Prob-Hashcat: Accelerating Probabilistic Password Guessing with Hashcat by Hundreds of Times

Ziyi Huang, Ding Wang, Yunkai Zou
College of Cyber Science, Nankai University, Tianjin 300350, China; {huangziyi, wangding, zouyunkai}@nankai.edu.cn
Key Laboratory of Data and Intelligent System Security (NKU), Ministry of Education, Tianjin 300350, China
Tianjin Key Laboratory of Network and Data Security Technology, Nankai University, Tianjin 300350, China

ABSTRACT

While the academic community has proposed dozens of probabilistic password guessing models to improve the success rate of password guessing, few studies have considered the *speed of generating password guesses* (which is a crucial factor in realistic password guessing scenarios). Real-world attackers often aim to crack more passwords in less time, and the speed of these models thus becomes a significant concern. Consequently, real-world attackers tend to prefer simple heuristic methods (such as Rule attack and Mask attack) and off-the-shelf password cracking tools (such as Hashcat and John the Ripper), over academic probabilistic password guessing models, despite the latter's superior scientific flavor.

To fill this gap, we introduce an *offline guessing speed theory* for measuring the speed of generating guesses, and a *probabilistic model parallelization framework* to accelerate probabilistic guessing models. We use our theory to elucidate the acceleration principles of our framework, and provide a method for integrating this framework with Hashcat to fully leverage GPU acceleration. Our framework is scalable, generalizable for various probabilistic models and applicable to guessing scenarios that tackle multiple hashes.

To exhibit the practical value of our theory and framework, we implement two probabilistic password guessing models, PCFG and OMEN, within our framework and develop a high-speed cracking tool, Prob-hashcat. Extensive experiments on eight large real-world password datasets demonstrate the effectiveness of Prob-hashcat: (1) Both models can generate over 100 million guesses per second on a common computer and handle computing and matching hashes; (2) The average speeds of PCFG and OMEN increase by 31~104 times and 213~646 times, respectively, compared to their original speeds; (3) Within 20 minutes, the cracking rates of PCFG and OMEN increase by 12%~42% and 16%~182%, respectively, compared to their original rates, and this advantage becomes larger as the cracking time increases. We, *for the first time*, addresses the issue of employing GPU acceleration for probabilistic password models with Hashcat, highlighting that these models are significantly greater threats in real-world guessing attacks than expected.

RAID 2024, September 30–October 02, 2024, Padua, Italy
© 2024 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-0959-3/24/09
https://doi.org/10.1145/3678890.3678919

CCS CONCEPTS

• **Security and privacy → Authentication**.

KEYWORDS

Password authentication; Offline guessing attack; Probabilistic password model; Hashcat; GPU acceleration

ACM Reference Format:
Ziyi Huang, Ding Wang, Yunkai Zou. 2024. Prob-Hashcat: Accelerating Probabilistic Password Guessing with Hashcat by Hundreds of Times. In *The 27th International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2024), September 30–October 02, 2024, Padua, Italy.* ACM, New York, NY, USA, 19 pages. https://doi.org/10.1145/3678890.3678919

1 INTRODUCTION

Text-based passwords remain the most prevalent method for user authentication and maintain a considerable role across diverse computer systems, attributed to their simplicity, ease of modification, and low deployment costs [26, 29, 60, 61]. However, users tend to choose popular strings as their passwords, resulting in an uneven distribution of passwords [48]. Consequently, attackers can employ a limited set of password guesses to crack a substantial quantity of passwords by adopting a strategy that prioritizes guessing popular passwords, referred to as password guessing attacks.

Large-scale password leaks (e.g., 20 million Cutout.Pro leak [15], seven million Freecycle leak [9] and one million Callaway leak [14]) turn the risk of guessing attacks into a concrete concern. Passwords are commonly stored and leaked in the form of hashed cipher-text, requiring attackers to recover the plain-text passwords for account access. Therefore, these leaked passwords naturally become the targets of guessing attacks. Guessing attacks working against leaked passwords are known as *offline guessing attacks*. Conversely, *online guessing attacks* involve making direct password guess attempts by interacting with the target server [50]. Their rate of attempts is primarily limited by various security mechanisms at the server (e.g., rate-limiting and lockout [29]), rather than the guessing algorithm itself. This makes online guessing irrelevant to the focus of the work discussed in this paper. Unless otherwise specified, *the term guessing attacks in the following refers to offline guessing attacks*.

