Using Parallel Techniques to Accelerate PCFG-based Password Cracking Attacks

Ming Xu, Shenghao Zhang, Kai Zhang, Haodong Zhang, Junjie Zhang, Jitao Yu, Luwei Cheng, Weili Han* *Member, IEEE*,

Abstract—Textual passwords play an important role among access-control mechanisms and are usually stored as ciphertext in the server. However, an attacker may attempt to hash a large number of candidate passwords to find the match of the target hash of a password database. To crack the password database, attackers in industry usually use the cracking software like Hashcat. Academic researchers recently proposed many data-driven probabilistic models, in which the Probabilistic Context-free Grammars (PCFG, for short) stand out. Despite the great cracking efficiency, the data-driven models are seldom used by industrial practice due to the significant slow generation speed of password candidates.

To bridge the gap and promote the efficient data-driven models being practically used in industry, we propose that using parallel techniques to accelerate the candidate password generation, enabling the integration of PCFG models into the practically-used Hashcat tool. To this end, we mainly propose two algorithms to accelerate the password generation for PCFG-based models: first, we design a storage structure with the memory load balance strategy to more evenly store the data structures used to generate passwords; second, we design an algorithm to produce candidate passwords in parallel by different threads. Based on the two algorithms, we propose `Parallel_PCFG`, and implement `Parallel_PCFG` upon Hashcat based on its built-in GPU kernel. We comprehensively evaluate `Parallel_PCFG` against state-of-the-art data-driven models, and find that `Parallel_PCFG` only takes 14.33% of time to achieve the same cracking rates compared with the best-performing models, paving a way about the integration between PCFG-based models and Hashcat.

Index Terms—Password Security, PCFG-based Models, Candidate Password Generation, GPU Parallelism

1 Introduction

Textual passwords are still widely used to protect accounts and private data in our daily life due to the advantages of easy use and low cost [3], [10], [38], and are typically stored in ciphertext to prevent malicious attacks. Human-created passwords are always not strong enough to resist password cracking attacks. Hackers have recovered many plain-text passwords by attempting to hash a large number of candidate passwords to find the match of the target ciphertext password database, which are referred to as *generic password cracking attacks*.

Recently, researchers proposed many data-driven probabilistic guessing models [9], [34], e.g., Probabilistic Context-free Grammars (PCFG, for short) [38], which they usually learn an educated password space based on the training sets to generate password candidates in descending probabilities [16], [27]. While PCFG becomes a powerful competitor across many data-driven models [15], [38], the generation methods of PCFG-based models significantly suffer from the slow generation speed. In real-world industrial cases, hackers seldom use data-driven models due to the following reasons: (1) The generation speed of data-driven models is slow, e.g., it usually takes a day or more days to generate around 10^{14} candidate passwords [35], [38], increasing the time costs. (2) When targeting ciphertext passwords, data-driven models should leverage the hash functions embeded in the cracking software like `Hashcat` to encrypt the generated plain-text password candidates. Existing industrial practices usually use the *pipeline mode* that serially transmits every generated candidate password to a cracking software, whose efficiency is significantly limited by the slow speed of candidate generation. (3) Data-driven models typically evaluate cracking rates based on an equal number of guesses [21], [38], [39], offering limited insight into how guessing actually performs over time (e.g., within half an hour). Some scenarios may be highly time-sensitive, such as when multiple attackers target the same system simultaneously, with the first to breach the system potentially siphoning off funds or stealing valuable information.

To compensate for the gap in the use of data-driven models in industrial guessing attacks, a rich literature [6], [9], [12] has presented efficient techniques for accelerating the password generation of data-driven probabilistic models. First, they optimize the model structure used to accelerate password generation. Typical example is OMEN [9] that classifies probabilities of a candidate password into ten levels to generate passwords of the same level simultaneously, i.e., generating candidate passwords in a coarsed descending probability order. Second, they leverage the distributed computer clusters, i.e., more computer resources, to generate candidate passwords. Typical examples are PCFG-manager method [6], [12] that distributes the password generation tasks to multiple machines with a thoughtful communication protocol. Still, it has significant potential to accelerate generation speed with the aid of parallel techniques, supporting the industrial use of data-driven models.

As the hardware develops, GPUs can be a promising way for accelerating the password generation tasks because its many-core architecture can offer a higher computational

*corresponding author

throughput with ample parallelism. We, in this paper, are motivated to improve password generation speed using the parallel GPU-based techniques. To this end, the challenges are as follows: 1) The storage structures for data-driven generation methods must be carefully tailored to align with the GPU's parallel generation algorithm. Since GPU runtime is typically constrained by the slowest thread, unbalanced storage structures can severely hinder performance. Optimizing these structures is crucial to minimize bottlenecks and maximize efficiency. 2) Ensuring that the output candidate passwords are in descending order presents another challenge. This objective involves assigning generation tasks to different threads and designing lookup tables to locate each thread's candidates becomes challenging.

To tackle the challenges above, we propose using the GPU-based parallel techniques to accelerate PCFG-based password cracking attacks, referred to as `Parallel_PCFG`. `Parallel_PCFG` includes two key algorithms: 1) A novel storage structure for PCFG with the memory load balance to maximize the memory use. 2) A parallel algorithm assigns password candidate generation to different threads and ensures descending generation by each thread. In detail, we design a lookup table so that threads can simultaneously locate, and generate the password candidates. We implement `Parallel_PCFG` in the popular password cracking software of `Hashcat` [13] due to its built-in GPU kernel.

We evaluate the cracking rates of `Parallel_PCFG` within the same time frame (e.g., half an hour) on six leaked public password datasets, and compare with multiple models including OMEN, PCFG Manager and other PCFG-based counter-parts. We find that `Parallel_PCFG` can achieve 30 ~ 60% cracking rates in just half hour across six datasets. Particularly, `Parallel_PCFG` only takes 14.33% of the total time to achieve the same cracking rates of the state-of-the-art models; `Parallel_PCFG` only needs 10 minutes on average to crack similar percentage of passwords from state-of-the-art models, demonstrating the effectiveness of reducing the time overhead of password cracking. We also compare that `Parallel_PCFG` can significantly outperform the default Hashcat with various rules-set, and even the optimized rules-set [4]. We believe that this work can serve as promising solutions that integrate the PCFG-based models into industrial tools of `Hashcat`, showing its potential to be industrially used.

We summarize the main contributions as follows:

- We propose two algorithms to enable the PCFG-based model to generate password candidates in parallel upon GPU. First, we design a storage with a *memory load balance* strategy for the data used in password generation of PCFG. Second, we design an algorithm to assign the generation tasks to multiple threads.
- We implement `Parallel_PCFG` in `Hashcat` based on the two algorithms, improving the password generation speed in PCFG-based models. We empirically evaluate that `Parallel_PCFG` can significantly reduce the time overhead in password cracking attacks, showing the potential of integrating PCFG into the `Hashcat` used in practical guessing scenarios. Specifically, `Parallel_PCFG` only takes 14.33% time to achieve the similar cracking rates by previous counterparts.
- We provide insights in terms of evaluating the password guessing performance from the timing perspective, rather than simply focusing on the number of guesses. We take the substantial step to unlock that 30 ~ 60% passwords can be guessed within a certain time (e.g, half an hour).

Organization. We first present background knowledge in Section 2. Then we explain related concepts (e.g., existing PCFG generation methods and the challenges of `Parallel_PCFG`) as preliminaries in Section 3. In Section 4, we detail the design of our `Parallel_PCFG`, which includes two components. We present our evaluation in Section 5. Finally, we discuss the economic costs in Section 6 and conclude this paper in Section 7.

2 Background Knowledge

2.1 Password Cracking Attacks

In recent decades, password cracking attacks are a hotspot for practical attack scenarios and research fields. On the one hand, a hacker could leverage the cracking softwares (e.g., Hashcat or JtR) to crack the target passwords by comparing the candidate guesses, which are obtained by applying the set of rules to a dictionary, and further compromise the dataset database or steal the personal data property of users.

On the other hand, researchers proposed several efficient data-driven guessing models [17], [19], [34] that learn an educated password space to produce a larger number of candidate passwords, which are used compared against target passwords. Usually, these data-driven models are used to evaluate the strength of a password via the effort (i.e., the number of guesses) of successfully cracking it, in which they adopts the efficient techniques like the Monte-Carlo [8] algorithms to map the probability to the number of guesses.

2.2 Related Works

Accelerating password cracking attacks. A rich literature has presented efficient techniques to accelerate the password generation methods for data-driven models, which generally involves optimizing the data structure or using the the distributed computer clusters. For one thing, they constantly improve the model structure to accelerate the password generation process. For PCFG-based methods, Weir et al. [14] released a compiled PCFG password guesser, referred to as *Compiled-PCFG* written in C programming language to achieve faster cracking and easier inter-connection with existing cracking softwares. For Markov-based methods, Durmuth et al. proposed the OMEN [9] (Ordered Markov Enumerator) as a faster password guessing method, which divides the probabilities into ten levels and generate candidate passwords of the same level at once. On the other part, researchers proposed to leverage the distributed computer clusters (i.e., more computer resources) to produce candidate passwords simultaneously. For example, Hranický et al. [11] improved the generation process of PCFG, referred to as *PCFG-manager*, based on the distributed clusters computers via the programming language of Go. They introduced a PCFG manager to handle many client's

machine to generate candidate passwords based on multiple computers at the same time.

