millions (mop). The rest displays the success rate of password guessing on testing datasets - i.e., the percentage telling how many generated password guesses were included in different testing datasets. The last column displays the *average success rate impact* (ASRI) which is calculated as:

$$ASRI = \frac{\sum_{i=1}^{n}(SR_i^{mod} - SR_i^{orig})}{n}$$

where SR_i^{orig} is the success rate on testing dataset number i before the modification of the PCFG, and SR_i^{mod} is the success rate on testing dataset number i after the modification of the PCFG, and n is the total number of testing datasets. In our case, $n = 4$. We use ASRI to analyze the influence of our modifications. Positive ASRI means that the success rate was improved while negative stands for decrease.

As we can see from results, removing long base structures resulted in a massive increase of password guessing speed which enabled to generate much more passwords within 10 minutes. We achieved the highest acceleration on *dw* and *r65* since they contain very complex passwords that create enormously long base structures. After the modification, we were able to generate over 14 times more password guesses. In contrast, training on *ms* and *tl* creates more simple grammars, and thus the speedup was not as rapid. Removing long base structures showed almost no impact on the success rate which confirms our assumption that their importance is negligible. From 16 testings, only 8 led to decrease by a maximum of 0.06 %. To our surprise, the ASRI was mostly positive since in 6 cases, removing long base structures improved the success rate by up to 0.7 % thanks to more passwords generated within the same time.

Next measurements analyzed grammars filtered by PCFG Mower to verify if the removal of low-probability rewriting rules brings any benefit. In all cases, the *mow* modification allowed the PCFG Manager to process the entire grammar

	grammar			password guesses			success rate				
tr	modification	base	cap	time	out size	mop	pr	ms	dw	r65	ASRI
	original	323	83	10m*	731 MB	78	45.03 %	26.83 %	98.27 %	41.39 %	
	longbase	288	83	10m*	12 GB	1,110	45.01 %	26.91 %	98.35 %	41.40 %	+0.04 %
dw	mow-1000M	106	40	25s	3.3 GB	373	44.54 %	24.47 %	96.42 %	38.36 %	-1.93 %
	mow-500M	106	40	25s	3.3 GB	373	44.54 %	24.47 %	96.42 %	38.36 %	-1.93 %
	mow-20M	86	32	2s	77 MB	9	44.18 %	24.12 %	95.65 %	38.00 %	-2.39 %
	original	256	39	10m*	1.5 GB	151	72.34 %	37.63 %	88.25 %	99.84 %	
	longbase	223	39	10m*	25 GB	2,210	72.30 %	37.63 %	88.14 %	99.81 %	-0.05 %
r65	mow-1000M	161	36	3m 31s	11 GB	980	72.17 %	37.17 %	87.73 %	99.61 %	-0.35 %
	mow-500M	123	31	1m 31s	4.5 GB	409	72.01 %	36.62 %	87.23 %	99.35 %	-0.71 %
	mow-20M	79	20	3.5s	130 MB	13.8	70.98 %	34.26 %	85.80 %	97.16 %	-2.47 %
	original	1574	179	10m*	5.7 GB	616	47.47 %	93.68 %	69.14 %	46.42 %	
	longbase	1430	179	10m*	9.5 GB	1,030	47.45 %	94.38 %	69.07 %	46.42 %	+0.15 %
ms	mow-1000M	110	25	3m	9.2 GB	941	46.37 %	82.40 %	66.74 %	43.04 %	-4.54 %
	mow-500M	78	20	1m	3.1 GB	334	45.13 %	79.67 %	64.71 %	42.62 %	-6.15 %
	mow-20M	21	20	2s	126 MB	15	33.25 %	61.17 %	54.28 %	35.58 %	-18.11 %
	original	1290	242	10m*	4.5 GB	520	55.27 %	36.87 %	69.85 %	43.86 %	
	longbase	1158	242	10m*	7.6 GB	870	55.23 %	37.15 %	69.79 %	43.87 %	+0.05 %
tl	mow-1000M	91	20	2m 43s	7.5 GB	884	54.06 %	30.94 %	66.08 %	40.37 %	-3.60 %
	mow-500M	48	19	1m 8s	1.8 GB	200	53.77 %	29.05 %	64.19 %	39.39 %	-4.86 %
	mow-20M	24	18	2s	133 MB	17	52.08 %	22.27 %	55.61 %	35.64 %	-10.07 %

Table 5: Success rates of original and modified PCFGs (* - reached the time limit)

in less than 4 minutes, showing that it can provide a suitable alternative to a "hard" limit for password guessing. More compact PCFGs produced smaller dictionaries. With more compact PCFGs, the generated dictionaries were smaller as well. Again, we achieved the best results with *dw* and *r65* datasets, where we were able to reduce the size from 12 GB (longbase) to 112 MB dictionary, and from 25 GB to 130 MB with a loss of success rate below 4 % in all cases. For *ms* and *tl*, filtering the grammar spared time and space as well, however, the *mow-20M* limit was too strict to provide satisfactory results. For *dw*, we received the same results with *mow-1000M* and *mow-500M*. The *dw*-trained grammar contains a high number of base structures with similar probabilities. Thus, a lot of them was removed by *mow-1000M* modification, and no further filtering was necessary.