Guessing attacks follow two main technical routes. Academia typically employs password guessing models to generate candidate guesses. The major ones are probabilistic context-free grammar (PCFG) models [55], Markov models [39], machine learning models (e.g., RFGuess [52]), and deep learning models (e.g., FLA [37], PassGAN [30] and AdaMs [43]). Most of these models are probabilistic. By learning the linguistic patterns of real passwords from password leaks, probabilistic password guessing models assign a probability value to each candidate guess. This allows for prioritizing the candidate guesses based on their probability values.

In contrast, the industry generally employs heuristic methods for creating candidate guesses, including the Rule attack and Mask attack. Specifically, the Rule attack applies various mangling rules (e.g., insertion, deletion, reversal, or capitalization) to each word in a dictionary to generate password guesses [24] (e.g., applying the insertion rule "$1$2$3" to word "Wang" creates guess Wang123). The Mask attack conducts brute-force cracking under a template specifying password length and character set for each position [22] (e.g., the template "?l?l?l?l?d?d?d" generates passwords with the first four positions being lowercase letters and the last three digits). Several password cracking tools, exemplified by *hashcat* [18] and *John the Ripper* [11], are highly optimized to execute these methods with high efficiency, along with hash computation and matching. This makes them suitable for rapid, large-scale guessing.

Password cracking techniques are applied in fields such as data recovery, public safety, and criminal investigations [34], with an essential need for *using fixed resources and within a limited time-frame* [33]. Especially when multiple attackers are engaged in a competition, they rely on faster cracking techniques to gain a competitive edge. To better understand real-world password cracking, IT security companies like KoreLogic organize contests that simulate realistic scenarios. KoreLogic hosts the annual *Crack Me If You Can* contest, where participants are required to recover as many passwords as possible *within 48 hours*, with *no restrictions* on the algorithms or the computational power used [10].

Real-world attackers [12, 13] mainly *rely on industrial methods* [35, 43]. These heuristic methods are overly dependent on the attackers' personal experience [27], making them *challenging to generalize and automate*. Although academic probabilistic models can automatically guess popular passwords, they usually have higher computational costs [35] and commonly suffer from *slow password generation rates* (at least about 10 times slower than industrial methods, see Table 1). Moreover, since these models can only generate plaintext guesses, tools like hashcat [18] are still required to compute hashes [33], resulting in additional costs. These factors lead to an *underestimation of the applicability and threats* of probabilistic guessing models in practical password cracking scenarios.

Password cracking tools significantly increase their processing speed by leveraging *parallelization techniques*, which are designed to optimize GPU hardware utilization [27]. Such techniques provide insights into speeding up probabilistic models. Some studies have attempted to accelerate probabilistic models by using parallelization techniques [32] and integrating password cracking tools [33]. However, while these accelerated models have shown progress, they still rely on CPU for generating guesses and do not match the speed of heuristic methods (see Table 1). This implies the difficulties in effectively accelerating probabilistic models, particularly with GPU parallelization. Here we explain why it is a challenge.

Firstly, developing an effective acceleration scheme is not straightforward. In real-world password cracking scenarios, many factors (e.g., attacker's resources [12, 13], cost of hashing [38]) impact the design of an acceleration scheme for probabilistic models. For example, it is effective to accelerate the password generation process when the time taken to generate a password is much longer than the time to compute its hash. Conversely, if the situation is reversed and the hash computation substantially takes longer, then the benefits of speeding up the password generation process diminish. Therefore, the development of an effective acceleration scheme requires a *comprehensive consideration of the entire cracking process* in real-world scenarios, rather than focusing solely on the model.