Combination of data-driven models and cracking software. In 2020, Radek Hranický [12] recommended that combining the data-driven models with the guessing softwares, especially the efficient PCFG to the GPU-based Hashcat guessing software, to launch password cracking attacks. Since the cracking softwares usually depend on the quality of the rules-set [22] (summarized by experts) to achieve a higher cracking performance, which also limits their wider application. For the data-driven models being used in real-world cracking, attackers can transmit the generated candidate passwords into the guessing soft-wares to crack cipher-text passwords. Unfortunately, it usually takes one day or more days to produce a large number of candidate passwords (10^{14}). Then, existing mechanisms usually adopt the *pipeline mode* as a channel that serially transmits every candidate password to Hashcat. The bottleneck lies in the generation speed of every candidate. Such undesirable workflow generally results in the significant time waste, limiting its practical application of real-world guessing. Besides, previously in 2018, on the official forum of Hashcat [18], Weir (one of the initiators of PCFG-based guessing methods, whose nickname on the forum is lakiw), and Steube (one of the authors of Hashcat, whose nickname on the forum is atom) discussed the topic of adding PCFG models to Hashcat. They tried to assign the candidate passwords to each thread by the order in priority queue. However, they discussed that they must initialize the priority queue and generate passwords from the beginning to determine the specific passwords at a particular position in the priority queue, making it challenging to locate the candidate passwords associated with each thread, ending this topic.

2.3 Password Cracking Scenarios

Password cracking scenarios are usually divided into online and offline guessing scenarios, whose main difference is that attackers can try different guesses. Attackers are limited with smaller guesses (e.g., less than 1,000 guesses) to crack target passwords due to the limitation of service providers, therefore, online guessing is often doing the targeted guessing tasks that compromise passwords of a specific use based on their personal information.

Offline cracking attacks generally try a large number of candidate guesses in descending probability to compromise a hashed general password database in the event of an password server breach. These candidate passwords (generated based on the data-drive models or rule-transformed passwords) are then used to be hashed by using the hashing algorithm (e.g., MD5, SHA-1) to compare with the cipher-text password, in which the matching probability is the cracking rate. Existing offline guessing works [24], [38] usually evaluate how many candidate passwords are matched with the target passwords under 10^{14} guesses. However, in an event that the breach is public, there may occur competition that several attackers are trying to compromise the same password database. In extreme cases (e.g., digital wallet), only the first attacker who cracks the password can get the reward, while the rest attackers waste all of their resource payment. Such competition makes the time cost of attack need to be considered by the attacker, which is undervalued by most existing offline guessing works.

2.4 Threat Model

In this paper, we mainly focus on the generic offline guessing scenarios [32], [38] that aim to recover cipher-text and plain-text passwords by the matches of the candidate passwords. We believe that other guessing scenarios like targeted guessing, masked guessing, or other real-world guessings [21], [31], [39] are beyond the scope of this work. We generate the password candidates given limited time (e.g, half hour), and measure the percentage of passwords are matched with the targets, serving as the passwords an attacker can crack within the certain time. Specifically, we assume that attackers can use the Hashcat software and a trained PCFG model, where they can perform password generation tasks in parallel, to show the potential cracking opportunities of PCFG models in real world.

3 Preliminaries

3.1 Probabilistic Context-free Grammars

Probabilistic Context-free Grammars (PCFG) is first proposed by Weir et al. [34] in 2009 (referred to as PCFG-2009). They continuously optimized the method and proposed the improved version of PCFG (referred to as PCFGv4.1 [33]).

PCFG is mainly divided into two phases: **training** and **generation**. In the training phase, PCFG counts the statistical information with their probabilities based on the training datasets. We describe the statistical information as follows: the grammars (a.k.a, structures) and specific strings (associated with structures) with their probabilities based on the training sets. The grammars (structures like L_4D_4) describe the category and length of characters that consist of the password, where the homogeneous L_n, D_n and S_n refer to a sub-structure in a structure, where each sub-structure contains the specific strings composed of letters, digits or special characters with n characters. For example, L_4 is a sub-structure that contains the specific strings like "pass".

$$Grammars \longrightarrow L_4D_4 : prob$$
$$L_4 \longrightarrow pass : prob_1; D_4 \longrightarrow 1234 : prob_2$$

PCFG-based generation phrases. In the process of generating candidate passwords, PCFG uses a priority queue to ensure the priority of candidate passwords with higher probabilities. To reduce the memory overhead of the queue, Aggarmal et al. [1] adopted the structure of pre-terminal and the algorithm of Deadbeat dad. Specifically, it can be useful to clarify several concepts in the candidate password generation in PCFG-based guessing models.

- ***Container***: a container refers to all of the specific strings with the same character categories, lengths and probabilities. Usually, a sub-structure (e.g., L_4) can contain several containers with different probabilities.
- ***Pre-terminal (PT, for short)***: a PT is a basic structure used for password generation, and refers to the collection of containers from all sub-structures in a grammar

(structure). Therefore, the candidate passwords generated from the same PT have the same probabilities.

Figure 1 shows an example of a PT, which contains two containers. The PT comes from the structure of L_4D_4 that consists of two sub-structures L_4 and D_4, whose top two containers (from L_4 and D_4) make up the first pre-terminal (PT-1). For example, the top container associated with the sub-structure L_4 is the set of "pass, word", whose probabilities are both 0.2.

PCFG generates candidate passwords by the structure of PTs, whose resulting candidate passwords from a PT are with the same probability. PCFG uses the Cartesian's product of the specific strings to generate the candidate passwords. As shown in Figure 1, the index sets of two specific strings are both $\{1, 2\}$, resulting in the four pairs of the Cartesian product as follows:

$$\{(1, 1), (1, 2), (2, 1), (2, 2)\}$$

Each pair can describe the combination of the specific strings from every containers that make up a candidate password, For example, (1, 1) describes the candidate password of *pass1234*, which is composed of the *pass* (the index is 1) from L_4 and 1234 (the index is 1) from D_4.

PT-1 (L_4D_4)

Type:	L	Type:	D
Length:	4	Length:	4
Probability:	0.2	Probability:	0.1
Specific strings:	1 pass	Specific strings:	1 1234
	2 word		2 5678

Fig. 1: Example of a pre-terminal, which comes from the structure of L_4D_4.

In the password generation process of PCFG based models, specifically, the priority queue initially contains the most possible (a higher probability) pre-terminal of each structure. Assuming the pre-terminal from Figure 1 (PT-1) is the top of the priority queue. After popping the pre-terminal and generating all the four candidate passwords based on it, we derive PTs with a lower priority. For example, we can replace the first containers (associated with L_4) with other containers of the lower probability of 0.1. Then, we put the newly obtained pre-terminal back into the priority queue. The priority queue will rearrange the order of all the pre-terminals in it. Then PCFG can take the new top pre-terminal out of the priority queue and repeat above steps of generation.

3.2 Challenges of Parallel Password Generation

Here, after we understand the PCFG generation methods, we present the issues when straightforwardly deploying the generation methods in parallel tasks across different threads, which are also the challenges. We first present the GPU parallelism logic and then analyze the issues when deploying the PCFG generation methods in parallel upon GPUs.

3.2.1 GPU parallelism logic.

As the hardware develops, GPUs, which are an electronic circuit that is originally designed to accelerate 3D graphics rendering, gradually become more flexible and programmable to solve more parallel problems. Especially, Hashcat generally performs the guessing tasks in GPUs, which also motivate us to develop GPU-based parallel methods for PCFG-based guessing models based on their superior guessing performance across data-driven models. Generally, we summarize the characteristics of GPUs in parallel tasks as follows.

- GPUs read data from memory with a fixed size of space (i.e., 128 bytes), which is also known as the **cache line** strategies. Therefore, GPU need to read data multiple times when a cache line stores a smaller size of data, resulting in the space and transmission time cost.
- GPUs adopt a **warp** (i.e., multiple threads), which usually involves 32 threads, to perform tasks in parallel in SIMT (Single Instruction Multiple Threads) structure. The warp executes the same instruction with different data resources, whose running time depends on the slowest thread.

3.2.2 Issue analysis

Based on the characteristic above, generating candidate passwords in parallel for PCFG still faces the following issues:

Imbalance issue: PCFG-based guessing methods generate candidate passwords by the unit of pre-terminals, i.e., they start from the top pre-terminal, enumerate all its possible candidate passwords and turn to the next pre-terminal. However, the candidate passwords generated by different pre-terminals are significantly imbalanced, i.e., those pre-terminals with a lower probability can generate more candidate passwords. As shown in Figure 2, we count the number of candidate passwords generated by the top 10,000 pre-terminals in the dataset of `CSDN` and find that the maximum value is 54 while the minimum value is only 1. This is because that the size of containers becomes imbalanced, particularly, those specific strings with the same higher probability are less, while most of specific strings share a lower probability, resulting the pheonmenon. Then when we straightforwardly assign the generation tasks associated with different PTs to different threads, the generation speed can be significantly limited, since the running time in GPUs depends on the slowest thread in a warp [11], promoting to a novel balanced storage structure.

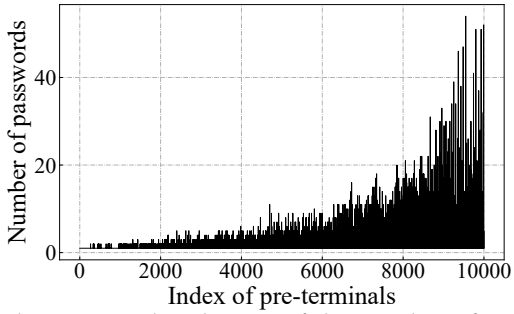


Fig. 2: The uneven distribution of the number of candidate passwords generated by different pre-terminals.

Parallel issue: Existing PCFG candidate password generation methods are on the level of PTs [11] with a priority queue, however, the priority order can be compromised

when we generate PTs in parallel. That said, we cannot directly produce PTs by different threads in parallel. Therefore, we carefully determine that the parallel granularity is on the level of candidate passwords (specific strings stored in a trained PCFG model). In this way, it requires a thoughtful parallel algorithms to assign the candidate password generation tasks to different threads and a careful design principle to locate every specific strings.