4.3 Evaluation

By modification of both PCFG Manager and existing grammars, we were able to make password guessing many times faster. What most helped the speedup was the use of a compiled programming language, the parallelization of generating terminal structures and removing rewriting rules for long base structures. For datasets we analyzed, such rules caused "more harm than good." The rewriting rules for long base structures mostly had insignificant probabilities but complicated the calculation of the computationally-complex Deadbeat dad algorithm. In all cases, the removal accelerated the password guessing dramatically.

Filtering grammars with PCFG Mower reduced the time required for password guessing rapidly. The settings, however, have to be selected wisely. With our experimental setup, we achieved the best results with PCFG Mower limit set to 500 millions of passwords. Stricter limitation produced decent results for only some cases. We assume that the success rate highly depends on the nature of selected datasets, and thus there is no universal solution.

5 Conclusion

Probabilistic methods certainly have their place in the area of password cracking. While Markov chains were adopted to existing tools a long time ago, probabilistic context-free grammars are currently more a subject of academic research than a ready-to-use technique. However, as the development of cracking methods continues by researchers, communities, and commercial subjects, the situation may change. Even the authors of Hashcat consider[10] adding support for generating "slow candidates."

From our standpoint, one of the main factors that currently complicate the use of PCFG-based techniques is the extensive amount of time required to generate password guesses. By using both analytic and experimental approach, we identified the critical spots that slowed down the entire process. We proposed methods that optimize the password guessing and allow better use of hardware resources. We experimentally proved that our new PCFG Manager is capable of generating passwords 8 to 40 times faster than the original tool from Weir et al.

Moreover, we proposed a way of PCFG filtering which provides a resource-saving alternative to a "hard" password guess limit. We showed that the systematic removal of selected rewriting rules might reduce the total amount of time required to generate password candidates without having a significant impact on the success rate. If one decides to use the filtering techniques, we recommend starting with the removal of long base structures that produce the least-probable passwords and perceptibly increase the number of necessary processor operations.

In our future research, we want to perform a more detailed analysis of the relation between PCFG filtering and the success ratio which may discover new factors that have not been revealed yet. Since the practical use of password cracking often involves a distributed environment, we currently work on distributed PCFG-based password guessing techniques which may provide a smarter alternative for a classic dictionary attack.

Acknowledgements The 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

[10] `https://hashcat.net/forum/thread-7903.html`

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.

References

1. Bishop, M., Klein, D.V.: Improving system security via proactive password checking. Computers & Security **14**(3), 233–249 (1995)
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. 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)
7. Houshmand, S., Aggarwal, S., Flood, R.: Next gen pcfg password cracking. IEEE Trans. Information Forensics and Security **10**(8), 1776–1791 (2015)
8. Kelley, P.G., Komanduri, S., Mazurek, M.L., Shay, R., Vidas, T., Bauer, L., Christin, N., Cranor, L.F., Lopez, J.: Guess again (and again and again): Measuring password strength by simulating password-cracking algorithms. In: Security and Privacy (SP), 2012 IEEE Symposium on. pp. 523–537. IEEE (2012)
9. 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
10. 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
11. Proctor, R.W., Lien, M.C., Vu, K.P.L., Schultz, E.E., Salvendy, G.: Improving computer security for authentication of users: Influence of proactive password restrictions. Behavior Research Methods, Instruments, & Computers **34**(2), 163–169 (2002)
12. Veras, R., Collins, C., Thorpe, J.: On semantic patterns of passwords and their security impact. In: NDSS (2014)
13. Vu, K.P.L., Proctor, R.W., Bhargav-Spantzel, A., Tai, B.L.B., Cook, J., Schultz, E.E.: Improving password security and memorability to protect personal and organizational information. International Journal of Human-Computer Studies **65**(8), 744 – 757 (2007). https://doi.org/10.1016/j.ijhcs.2007.03.007
14. Weir, C.M.: Using probabilistic techniques to aid in password cracking attacks. Ph.D. thesis, Florida State University (2010)

15. 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