Secondly, designing parallelization schemes for GPUs is more challenging compared to CPUs due to the architectural differences. The parallelization of CPUs relies on a smaller number of cores and complex control flows. These can be flexibly managed through the operating system's time-sharing mechanism, allowing effective operation even when the number of parallel threads exceeds the number of cores. In contrast, GPUs possess thousands of processing cores, requiring developers to meticulously control the distribution of threads and data in parallel to fully utilize these cores [16]. Consequently, GPU-based parallelization models necessitate algorithms capable of intelligently designing resource scheduling strategies [41] and *appropriately allocating tasks to all available threads*.

Thirdly, it is a challenge to generate massive guesses with unevenly distributed probabilities in large-scale parallelization. The difficulty stems from the fact that a substantial set of passwords is simultaneously produced and hashed. Discrepancies in probabilities among these passwords do not influence the cracking sequence, essentially rendering each password in the set equally likely. The uneven distribution of passwords is crucial because it provides a scientific foundation for guiding the order of password guesses [36, 55]. Thus, parallelized models need a rational method to assign a probability to each password during parallel generation, ensuring that the generated passwords adhere to an *uneven distribution*.

1.1 Related work

In 2005, Narayanan and Shmatikov [39] first proposed a Markov-based password guessing model. In 2009, Weir et al. [55] introduced a model based on the probabilistic context-free grammar (PCFG). In 2016, Melicher et al. [37] first utilized deep learning to construct a model based on RNN called FLA. In 2019, Hitaj et al. [30] proposed the PassGAN model, demonstrating the potential of generative adversarial networks in the field of password guessing. In 2021, Pasquini et al. [43] introduced the adaptive dynamic mangling rules attack (AdaMs), combining deep learning with Rule attack. In 2023, Wang et al. [52] proposed a model using random forests, advancing the application of machine learning in password cracking.

These password guessing models generally suffer from slow guess generation speeds. Currently, studies on accelerating guessing models predominantly target only PCFG and Markov models. In 2015, Duermuth et al. [28] introduced OMEN, which outputs guesses from Markov in descending order of probability, saving the time of sorting passwords. In 2019, Weir released a C language PCFG program, compiled_pcfg [5], capable of generating passwords faster than the earlier PCFG implemented in Python, named pcfg_cracker [23]. In the same year, Hranicky et al. [32] proposed pcfg-manager, a model designed to accelerate PCFG using CPU parallelization. In 2020, they enhanced pcfg-manage by adopting a distributed approach [33] and integrated it with hashcat [18] through a pipeline. However, these models still use CPUs to generate passwords and are difficult to apply in real-world cracking scenarios.

1.2 Our contributions

We summarize our key contributions as follows:

- **An offline guessing speed theory**. We conduct a comprehensive measurement of the speeds for 11 leading password guessing methods/models, along with 9 hashes frequently utilized in real-world applications. In light of this measurement, we propose an offline guessing theory and provide guidance for the effective allocation of computing resources in real-world password cracking scenarios.
- **A probabilistic model parallelization framework**. We address the inherent limitations in accelerating guessing processes and propose a probabilistic model parallelization framework. By applying the offline guessing speed theory, we explain the principles underlying the framework's high speeds and introduce Prob-hashcat, the *first* GPU-based probabilistic cracking tool. Specifically, we integrate two most representative probabilistic models, namely PCFG [55] and Markov (OMEN) [28], into our framework, significantly boosting their cracking speeds and success rates.
- **Extensive evaluation**. Extensive experiments on eight large real-world password datasets demonstrate the effectiveness and practicability of our Prob-hashcat. Within 20 minutes, we improve the speeds of PCFG and OMEN attacks by 31~104 times and 213~646 times, respectively. This improvement leads to an increase in cracking rates of 12%~42% for PCFG attacks and 16%~182% for OMEN attacks, and the advantage increases as the cracking time extends.

2 PRELIMINARIES

We first review the PCFG model [55] and OMEN model [28], which form the basis for our subsequent enhancements. Subsequently, we evaluate the speeds of 11 password guessing methods/models and their influencing factors. Finally, we present and categorize nine hashes commonly used in real-world applications.