4 PARALLEL_PCFG: DESIGN

In this section, we present the design to enable the parallel password generation of PCFG-based methods, which mainly includes two key components of a storage with memory load balancing strategy and a parallel algorithm to assign the generation tasks to different threads.

4.1 Overview

We present the workflow of the parallel PCFG-based models when deployed in Hashcat softwares in Figure 3. The main difference lies in the two designed algorithms (in shaded part) that enables generating passwords in parallel upon the GPU kernal in Hashcat, while existing general models usually adopt *pipeline mode* to transmit every generated candidate passwords in sequence to GPUs. `Parallel_PCFG` first loads the PCFG models and creates the priority queue for pre-terminals on CPU. Then, we use a designed storage structure with a load balancing strategy. Basically, we adopt a dictionary to store and locate all the specific strings in a trained PCFG model and the PTs, `Parallel_PCFG` transmits the PCFG models and PTs to GPU. Next, `Parallel_PCFG` assigns the password generation tasks to different threats to generate candidate passwords in parallel. Finally, `Parallel_PCFG` leverages the hash functions implemented in the GPU kernel of `Hashcat` to encrypt these passwords and match the encrypted passwords with target passwords.

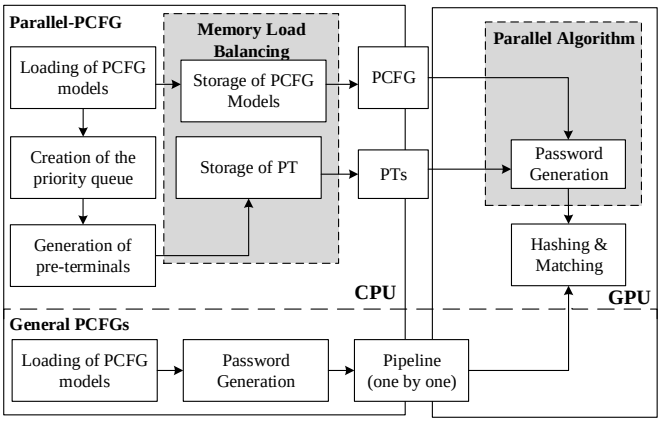


Fig. 3: The workflow of PCFG-based models in `Hashcat`, whose shaded part is our main modified module.

4.2 Memory Load Balancing

Here, we introduce our load balancing mechanism that evenly stores all the specific strings in a trained PCFG model and the PTs generated based on the PCFG model. First, we design a dictionary to store all the specific strings in a PCFG model. Second, we design a storage structure of PTs to improve the efficiency of memory access.

4.2.1 Storage of PCFG models.

For all the specific strings in a trained PCFG model, we design a one-dimensional array and use the three-level principle for storing them by order of *type, length and probability*. That said, we first sort them by type (i.e., letters, digits, or special symbols, and so on); For the specific strings with the same character type, we sort them by length (i.e., from short to long length); For those specific strings with the same type and length, we sort them by descending probability (i.e., from high to low probability).

Take Figure 4 as an example, the character "*a*" comes first in the dictionary (i.e., the index is 1 in the dictionary), since it has the highest probability in the letter strings with one character.

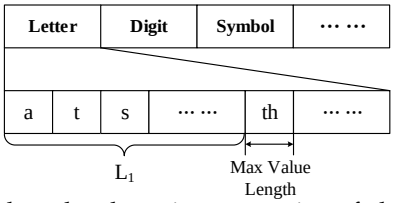


Fig. 4: The three-level sorting strategies of the dictionary storage for all the specific strings in PCFG models.

Then, we can locate the specific strings by its position (i.e., index) in the dictionary by the following Formula:

$$position = starting\_position + rank \times length$$

where *starting_position* refers to the starting position of the specific strings with the same type and length, *rank* refers to the probability ranking (i.e., the number in descending probability across the strings with the same type and length), and *length* refers to the length of this string.

To locate the specific strings in our designed dictionary, we should use another array with the one-dimensional size of $T \times L$ to store the starting position for the specific strings with the same types and lengths, where T refers to the number of types and L indicates the maximum length of the types.

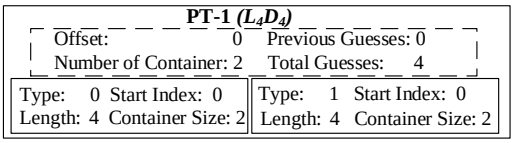


Fig. 5: Our storage structure of the pre-terminal.

4.2.2 Storage of PTs.

Based on the dictionary of all the specific strings of a PCFG model, we design a novel storage structure of PTs to make the passwords generated by different PTs towards even, supporting the parallel generation algorithm. To assign each thread to generate each candidate password simultaneously,

each warp to generate passwords from the same PT, reducing the time wait delay between threads. Besides, existing storage of PTs might store much redundant information, for example, "word" may both exists in a PT associated with the structure of L_4D_4 or a PT with the structure of L_4.

First, we optimize the storage of the PTs to storage the specific strings once. As shown in the Figure 5, we add four values (i.e., the offset, number of container, previous guesses and total guesses). We can divide the data structure of a pre-terminal into the header data (showed by the dotted box) that stores the statistical information of the pre-terminal and the body data (showed by the solid box) that stores the the detailed information of the containers in the PT. In this way, the redundant storage of original structure is reduced, which is also prepared for subsequent calculation of parallel algorithms. We summarize the detailed meanings of these values as follows in Table 1

TABLE 1: Notations used in a PT.

Notations	Description
Offset	It refers to the thread index in the final password generated by the last PT. The offset of the PT-1 is 0. When the PT-1 generate 4 candidate passwords, the offset in the PT-2 is 4, which describes the shift value from the last PT.
Number of Container	The number of containers in the current PT. Namely, the number of sub-structures in the PT's structure.
Previous Guesses	The total number of passwords generated by all the previous PTs.
Total Guesses	The total number of passwords that can be generated by the current PT.
Types	The type of the specific strings in the container.
Length	The length of the specific string in the current container.
Start Index	The starting position of the specific strings among all the strings with the same type and length.
Container Size	The number of specific strings contained in the current container.

Second, we adjust the storage size of PTs to the integer times of the cache line to reduce the memory access (i.e., increase the number of PTs in one transmission). This is because when the storage size of a PT is larger than the cache lines, GPU loads the PTs with multiple times. Specifically, we use 8 bytes, 8 bytes, 1 byte and 7 bytes for Offset, Previous Guesses, Number of Container, and Total Guesses (8 + 8 + 1 + 7 = 24 bytes); For every container in the body data, we use 1 byte, 1 byte, 1 byte and 5 bytes for Type, Start Index, Length, and Container Size (1 + 1 + 1 + 5 = 8 bytes). Besides, we limit the maximum number of containers in a password to 29 since passwords with more than 29 sub-structures are rare (<0.1%) across password datasets. When we use 29 as the maximum number of containers, each PT will occupy 256 bytes, based on the following calculation:

$$24 \text{ bytes (head data)} + 8\times29 \text{ (body data)} = 256$$

Since the size of a cache line in GPUs is generally 128 bytes, our storage size of PTs is exactly twice the size of a cache line so that we can mitigate the multiple reading for PTs caused by the in-appropriate storage.

Finally, we use a smoothing technique that re-assigns the probabilities for the specific strings with the same type and length so that every containers can include the same number of specific strings, further guaranteeing a similar number of candidate passwords generated by every PTs. In this way, we can enable the threads in a warp process the same PT to generate passwords in parallel, thereby reducing the waiting delay among threads in a warp.

To this end, we sort the specific strings with the same type and lengths by descending probabilities, divides them into several groups, and re-assign the probability of the strings from the same group as the average probability across all the strings of the group. Particularly, we set the number of specific strings in a group is 2^n ($n \geq 1$), that said, the adjusted container size is 2^n. We choose the size of 2^n to guarantee that the PTs would produce 32 (2^5) integer multiples of candidate passwords. To our knowledge, a warp generally includes 32 threads. Therefore, with the 32 integer multiples of candidate passwords, a warp can process candidate passwords from the same PT. When the n becomes larger (i.e., ≥ 5), the number of candidate passwords generated by all PTs can be an integer multiple of 32; When the n becomes smaller, we should repeat some PTs to make the generated passwords reach 32. Take $n = 3$ as an example, then each container includes 8 specific strings. Therefore, the PTs coming from a structure with more than two sub-structures (the number of containers is more than two) would straightforwardly generate 64 (8×8 when the number of containers is exactly two) or more candidate passwords. For the PTs with only one container, we use the extra storage that repeats the the specific strings for 4 times to satisfy 32 threads, in order to make sure that threads in a warp can process the passwords from the same PT.

We show the number of repetition for different n in Table 2. We can conclude that when we select a smaller n, we need more repetitions, resulting in the space waste. On the contrary, when we select a larger n, our smoothing becomes coarse-grained that more specific strings share the same probabilities, affecting the final guessing performance. Further, we empirically analyze the selection of n and recommend the parameter as 3 in Section 5.3.

TABLE 2: Repetitions of the selection of n. The "-" refers to no repetitions in associated container size.

Container size	Number of containers in a PT			
	1	2	3	4
2 ($n = 1$)	16 times	8 times	4 times	2 times
4 ($n = 2$)	8 times	2 times	-	-
8 ($n = 3$)	**4 times**	-	-	-
16 ($n = 4$)	2 times	-	-	-

4.3 Parallel Algorithm

Then, we design a parallel algorithm that each thread generates one candidate password at the same time. Besides, we make the threads in a warp generates passwords from

the same PT to fully leverage parallelism. To this end, we should design two index algorithms for candidate password identification in every threads, and specific string identification that makes up the candidate password.

4.3.1 Candidate Password Identification

First, we design a two-dimensional look-up table to locate the the candidate passwords generated in each thread. In specific, we use the PT_id-Guess_id (e.g., j-k) to describe the Guess_id (k) th candidate passwords generated from the PT_id (j) th pre-terminal. That said, we introduce the *guess_id* to describe the index of the candidate password in the generation order of a PT (i.e., the first dimensional index). Then, we can summarize that the *Guess_id* can be calculated by the Formula 1:

$$\begin{aligned} Guess\_id = (Thread\_id - Offset[PT\_id] + Thread\_size) \ \% \ Thread\_size \\ + n \times Thread\_size \\ \text{where } Guess\_id \leq Total\_Guess[PT\_id] \text{ and } n \in N \end{aligned} \tag{1}$$

where the *Offset* and *Total_guesses* describe the shift value from the last PT (*Thread_id* in the last password generated the last PT) and the total number of candidate passwords generated by the current PT.

In Formula 1, we can get the thread index of the first password generated by the current PT by *Thread_id - Offset*. Here, to avoid the negative value, we also take the modulus of the total number of threads (i.e., *Thread_size*) after adding the *Thread_size*. Then, we add an integer multiple of the total number of threads to label the next candidate password with *Guess_id*, while guaranteeing the *Guess_id* cannot exceed to *Total_guesses*) since the current PT can only generates the *Total_guesses*) number of passwords. We traverse the values of *Thread_id* and *PT_id* from 1, and calculate the corresponding *Guess_id* for every *Thread_id* and *PT_id* until the *Guess_id* is greater than *Total_guesses*. Then, we start to calculate the *Guess_id* of candidate passwords generated by the next PT.

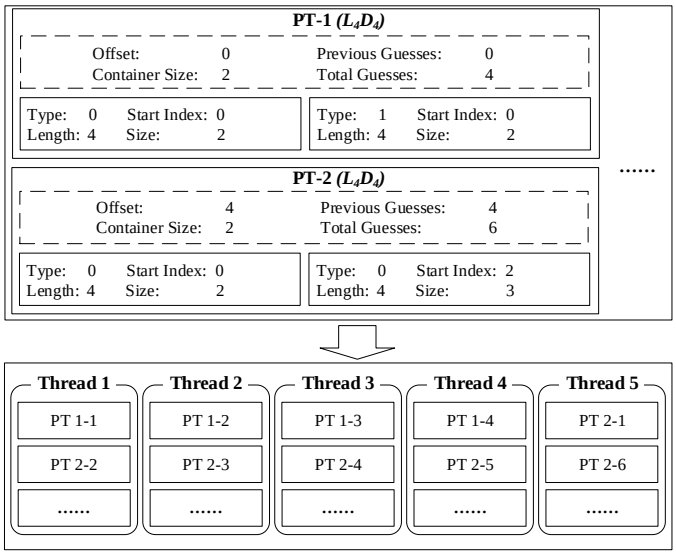


Fig. 6: The allocation of parallel generation tasks on GPU.

Then, we take the Figure 6 as an example to present the process of assigning the password generation to different threads based on the *Guess_id* index algorithm. We use the "PT *j-k*" as the result of second-dimensional lookup table to locate the *k-th* (*Guess_id*) candidate password generated from the *j-th* (*PT_id*) PT.

In this example, we assume that the *Thread_size* is 5. For the first PT-1, the shift value is 0 (i.e., *Offset*[1] is 0), and the *Total_Guesses*[1] is 4; For PT-2, the *Offset*[2] is 4, and the *Total_Guesses*[2] is 6.

Then, we calculate the *Guess_id* for every threads. For the first thread (*Thread_id* = 1), when *PT_id* is 1, we can get

$$Guess\_id = (1 - 0 + 5) \ \% \ 5 + n \times 5 = 1 + n \times 5$$

Considering that *Guess_id* should be less than or equal to *Total_Guesses* (i.e., "4" in this example), so *Guess_id* can only take 1. That said, we can locate the first index of the generated password from the first thread (*Thread_id* = 1) is "PT 1-1".

Similarly, we calculate the *Guess_id* when the *PT_id* is 2 for the first thread (*Thread_id* = 1). When *PT_id* is 2, the *Guess_id* is calculated as:

$$Guess\_id = (1 - 4 + 5) \ \% \ 5 + n \times 5 = 2 + n \times 5 \quad .$$

Similarly, we take 2 as the *Guess_id* since *Guess_id* should be less than or equal to 6. That said, we can locate the next index of the generated password for the first thread (*Thread_id* = 1) is "PT 2-2".

For every threads, we can repeat the steps above to obtain the $PT\,j{-}k$ to locate the k th candidate password from j th PT. By changing the value of *Thread_id*, we can calculate the indexes of passwords generated by other threads. Then we just should locate the specific strings for the candidate passwords from the PT $j - k$, then we can produce the candidate password by every threads.

TABLE 3: Indexes (i.e., *Str_index*) of specific strings for different *Guess_id* (we take the PT-1 in Figure 6 as an example).

Guess_id	*Str_index*
1	{1, 1}
2	{1, 2}
3	{2, 1}
4	{2, 2}

4.3.2 Specific String Identification

To generate candidate passwords, we still should locate the specific string to finally produce them. When we obtain *Guess_id* of a candidate password, we can leverage the *Guess_id* to inversely deduce the index of every specific strings stored in a PCFG model. First, we show the process of calculating the *Guess_id*. Based on the Cartesian Product in the generation methods, the four candidate passwords generated from the PT-1 (shown in Figure 6) are deduced in Table 3 (detailed in Section 3.1). Then, we can summarize the relationship between the *Guess_id* and the indexes of specific strings ($Str\_index[id]$) in table 3 into Formula 2:

$$Guess\_id = (Str\_index[1] - 1) \times Container\_size[1] + Str\_index[2] \tag{2}$$

As shown in Formula 2, for the candidate password with the *Guess_id* of 1, its *Str_index*[1] and *Str_index*[2] are both 1. The $Container\_size[Container\_id]$ refers to the total number

of the specific strings in the corresponding container of *Container_id*. Then, the example of the first line in this Formula is

$$Guess\_id = (1-1) \times 2 + 1 = 1$$

We abstract Formula 2 into the algorithm 1.

Based on the Formula 2, we can deduce the reverse algorithm that locates the index of specific strings by the modular operation. In Table 3, given the *Guess_id* of 3, the steps of calculating the indexes of specific strings in each container are as follows:

- *Initial state*: *Str_index* is {-1, -1}, *Guess_id* is 3, *Container_size* is {2, 2}.
- *Step 1*: *Str_index* becomes {-1, 1}, *Guess_id* becomes 2. *Str_index*[2] = *Guess_id* % *Container_size* = 3 % 2 = 1. Then, *Guess_id* = ⌈ *Guess_id* ÷ *container_size*[2] ⌉, that is, *Guess_id* = ⌈ 3 ÷ 2 ⌉ = 3 | 2 + 1 = 2.
- *Step 2*: *Str_index* becomes {2, 1}. The calculation ends. *Str_index*[1] = *Guess_id* % *Container_size*[1] = 2 % 2 = 0. When the result of modulus operation is 0, we need to add the number of specific strings in current container (i.e., *Container_size*). Thus, *Str_index*[1] = *Guess_id* % *Container_size*[1] + *Container_size*[1], that said, *Str_index*[1] = 2 % 2 + 2 = 2.

We abstract the above steps into Algorithm 2.

Complete design. We use the *Guess_id* calculation method to calculate *PT_id-Guess_id* to locate the candidate passwords generated by every threads. Then, we calculate the index of specific strings associated with the candidate passwords by the *Guess_id*. The *startingindex* is stored in a PT, which describes the starting position of the specific strings with the same type and length in the dictionary of all specific strings of the trained PCFG model. The shift value can be obtained by the addition of *Str_id* and the *starting_index*. We leverage the one-dimensional array with the $T \times L$ of all the starting position for the specific strings with the same type and length to obtain its position, and then obtain the specific string based on the dictionary. Finally, we concatenate each specific string to get the candidate password.

Implementation optimization of the running logic. As shown in Figure 7, existing generation logic is serially implementing the generating of passwords and processing of pre-terminals, resulting in the unnecessary overhead of waiting time. In the optimized implementation, we generate passwords and process the pre-terminals asynchronously. When GPU processes the previous batch of pre-terminals, CPU starts to generate the next batch of the pre-terminals. We use two spaces of the same size and two threads in CPU to asynchronously execute the generation and processing of the pre-terminals, whose size is equal to the total space corresponding to the pre-terminals that can be accommodated in one transmission. To avoid conflicts when different threads access the same space, we use the semaphore and the mutex by locking and unlocking the two spaces, and releasing the empty signals and full signals in turn.

5 Evaluation

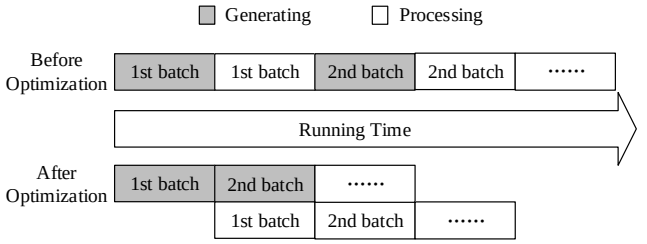


Fig. 7: Comparison of running process before and after the optimization of the running logic of `Parallel_PCFG`.

Algorithm 1 Forward algorithm for *Guess_id* calculation.