2.1 PCFG

In 2009, Weir et al. [55] first proposed the probabilistic context-free grammar (PCFG) guessing model. The core concept of the PCFG model is to divide a password string into several *substrings*, each with a corresponding *tag*. By replacing each substring with its corresponding tag, one can obtain the *base structure* of the original password. The probability of a password is equal to the probability of its base structure multiplied by the probability of deriving specific substrings from each tag in the base structure. For example, the password `Wang123` can be divided into the substrings `"Wang"` and `"123"`, with `"Wang"` corresponding to the tag L_4, indicating a 4-letter segment, and `"123"` corresponding to the tag D_3, indicating a 3-digit segment. After such division, the base structure of `Wang123` is identified as L_4D_3. The probability of `Wang123` is calculated by:

$$\Pr(\texttt{Wang123}) = \Pr(L_4D_3) \cdot \Pr(L_4 \longrightarrow \texttt{Wang}) \cdot \Pr(D_3 \longrightarrow \texttt{123})$$

In the training stage, the probability of a base structure is determined by its relative frequency in the training set. Similarly, the probability of deriving a substring from its corresponding tag is obtained from statistical results of the training set or a custom dictionary. In the generation stage, the PCFG model utilizes a priority queue to generate passwords in descending order of probability. The implementation details are provided in Appendix A.

2.2 Markov and OMEN

The Markov model was proposed at CCS'05 [39] and then improved with smoothing and normalization methods [36]. In the Markov model, the conditional probability of a character depends only on its preceding characters, and a string's probability equals the product of conditional probabilities of all its characters. A detailed introduction of the Markov model is provided in Appendix B.

The Markov model is incapable of producing passwords in a descending probabilistic order and requires all generated passwords to be reordered. To address this issue, Duermuth et al. [28] proposed the Ordered Markov Enumerator (OMEN) model in 2015. The OMEN model is a variant of the Markov n-gram model using distribution-based normalization [36]. It maps probabilities into multiple levels through logarithmic calculations. For example, for a character with a conditional probability of *prob*, the OMEN model calculates its *level* with two hyperparameters, c_1 and c_2, as follows:

$$level = -\lfloor \texttt{log}(c_1 \times prob + c_2) \rfloor \tag{1}$$

The OMEN model characterizes the likelihood of a password by summing up the level of its length and the levels of all its characters. During the training stage, the OMEN model computes the probability of each length and the conditional probability of each character, similar to a standard Markov n-gram model [36], and then converts these probabilities into levels. In the generation stage, the OMEN model employs a depth-first search algorithm to generate passwords with a specific sum of character levels. By prioritizing the generation of passwords with lower sums of levels, which correspond to passwords with higher probability, the OMEN model can generate password guesses in descending order of likelihood.

2.3 Features of guessing methods/models

In real-world password cracking scenarios, an important feature of password guessing methods/models is the *speed of generating guesses*, typically measured by the average number of guesses generated per second or the average time required to generate a single guess. This speed is generally related to the algorithm type and programming language used, as well as the use of GPU acceleration. Note that using the GPU for generating guesses may limit its availability for other steps (e.g., computing hash values).

Besides the speed of generation, *whether a guessing model is probabilistic* also plays a role. Probabilistic models can be further classified based on their ability to directly generate password guesses in a probability-descending order. Generating guesses in descending order of probability allows for cracking more passwords with the same number of guesses. This is particularly crucial in cracking scenarios that tackle salted hashes, as each guess necessitates recalculating hashes for all unsolved salts. Consequently, a higher cracking rate with the same number of guesses signifies a reduction in the number of hash computations required. For probabilistic models that do not directly generate guesses in descending order, re-sorting a large number of unordered guesses can be time-consuming. Attackers need to make a wise decision about whether to use these models and, if so, whether to reorder their generated results.

We evaluate six leading trawling guessing models (i.e., PCFG [55], Markov [36], FLA [37], PassGAN [30], AdaMs [43], and RFGuess

Table 1: Features of password guessing methods/models.