Input: *Str_index*, *Container_size*, *Number_of _Container*
Output: *Guess_id*

```
Container_id ← 1
Guess_id ← -1
while Container_id ≤ Number_of _Container do
    if Guess_id == -1 then
        Guess_id = Str_index[Container_id]
    else
        Guess_id =
        (Guess_id - 1) × Number_of _Container[Container_id] +
        Str_index[Container_id]
    end if
    Container_id += 1
end while
return Guess_id
```

Algorithm 2 Backward algorithm of *Str_id* calculation.

Input: *Guess_id*, *Container_size*, *Number_of _Container*
Output: *Str_index*

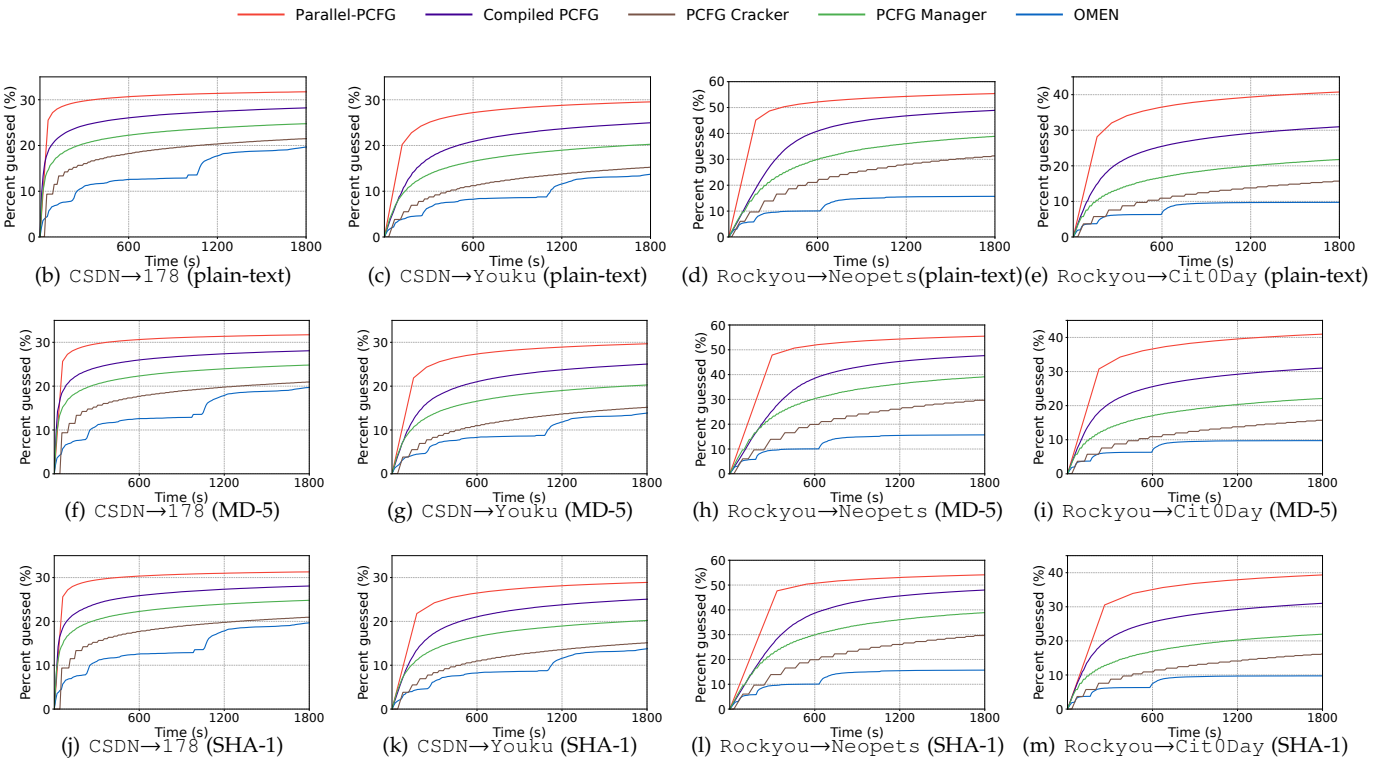
```
Container_id ← Number_of _Container
Str_index ← {}
while Container_id >0 do
    mod = Guess_id % Container_size[Container_id]
    if mod == 0 then
        mod = mod + Container_size[Container_id]
    end if
    Guess_id = ⌈ Guess_id ÷ Container_size[Container_id] ⌉
    Str_index.addFirst(mod)
    Container_id -= 1
end while
return Str_index
```

5.1 Experiment Settings

System environment. We use the computer with CPU of Intel Xcon Silver 4210 (2.20GHz, 40 cores), 128 GB memory and four GPU cards of GeForce RTX 2090 Ti.

Dataset. We use six large-scale password datasets leaked from real websites in the evaluation experiments: `CSDN`, `178`, `Youku`, `Rockyou`, `Neopets` and `Cit0Day`. These datasets are representative in terms of regional span and time span and have been widely used in relevant studies [2], [7], [20], [23], [25], [26], [28], [29], [30], [35], [36], [37]. We show the basic information of these six datasets in Table 4.

For the dataset cleaning [17], [38] , we clean up passwords containing non-printable ASCII codes and passwords with a length longer than 32 from the datasets. To evaluate the generalization ability of the guessing methods (i.e., cracking new, unseen passwords), we remove duplicate passwords with the training sets and obtain a total of 120



(b) CSDN→178 (plain-text) (c) CSDN→Youku (plain-text) (d) Rockyou→Neopets(plain-text) (e) Rockyou→Cit0Day (plain-text)

(f) CSDN→178 (MD-5) (g) CSDN→Youku (MD-5) (h) Rockyou→Neopets (MD-5) (i) Rockyou→Cit0Day (MD-5)

(j) CSDN→178 (SHA-1) (k) CSDN→Youku (SHA-1) (l) Rockyou→Neopets (SHA-1) (m) Rockyou→Cit0Day (SHA-1)

Fig. 8: Cracking rates of Parallel_PCFG in half hour.

TABLE 4: Information of six leaked real password datasets.

Dataset	Language	Year	Raw PWs	Used PWs
CSDN	Chinese	2011	6,425,243	6,422,884
178	Chinese	2011	9,071,979	3,461,974
Youku	Chinese	2016	47,607,615	25,000,000
Rockyou	English	2009	32,584,165	325,825,32
Neopets	English	2016	67,672,205	67,672,205
Cit0Day	English	2020	86,835,796	25,000,000

TABLE 5: Baseline methods

Method	Description
PCFG Cracker [1]	Python-based PCFG Generation
Compiled PCFG [2]	C-based PCFG Generation
PCFG Manager [3]	PCFG generation for Distributed Clustering
OMEN [4]	Markov-based Generation

million passwords. Besides, Hashcat usually de-duplicates the testing datasets and adjusts additional parameters when reading large-scale datasets, we show the unique number of of testing datasets in the last column in Table 4, where we also randomly select 25 million unique passwords from Youku and Cit0Day to avoid additional parameters for fair comparison. We mainly use the three testing cases: plain-text, MD-5 hash function encrypted, and SHA-1 hash function encrypted to evaluate the plain-test and cipher-text password cracking attacks.

Ethical claim. We claim that our study focuses on the overall characteristics of datasets without any analysis of individual features for the requirement of ethical practice. We obtain the encrypted datasets by encrypting the plain-text leaked datasets with corresponding hash functions, instead of trying to compromise the leaked encrypted datasets. Namely, we do not compromise the real-world unleaked datasets. We believe that our research is in line with the ethical concerns.

5.2 Experiment Comparison and Results

We empirically compare the cracking rate in terms of the comparison against the state-of-the-art data-driven models and the Hashcat-based mask and rule-based attacks.

5.2.1 *Comparison against data-driven models.*

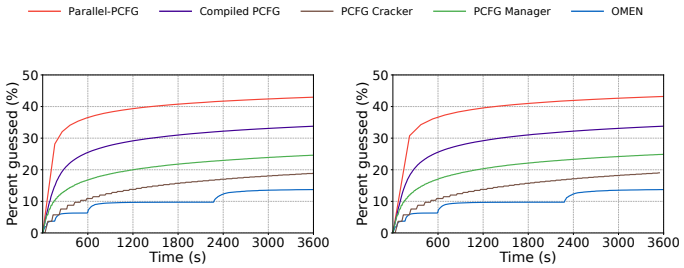
We compare Parallel_PCFG against several state-of-the-art models when deployed in Hashcat. We show the baseline models in Table 5, For PCFG-based models, we compare against the PCFG Cracker, Compiled PCFG and PCFG Manager since they are representative to accelerate password generation tasks. PCFG-manager designs a distributed strategy to generate passwords in a distributed computer cluster. Besides, we also compared with OMEN [9] because OMEN is a fast model for password generation across Markov-based models. Due to the time-consuming nature of the neural-network-based guessing model [35] (e.g. 16 days to generate more than 10^10 passwords), we have decided not to evaluate these models in this work.

For PCFG-cracker and Compiled-PCFG, we adopt the usual practice that serially transmits their generated passwords to Hashcat via *pipeline mode* for cipher-text matching. For PCFG Manager, we set the number of parallel threads as 50 since it performs best under our system environment.

For OMEN, we set default settings of 10 levels for probability smoothing to speed up the generation tasks. For `Parallel_PCFG`, we set the container size as 8. For GPU parameters, we set the grid size as 32, the block size as 1024. We empirically evaluate the impact of these parameters to the guessing performance of `Parallel_PCFG` in Section 5.3.

We show our results in Figure 8, in which we generally evaluate the cracking rates of several models within a fixed cracking time (i.e., half hour). We can find that `Parallel_PCFG` outperforms all of the baseline methods and achieves an average of 14.85% improvement than the best guessing performance of the baseline methods (i.e., Compile-PCFG). PCFG Manager, which adopts the parallel strategies across a cluster of computers (i.e., a large number of computers) in CPU, does not performs best, since they are superior with more CPUs. However, with one CPU, PCFG Manager should waste time for scheduling, thereby achieving a middle guessing performance. Therefore, it becomes a strict requirement for PCFG Manager bringing its better guessing ability into full play.

Cracking in longer time. Further, we can observe that the cracking rates on half hours can almost converge, i.e., only a little bit improvement over a longer period of time; Besides, a short period of time (i.e., half hours) would be more widely used in real world than a long period. Still, we supplement the evaluation of longer time (i.e, from half hour to an hour) in Figure 9. We argue that the cracking time of half hours can be enough to obtain a large gains, and we don't have to waster longer time on pursuing small gains.



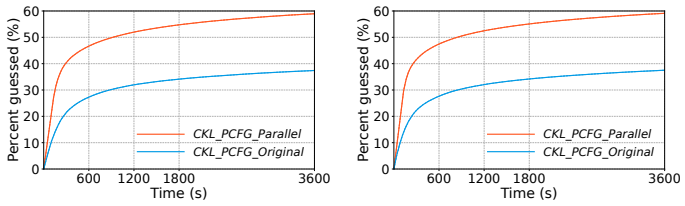
(b) `Rockyou→Cit0Day` (plain-text) (c) `Rockyou→Cit0Day` (MD-5)

Fig. 9: Comparison with the longer time (e.g., an hour) settings.

Generalization to other PCFGs. Our parallel password generation framework can be generalized to other PCFG-based methods with its grammars of pattern type and length (e.g., "L_4"). We then adapt our parallel password generation framework to the optimized PCFG models (e.g., CKL-PCFG [38]) on GPU. We conduct our experiments on two guessing scenarios and show the results in Figure 10. We observe that the *CKL_PCFG_Parallel* achieves nearly 60% cracking rates in one hour both on the plain-text and the hashed test sets, approximately 22% higher than *CKL_PCFG_Original*, showcasing the generation ability of our parallel frameworks for the optimized CKL_PCFG.

5.2.2 Comparison against mask and rule-based attacks.

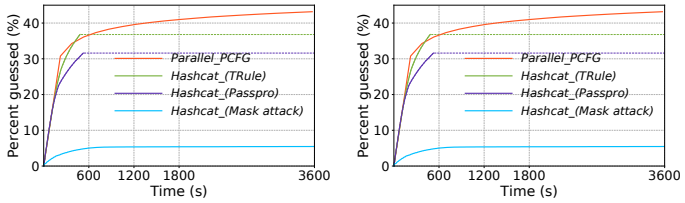
We also compare the efficiency against the mask and the rule-based attacks to gain more insights when compared



(a) `Rockyou→Cit0Day` (plain-text) (b) `Rockyou→Cit0Day` (MD-5)

Fig. 10: Performance of CKL-PCFG before and after the parallel design.

with the industrial guessing scenarios. For completeness, we compare `Parallel_PCFG` against the rule-based `Hashcat` attack with two widely-used rules and the mask attack mode. The rule-based attacks generally apply a rules-set to the password dictionary to produce password candidates. The mask attack mode consists of only searching passwords that match a specific pattern (i.e., mask like "p@**word"). We settle down two different rule-sets, Passpro and TRule [5]. Passpro is a common handwritten rule set. TRule is an optimized rule set recently proposed in 2022. Specifically, we pick the TRule with 1,728 optimized rules by Di Campi's work [5], where the rules-set is lightweight and can achieve the highest guessing performance than the other rules-set in their empirical study. The 1,728 optimized TRule rules and the masked set are both downloaded from the repository [5].



(a) `Rockyou→Cit0Day` (plain-text) (b) `Rockyou→Cit0Day` (MD-5)

Fig. 11: Comparison with the multiple `Hashcat` based attacks.

We show our results in Figure 11, where we can find that `Parallel_PCFG` achieves higher cracking rates. The mask attack mode generally yields lowest efficiency, perhaps because the trained masked passwords exhibit a mismatch with the evaluation sets of `Cit0Day` due to the straightforward configuration for generic guessing. In summary, we recommend combining data-driven models (e.g., PCFG) and the guessing softwares (e.g., `Hashcat`) be used more widely, as the pure guessing softwares (e.g., `Hashcat`) needs a summarized rules-set (e.g., TRule) to achieve a sound performance.

Reduced cracking time. We supplement to show how much cracking time our `Parallel_PCFG` can save in a longer cracking time (an hour) in Table 6, where we find that `Parallel_PCFG` achieves the similar cracking rate in an

5. https://github.com/focardi/PasswordCrackingTraining/tree/master

average of 10 minutes than that from Compiled-PCFG, indicating that Parallel_PCFG can largely reduce the time overhead, especially when testing sets becomes larger. Parallel_PCFG performs better upon Youku than 178, and similarly saves more cracking time upon Neopets than Cit0Day. Both the Neopets and Youku have the significant larger size, indicating that Parallel_PCFG can greatly benefit those guessing scenarios with larger testing sets. This key reason could be that the models upon larger testing sets requires more time in GPU processing, which can be reduced in Parallel_PCFG. When attackers face to crack a large amount of passwords, Parallel_PCFG can be extremely useful.

TABLE 6: Time (Minutes) required for Parallel_PCFG to achieve the same cracking rate that Compiled-PCFG achieves in longer cracking time (i.e., an hour).

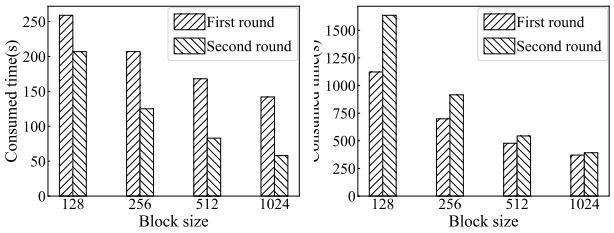
Experiments	Hash	Time	Reduction(%)
CSDN→178	Plain-text	13.58	77.37%
	MD-5	12.27	79.55%
	SHA-1	18.18	69.70%
CSDN→Youku	Plain-text	7.35	87.75%
	MD-5	7.95	86.75%
	SHA-1	9.80	83.67%
Rockyou→Neopets	Plain-text	5.00	91.67%
	MD-5	4.53	92.45%
	SHA-1	4.98	91.70%
Rockyou→Cit0Day	Plain-text	12.83	78.62%
	MD-5	10.30	82.83%
	SHA-1	13.20	78.00%

5.3 Influence of Parameters

In this section, we explore the Influence of several parameters on the performance of Parallel_PCFG. Two parameters are considered here: the block size of GPU (i.e., the number of threads in a single block) and the grid size of GPU (i.e., the number of blocks in the grid). We also evaluate the impact of the container size on the performance of Parallel_PCFG.

Influence of the block size. We set the grid size to 1 and use two container sizes of 4 and 8 for comprehensive comparision. We consider the first two rounds since the first round involves the initialization of the GPU kernel, while the second round and subsequent rounds of generation do not have the initialization step. As shown in Figure 12, the consumed time of Parallel_PCFG decreases as the block size increases. We obtain the minimum time overhead when the block size is 1024. For Parallel_PCFG, the increase in the number of threads in the block improves the ability of parallel processing, which finally reduces the consumed time. Therefore, we recommend setting the block size to 1024.

Influence of the grid size. When exploring the impact of grid size, we set the block size to 1024 based on the result of the previous section to control the variables. The experimental results are shown in Figure 13, where the consumed time of Parallel_PCFG first decreases and then increase. We can observe the minimum time when the grid size is between 16 and 64. Based on the results, we recommend setting the grid size to 32, given that a small and large



(a) The container size is 4 (b) The container size is 8

Fig. 12: Influence of block size on parallel PCFG password processing speed.

size can lead to the extra time overhead. When the grid size is too small, the total number of GPU threads is small, so each thread needs to process more passwords in each round of guessing. However, when the grid size is too large, GPU fails to fully schedule all blocks since the number of streaming multiprocessors in GPU is limited, increasing the overall time consumption.

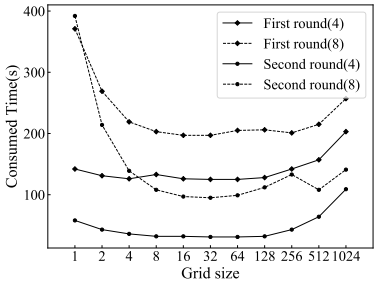
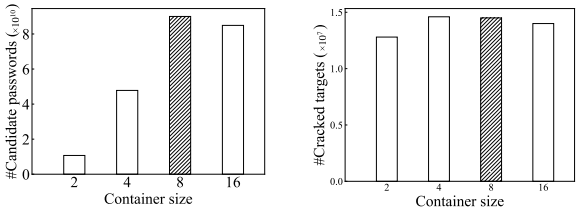


Fig. 13: Influence of grid size on Parallel_PCFG. The value in parentheses of the legend refers to the container size of each segment.

Influence of the container size. We explore the influence of the container size on the performance of Parallel_PCFG by comparing the number of processed candidate passwords and the number of cracked target passwords. We select the 15 minutes as a fixed time period, set the block size to 1024, and set the grid size to 32. We show the experimental results in Figure 14.



(a) Processed passwords (b) Cracked targets

Fig. 14: Influence of the container size on the number of processed passwords and cracked targets by Parallel_PCFG at the same time. The symbol "#" represents "the number of".