Methods/Models	Accelerated variants*	Generation speed	Algorithm type	Language	GPU usage	Probabilistic	Guesses in descending order
Rule (Hashcat) [18]		2.3×10^7/s	Heuristic	C	✓		
Mask (Hashcat) [18]		1.1×10^7/s	Heuristic	C	✓		
compiled_pcfg [5]	✓	3.9×10^6/s	Statistics	C		✓	✓
AdaMs [43]		2.7×10^6/s	Deep learning	C, C++†	✓		
pcfg-manager [32, 33]	✓	2.2×10^6/s	Statistics	Golang		✓	✓
OMEN [28]	✓	2.1×10^6/s	Statistics	C		✓	✓
pcfg_cracker [23, 55]		3.6×10^5/s	Statistics	Python		✓	✓
PassGAN [30]		1.3×10^5/s	Deep learning	Python	✓		
Markov [36]		2.5×10^4/s	Statistics	Python		✓	
FLA [37]		7.5×10^3/s	Deep learning	Python	✓	✓	
RFGuess [52]		5.0×10^1/s	Machine learning	Python		✓	

*Compiled_pcfg and pcfg-manager are accelerated variants of pcfg_cracker, and OMEN is an accelerated variant of the Markov model.
†The AdaMs model employs Python for training and uses C and C++ for generating guesses.

Table 2: Classification of hash functions.

Hash	Type	Speed in hashcat	Speed in python	Iterations	Salted	Use case	Description*
MD5	Fast	6.7×10^{10}/s	1.0×10^6/s	1		General purpose	32-character digest, in hexadecimal format
SHA1	Fast	2.1×10^{10}/s	1.0×10^6/s	1		General purpose	40-character digest, in hexadecimal format
nsldaps	Fast	2.1×10^{10}/s	4.6×10^5/s	1	✓	LDAP servers	46-character in total, starts with `{SSHA}`
SHA2-256	Fast	9.2×10^9/s	9.5×10^5/s	1		General purpose	64-character digest, in hexadecimal format
SSHA-512	Fast	3.1×10^9/s	4.1×10^5/s	1	✓	LDAP servers	105-character in total, starts with `{SSHA512}`
md5crypt	Slow	2.6×10^7/s	7.5×10^3/s	1,000	✓	Unix systems, Network devices	22-character digest†, starts with `$1$`
sha1crypt	Slow	1.8×10^6/s	2.3×10^2/s	5,000	✓	NetBSD systems, Network devices	48-character in total, starts with `$sha1$5000$`
sha256crypt	Slow	8.6×10^5/s	3.3×10^2/s	5,000	✓	Unix systems	43-character digest†, starts with `$5$`
bcrypt	Slow	1.3×10^4/s	7.0×10^1/s	256	✓	General purpose	60-character in total, starts with `$2a$08$`

*Salts and iterations modify the length and prefix of hashes. Hash settings from the *Crack me if you can* contest are utilized in this experiment.
†Both md5crypt and sha256crypt functions have a fixed-length digest and variable-length salt, resulting in an unpredictable total string length.

[52]), along with the Rule and Mask attack modes in hashcat [18]. Among these, the PCFG model has two accelerated variants: compiled_pcfg [5], developed in C, and pcfg-manager [32, 33], which uses CPU parallelization for acceleration. The Markov model's variant, OMEN [28], sequences passwords in descending order during generation. Although our evaluation primarily focuses on trawling password guessing (where attackers cannot obtain additional information about passwords), our analytical approach is also applicable to targeted password guessing [42, 50, 51] (attackers can access personally identifiable information or existing passwords used on other sites) and conditional password guessing [44, 59] (attackers can obtain the length of the password or some of its characters).

The evaluation utilizes the Rockyou password dataset as the dictionary or training set for these methods/models, which contains 32,565,286 passwords and 14,325,137 unique passwords after cleaning. We employ the ClixSense password dataset as the target set, which contains 2,221,680 passwords and 1,627,806 unique passwords after cleaning. Our computational hardware consists of an Intel Xeon Gold 6230R CPU and an NVIDIA RTX 3090 GPU. The password datasets and the hardware used throughout the paper will be detailed in Section 6.1. Although the rate at which methods/models generate guesses varies with the dataset and computing resources, the number of guesses generated per second under identical conditions can be used to compare their speed. The setups of these methods/models are consistent with industry practices or original papers (see Appendix C for more details).