By observing Figure 14, we can find that with the increase of the container size, the number of processed

TABLE 7: Cracking rates between Parallel_PCFG with 1 GPU card and 4 GPU cards, indicating the intrinsic feature that Parallel_PCFG has reached full potential with 1 GPU card in the password generation task. Parallel_PCFG reaches its potential and fully uses the single GPU card, avoiding the complexities and costs associated with multi-GPU setups.

	1 GPU card	4 GPU cards
CSDN → 178	31.50%	32.00%
CSDN → Youku	31.00%	32.00
Rockyou → Neopets	61.00%	63.00%
Rockyou → Cit0Day	42.00%	43.00%

passwords and the number of cracked first increase and then decrease. Given that increasing the container size of 8 will not significantly improve the processing speed and the cracked targets, we settle down the container size of 8 in our experiments. We analyze the reasons below. When the container size is small, the number of candidate passwords that a pre-terminal can generate will be small, so the proportion of consumed time of generating will be smaller. More time will be used for data transmission. When the container size is large, the proportion of consumed time of GPU processing will be larger. However, the processing speed of candidate passwords would reach the bottleneck since the time of one transmission remain unchanged. Similarly, when the container size is small, Parallel_PCFG must repeat the contents of some pre-terminals to ensure that all the threads in a warp will handle the same pre-terminals, resulting in a memory space waste. When the number of fragment items is large, the position of the items in the segment changes greatly compared with its original position, resulting in a large change in the order of generated candidate passwords, leading to the reduction in the cracked targets.

Influence of more GPU cards. Parallel_PCFG is mainly designed to fully use the single GPU power with efficient workload distribution and thread management, ensuring that the single GPU is always operating at its highest potential. By focusing on single GPU efficiency, Parallel_PCFG avoids the complexities and costs associated with multi-GPU setups. We have shown that Parallel_PCFG reaches full potential in the password generation task by the comparison between 1 GPU card and 4 GPU cards in Table 7, unlocking the intrinsic feature that Parallel_PCFG almost reaches its full potential in candidate password generation with one GPU card. We believe that it is our future work to explore how to accelerate the multiple-card scenarios, which requires a thoughtful and carefully crafted design.

6 Discussion

Economic costs: We use GPU to accelerate the speed of PCFG-based methods to generate candidate passwords, which significantly reduces the time cost of guessing attacks. The price of the computer used is $8000, and the price of the GPU card used in our experiment is $1000. Before using GPU, PCFG-based methods can successfully crack 2,102 passwords per second, translating into 263 passwords per second per $1000; After using GPU, PCFG-based method can successfully crack 2,437 passwords per second, translating into 271 passwords per second per $1000. Even one second can crack 8 more passwords with the same economic cost ($1000), let alone half hour or longer times of password cracking. In summary, Parallel_PCFG can achieve similar cracking rates within around 10 minutes than that from traditional PCFG-based methods. The cost of time is shortened by 85.67% when the cost of money increases by 12.5%.

Full GPU acceleration. The running logic of Parallel_PCFG has been carefully optimized to achieve CPU-GPU collaboration, which is achieved by asynchronously generating (in CPU) and processing (in GPU) pre-terminals, ensuring GPU cores engaged, minimizing waste time and maximizing throughput. By the measures above, Parallel_PCFG achieves better schedule work (e.g., between CPU and GPU, balanced memory loading, or parallel threats) to run the cracker at full speed.

Relevance with practical takeaways and defenses. This paper offers valuable insights into password guessing performance by focusing on time-based metrics rather than simply the number of guesses. We argue that time metrics more accurately reflect real-world industrial scenarios, whereas guess counts are more commonly emphasized in academic research. Relying solely on guess counts can be misleading, as the varying cost of each guess attempt may introduce bias. Given the highlighted vulnerabilities, enforcing strict password creation policies is essential. For example, we can encourage users to employ password managers to create robust passwords. We also encourage the two-factor authentication mechanisms to strengthen the authentication.

Future work: Enhancing the speed of generating password guesses is a critical challenge in the field. We encourage more focused research on this issue and emphasize the importance of understanding how password guessing evolves over time. In future work, we will explore acceleration techniques for specific scenarios, such as masked or targeted guessing, given their unique structures and customization requirements. Besides, we plan to investigate how to better organize multiple GPU cards to enhance collaboration and significantly speed up password candidate generation.

7 Conclusion

In this paper, we manage to accelerate the password generation of PCFG-based methods using parallel techniques. We mainly propose two algorithms: a storage structure with a memory balancing mechanism and a parallel algorithm that assigns the generation tasks to different threads. Based on the above two algorithms, we propose and implement a Parallel_PCFG in the popular password cracking software Hashcat. We empirically evaluate the cracking rate of Parallel_PCFG from the perspective of cracking time period and find that Parallel_PCFG can significantly save the cracking time with several competitors. Particularly, Parallel_PCFG only takes 14.33% of time to achieve the same cracking rate achieved by the best-performing models, demonstrating that Parallel_PCFG can significantly improve the generation speed of candidate passwords and reduce the time cost of password cracking attacks.

ACKNOWLEDGMENT

This paper is supported by NSFC (No. 62172100, U1836207).

REFERENCES

[1] Sudhir Aggarmal and Charles Matthew Weir. *Using probabilistic techniques to aid in password cracking attacks*. PhD thesis, Florida State University, 2010.

[2] Jeremiah Blocki, Benjamin Harsha, and Samson Zhou. On the economics of offline password cracking. In *2018 IEEE Symposium on Security and Privacy, SP 2018, Proceedings, 21-23 May 2018, San Francisco, California, USA*, pages 853–871. IEEE Computer Society, 2018.

[3] Joseph Bonneau, Cormac Herley, Paul C. van Oorschot, and Frank Stajano. The quest to replace passwords: A framework for comparative evaluation of web authentication schemes. In *IEEE Symposium on Security and Privacy, SP 2012, 21-23 May 2012, San Francisco, California, USA*, pages 553–567. IEEE Computer Society, 2012.

[4] Alessia Michela Di Campi, Riccardo Focardi, and Flaminia L. Luccio. The revenge of password crackers: Automated training of password cracking tools. In Vijayalakshmi Atluri, Roberto Di Pietro, Christian Damsgaard Jensen, and Weizhi Meng, editors, *Computer Security - ESORICS 2022 - 27th European Symposium on Research in Computer Security, Copenhagen, Denmark, September 26-30, 2022, Proceedings, Part II*, volume 13555 of *Lecture Notes in Computer Science*, pages 317–336. Springer, 2022.

[5] Alessia Michela Di Campi, Riccardo Focardi, and Flaminia L. Luccio. The revenge of password crackers: Automated training of password cracking tools. In Vijayalakshmi Atluri, Roberto Di Pietro, Christian Damsgaard Jensen, and Weizhi Meng, editors, *Computer Security - ESORICS 2022 - 27th European Symposium on Research in Computer Security, Copenhagen, Denmark, September 26-30, 2022, Proceedings, Part II*, volume 13555 of *Lecture Notes in Computer Science*, pages 317–336. Springer, 2022.

[6] Dasio. Pcfg-manager.

[7] Xavier de Carné de Carnavalet and Mohammad Mannan. From very weak to very strong: Analyzing password-strength meters. In *21st Annual Network and Distributed System Security Symposium, NDSS 2014, San Diego, California, USA, February 23-26, 2014*. The Internet Society, 2014.

[8] Matteo Dell'Amico and Maurizio Filippone. Monte carlo strength evaluation: Fast and reliable password checking. In Indrajit Ray, Ninghui Li, and Christopher Kruegel, editors, *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, Denver, CO, USA, October 12-16, 2015*, pages 158–169. ACM, 2015.

[9] Markus Dürmuth, Fabian Angelstorf, Claude Castelluccia, Daniele Perito, and Chaabane Abdelberi. OMEN: faster password guessing using an ordered markov enumerator. In Frank Piessens, Juan Caballero, and Nataliia Bielova, editors, *Engineering Secure Software and Systems - 7th International Symposium, ESSoS 2015, Milan, Italy, March 4-6, 2015. Proceedings*, volume 8978 of *Lecture Notes in Computer Science*, pages 119–132. Springer, 2015.

[10] Cormac Herley and Paul C. van Oorschot. A research agenda acknowledging the persistence of passwords. *IEEE Secur. Priv.*, 10(1):28–36, 2012.

[11] Radek Hranický, Filip Listiak, Dávid Mikus, and Ondrej Rysavý. On practical aspects of PCFG password cracking. In Simon N. Foley, editor, *Data and Applications Security and Privacy XXXIII - 33rd Annual IFIP WG 11.3 Conference, DBSec 2019, Charleston, SC, USA, July 15-17, 2019, Proceedings*, volume 11559 of *Lecture Notes in Computer Science*, pages 43–60. Springer, 2019.

[12] Radek Hranický, Lukás Zobal, Ondrej Rysavý, Dusan Kolár, and Dávid Mikus. Distributed PCFG password cracking. In Liqun Chen, Ninghui Li, Kaitai Liang, and Steve A. Schneider, editors, *Computer Security - ESORICS 2020 - 25th European Symposium on Research in Computer Security, ESORICS 2020, Guildford, UK, September 14-18, 2020, Proceedings, Part I*, volume 12308 of *Lecture Notes in Computer Science*, pages 701–719. Springer, 2020.

[13] Gabriele Gristina Jens Steube. Hashcat: a password recovery tool.

[14] lakiw. Compiled-pcfg.

[15] lakiw. Pcfg-cracker.

[16] Enze Liu, Amanda Nakanishi, Maximilian Golla, David Cash, and Blase Ur. Reasoning analytically about password-cracking software. In *2019 IEEE Symposium on Security and Privacy, SP 2019, San Francisco, CA, USA, May 19-23, 2019*, pages 380–397. IEEE, 2019.