We summarize the features of the above 11 methods/models in Table 1. Considering algorithm types, heuristic methods offer the highest speed, followed by statistics-based models, and models based on deep learning and machine learning are usually slower. In terms of programming languages, methods/models implemented in C language typically outperform those in Python in speed. Although whether a method or a model is probabilistic or not does not directly affect its speed, Table 1 shows that non-probabilistic methods/models usually have higher speeds due to their simpler structures, while probabilistic models not ordering guesses by descending probability are found to be the slowest. This is because these models need to traverse a search tree to generate guesses, taking significant time on each character.

2.4 Hash functions classification

Hashcat [18] categorizes hashes into *fast hashes* and *slow hashes*, based on the speed of crypto within the kernel. As a rule of thumb, hashcat recommends classifying hashes as slow hashes if they involve over 100 iterations from any crypto primitive or are expected to calculate fewer than 10 million guesses per second per GPU [19]. The purpose of this classification is that if the crypto of the kernel is very fast, reading from the GPU memory would become a bottleneck. Hashcat [19] uses an *amplifier* algorithm for fast hashes to ensure that the communication latency has a negligible impact on the cracking speed [17]. We will discuss the principles and advantages of the amplifier algorithm in Section 4.4.

Apart from speed considerations, the use of salting is another factor in classifying hash functions. To verify the hit of a guess, computation of hashes is required for each distinct salt in the target set. Therefore, the computational cost of a salted hash approximates the cost of a hash with a single salt, multiplied by the number of different salts in the target set. Additionally, while the initial speed of calculating salted hashes is slow, as more passwords are cracked and the number of unsolved salts in the target set decreases, the computational speed for salted hashes will gradually increase.

We evaluate nine popular hashes in Table 2. These hashes are widely used in various scenarios requiring password storage, including LDAP servers, operating systems, and network devices. The 2023 *Crack me if you can* contest [10] employed these hashes for password encryption, highlighting their *significance in the current password cracking industry*. Although some of the hash functions (e.g., MD5, SHA1) have been proven to be insecure [46, 53], many websites still use them to store user passwords [1, 4], owing to the lack of adequate security expertise among developers [40].

We measure the speed of hashes under two implementations, utilizing an Intel Xeon Gold 6230R CPU and an NVIDIA RTX 3090 GPU. The first is an optimized implementation in hashcat [18] that leverages GPU parallelization, while the second is a CPU-based implementation via Python library functions. The iterations and salt settings are consistent with the specifications used in the *Crack me if you can* contest [10]. As shown in Table 2, in both implementations, there is a significant difference in speed between fast hashes and slow hashes, with the slowest fast hash being approximately 100 times faster than the fastest slow hash. Additionally, comparing Tables 1 and 2 reveals that the speed of password guessing models is significantly lower than that of fast hash computation. We will discuss the implications of this disparity in Section 4.1.

3 OFFLINE GUESSING SPEED THEORY

3.1 Threat model

In this paper, we focus mainly on offline password guessing, where an attacker attempts to recover plaintext passwords from a set of hashes. It is assumed that the attacker has a fixed amount of computing power and strives to recover as many passwords as possible within a given time limit, without considering the electrical and hardware costs for generating candidate guesses and computing hashes. Besides, the attacker is permitted to use all available password guessing methods/models, as well as any cracking tools. They can also obtain any publicly available password sets from online sources as training samples. This assumption of the attacker is realistic in actual cracking scenarios as shown in Section 1.

We conceptualize the entire process of an offline guessing attack as an *integrated system*, which takes hashes as input and produces cracking results as output. The time spent on the guessing attack is the total time of all operations within this system. Attackers have the option to pre-download password sets for use as dictionaries or training sets, pre-train password guessing models, and even generate candidate guesses in advance. Since these preparatory activities do not rely on the input hashes and can be completed before the offline guessing attack begins, their time expenditure may be excluded from the total time of this system.

Although generating candidate guesses in advance might reduce the time spent on the offline guessing attack process, this approach is not always the optimal strategy. On the one hand, password methods/models exhibit improved performance when the training set and target set come from similar distributions [57]. Some models (e.g., AdaMs [43]) even dynamically adjust their strategies based on already cracked passwords. Therefore, attackers may use given hashes to fine-tune their attack. On the other hand, outside of the offline guessing process, the actual time available to attackers is limited. It is impractical for them to spend several months preparing a large candidate guess dictionary. Moreover, storing candidate passwords requires significant storage space (1TB for 10^{11} guesses if each password, with its separator, averages 10 bytes), thus compromising the approach's flexibility and practicality.