[17] Jerry Ma, Weining Yang, Min Luo, and Ninghui Li. A study of probabilistic password models. In *2014 IEEE Symposium on Security and Privacy, SP 2014, Berkeley, CA, USA, May 18-21, 2014*, pages 689–704. IEEE Computer Society, 2014.

[18] Jens Steube Matt Weir. Adding pcfgs to hashcat's brain.

[19] William Melicher, Blase Ur, Sean M. Segreti, Saranga Komanduri, Lujo Bauer, Nicolas Christin, and Lorrie Faith Cranor. Fast, lean, and accurate: Modeling password guessability using neural networks. In Thorsten Holz and Stefan Savage, editors, *25th USENIX Security Symposium, USENIX Security 16, Austin, TX, USA, August 10-12, 2016*, pages 175–191. USENIX Association, 2016.

[20] Bijeeta Pal, Tal Daniel, Rahul Chatterjee, and Thomas Ristenpart. Beyond credential stuffing: Password similarity models using neural networks. In *2019 IEEE Symposium on Security and Privacy, SP 2019, San Francisco, CA, USA, May 19-23, 2019*, pages 417–434. IEEE, 2019.

[21] Dario Pasquini, Giuseppe Ateniese, and Carmela Troncoso. Universal neural-cracking-machines: Self-configurable password models from auxiliary data. *CoRR*, abs/2301.07628, 2023.

[22] Dario Pasquini, Marco Cianfriglia, Giuseppe Ateniese, and Massimo Bernaschi. Reducing bias in modeling real-world password strength via deep learning and dynamic dictionaries. In Michael Bailey and Rachel Greenstadt, editors, *30th USENIX Security Symposium, USENIX Security 2021, August 11-13, 2021*, pages 821–838. USENIX Association, 2021.

[23] Dario Pasquini, Ankit Gangwal, Giuseppe Ateniese, Massimo Bernaschi, and Mauro Conti. Improving password guessing via representation learning. In *42nd IEEE Symposium on Security and Privacy, SP 2021, San Francisco, CA, USA, 24-27 May 2021*, pages 1382–1399. IEEE, 2021.

[24] Joshua Tan, Lujo Bauer, Nicolas Christin, and Lorrie Faith Cranor. Practical recommendations for stronger, more usable passwords combining minimum-strength, minimum-length, and blocklist requirements. In Jay Ligatti, Xinming Ou, Jonathan Katz, and Giovanni Vigna, editors, *CCS '20: 2020 ACM SIGSAC Conference on Computer and Communications Security, Virtual Event, USA, November 9-13, 2020*, pages 1407–1426. ACM, 2020.

[25] Blase Ur, Patrick Gage Kelley, Saranga Komanduri, Joel Lee, Michael Maass, Michelle L. Mazurek, Timothy Passaro, Richard Shay, Timothy Vidas, Lujo Bauer, Nicolas Christin, and Lorrie Faith Cranor. How does your password measure up? the effect of strength meters on password creation. In Tadayoshi Kohno, editor, *Proceedings of the 21th USENIX Security Symposium, Bellevue, WA, USA, August 8-10, 2012*, pages 65–80. USENIX Association, 2012.

[26] Blase Ur, Fumiko Noma, Jonathan Bees, Sean M. Segreti, Richard Shay, Lujo Bauer, Nicolas Christin, and Lorrie Faith Cranor. "i added '!' at the end to make it secure": Observing password creation in the lab. In Lorrie Faith Cranor, Robert Biddle, and Sunny Consolvo, editors, *Eleventh Symposium On Usable Privacy and Security, SOUPS 2015, Ottawa, Canada, July 22-24, 2015*, pages 123–140. USENIX Association, 2015.

[27] Blase Ur, Sean M. Segreti, Lujo Bauer, Nicolas Christin, Lorrie Faith Cranor, Saranga Komanduri, Darya Kurilova, Michelle L. Mazurek, William Melicher, and Richard Shay. Measuring real-world accuracies and biases in modeling password guessability. In Jaeyeon Jung and Thorsten Holz, editors, *24th USENIX Security Symposium, USENIX Security 15, Washington, D.C., USA, August 12-14, 2015*, pages 463–481. USENIX Association, 2015.

[28] D. Wang, Y. Zou, Q. Dong, Y. Song, and X. Huang. How to attack and generate honeywords. In *2022 2022 IEEE Symposium on Security and Privacy (SP) (SP)*, pages 489–506, Los Alamitos, CA, USA, may 2022. IEEE Computer Society.

[29] Ding Wang, Haibo Cheng, Ping Wang, Xinyi Huang, and Gaopeng Jian. Zipf's law in passwords. *IEEE Trans. Inf. Forensics Secur.*, 12(11):2776–2791, 2017.

[30] Ding Wang, Ping Wang, Debiao He, and Yuan Tian. Birthday, name and bifacial-security: Understanding passwords of chinese web users. In Nadia Heninger and Patrick Traynor, editors, *28th USENIX Security Symposium, USENIX Security 2019, Santa Clara, CA, USA, August 14-16, 2019*, pages 1537–1555. USENIX Association, 2019.

[31] Ding Wang, Yunkai Zou, Yuan-an Xiao, Siqi Ma, and Xiaofeng Chen. Pass2edit: A multi-step generative model for guessing edited passwords. In Joseph A. Calandrino and Carmela Troncoso, editors, *32nd USENIX Security Symposium, USENIX Security 2023, Anaheim, CA, USA, August 9-11, 2023*, pages 983–1000. USENIX Association, 2023.

[32] Ding Wang, Yunkai Zou, Zijian Zhang, and Kedong Xiu. Password guessing using random forest. In Joseph A. Calandrino and Carmela Troncoso, editors, *32nd USENIX Security Symposium, USENIX Security 2023, Anaheim, CA, USA, August 9-11, 2023*, pages 965–982. USENIX Association, 2023.

[33] Matt Weir. Practical pcfg password cracking.

[34] Matt Weir, Sudhir Aggarwal, Breno de Medeiros, and Bill Glodek. Password cracking using probabilistic context-free grammars. In *30th IEEE Symposium on Security and Privacy (S&P 2009), 17-20 May 2009, Oakland, California, USA*, pages 391–405. IEEE Computer Society, 2009.

[35] Daniel Lowe Wheeler. zxcvbn: Low-budget password strength estimation. In Thorsten Holz and Stefan Savage, editors, *25th USENIX Security Symposium, USENIX Security 16, Austin, TX, USA, August 10-12, 2016*, pages 157–173. USENIX Association, 2016.

[36] Simon S. Woo. How do we create a fantabulous password? In Yennun Huang, Irwin King, Tie-Yan Liu, and Maarten van Steen, editors, *WWW '20: The Web Conference 2020, Taipei, Taiwan, April 20-24, 2020*, pages 1491–1501. ACM / IW3C2, 2020.

[37] Yang Xiao and Jianping Zeng. Dynamically generate password policy via zipf distribution. *IEEE Trans. Inf. Forensics Secur.*, 17:835–848, 2022.

[38] Ming Xu, Chuanwang Wang, Jitao Yu, Junjie Zhang, Kai Zhang, and Weili Han. Chunk-level password guessing: Towards modeling refined password composition representations. In Yongdae Kim, Jong Kim, Giovanni Vigna, and Elaine Shi, editors, *CCS '21: 2021 ACM SIGSAC Conference on Computer and Communications Security, Virtual Event, Republic of Korea, November 15 - 19, 2021*. ACM, 2021.

[39] Ming Xu, Jitao Yu, Xinyi Zhang, Chuanwang Wang, Shenghao Zhang, Haoqi Wu, and Weili Han. Improving real-world password guessing attacks via bi-directional transformers. In Joseph A. Calandrino and Carmela Troncoso, editors, *32nd USENIX Security Symposium, USENIX Security 2023, Anaheim, CA, USA, August 9-11, 2023*, pages 1001–1018. USENIX Association, 2023.

Ming Xu is a research fellow at National University of Singapore. Her research interests lie in pursuing the beautiful convergence of chaotic data into informed insights in the usable security and privacy aspects. She received her Ph.D degree at Fudan University in 2023. Her research interest mainly includes the usable security and privacy including the password security.

Shenghao Zhang is a graduate student in Fudan University. Her research interest mainly includes the password security and system security. He is currently a member of the Laboratory of Data Analysis and Security. His research interest mainly includes password security and system security.

Kai Zhang is an associate professor at Fudan University. He received his Ph.D. at University of Science and Technology of China in 2016. He is currently a member of the Laboratory of Data Analytics and Security. His research areas include database systems, networked systems, parallel and distributed computing.

Haodong Zhang is a graduate student in Fudan University. He is currently a member of the Laboratory of Data Analysis and Security. His research interest mainly includes password security and system security.

Junjie Zhang is a graduate student in Fudan University. He is currently a member of the Laboratory of Data Analysis and Security. His research interest mainly includes password security and system security.

Jitao Yu is a graduate student in Fudan University. He is currently a member of the Laboratory of Data Analysis and Security. His research interest mainly includes password security and system security.

Luwei Cheng is a graduate student in Fudan University. He is currently a member of the Laboratory of Data Analysis and Security. His research interest mainly includes password security and system security.

Weili Han is a full Professor at Software School, Fudan University. He received his Ph.D. at Zhejiang University in 2003. Then, he joined the faculty of Software School at Fudan University. From 2008 to 2009, he visited Purdue University as a visiting professor funded by China Scholarship Council and Purdue University. His research interests are mainly in the fields of Data System Security, Access Control, and Password Security. He is now the distinguished member of CCF and the members of the IEEE, ACM, SIGSAC. He serves in several leading conferences and journals as PC members, reviewers, and an associate editor.