3.2 Overview of offline guessing speed theory

We define an independent sub-step within the offline guessing attack process as a *Phase*. For any two different Phases, A and B, either one is part of the other, or they do not *compete for hardware resources*. Competing for hardware resources means that, at the same moment, the combined demands of Phases A and B exceed the available resources. This does not include situations where A and B utilize distinct resources, such as one using the CPU and the other using the GPU, or operating on different CPU cores. The *speed of a Phase* is calculated by dividing the number of guesses by the time spent on that Phase, which is also equivalent to the inverse of the *average time spent per guess* in that Phase. The entire offline guessing attack process is also considered a Phase, and its speed represents the overall speed of the attack. Phases can be categorized into the following three types based on divisibility:

- **Atomic Phase:** A Phase cannot be divided into two Phases.
- **Serial Phase:** A Phase can be divided into two sub-Phases, A and B, with B starting only after the completion of A.
- **Parallel Phase:** A Phase can be divided into two sub-Phases, A and B, where A and B can operate concurrently.

For instance, if an attacker first generates a candidate dictionary and then hashes each password in it, such a guessing process constitutes a serial Phase, with the dictionary generation and the hash computation representing two sub-Phases. Alternatively, if the attacker employs a pipeline, redirecting the standard output of the generation algorithm to the input of the hashing algorithm, with both algorithms running on different CPU cores, this setup of the guessing process exemplifies a parallel Phase.

Given these definitions, the speed of a parent Phase could be derived from the speeds of its child Phases. Assuming that Phase P can be divided into Phases A and B, with the average times spent per guess on Phases A and B being t_A and t_B respectively, the average time spent on Phase P per guess can be represented as:

$$t_P = \begin{cases} t_A + t_B, & \text{P is a serial Phase} \\ \mathtt{max}(t_A,\ t_B), & \text{P is a parallel Phase} \end{cases} \tag{2}$$

Note that in a parallel Phase, the initiation times of the two sub-Phases may vary. When generating a sufficiently large number of guesses, the impact of this variation on an individual guess becomes negligible. Therefore, such variations can be disregarded. Additionally, due to time consumed in manual operations and communication between program modules, the actual average time spent on Phase P per guess is greater than the theoretical t_P. The theoretical time t_P can accurately represent the actual time only if the additional delay attributed to each guess is relatively low.

3.3 Analysis of guessing attack processes

The offline guessing speed theory can be applied to analyze various attack processes in real-world password cracking scenarios. Based on the discussion in Section 3.1, we assume that attackers have excluded model training from the offline guessing attack process. Our analyses are then conducted separately for attackers who have generated a candidate password dictionary before the offline guessing process and those who have not. Attackers who have prepared a candidate dictionary execute a *dictionary attack* by merely computing hash values and matching against a target set for each guess in the dictionary. Since attackers typically employ hashcat [18] to integrate the processes of hashing and matching, a dictionary attack generally consists of an atomic Phase.

For attackers who do not generate candidate guesses in advance, one approach involves generating guesses followed by hash computations and matching, defined as a *static guessing attack*. Another approach uses a pipeline, connecting password guessing methods/models with hash calculation and matching functions, thereby enabling them to operate in parallel, termed a *dynamic guessing attack*. We will use the offline guessing speed theory to analyze the processes of offline guessing attacks under these two approaches and propose suggestions for accelerating these attacks.

Static guessing attack. A static guessing attack refers to the process where the attacker first generates all candidate password guesses, and *then* computes hash values and searches for matches in the target set. The process of a static guessing attack typically constitutes a serial Phase SA, which can be divided into a guess generation Phase G and a hash computation and matching Phase H. In a static guessing attack, the delay caused by manual operations is negligible when apportioned to each password. As a result, following Eq. 2, given the average times t_G and t_H spent on Phase G and Phase H per guess respectively, the average time t_{SA} spent on Phase SA per guess can be approximated as:

$$t_{SA} = t_G + t_H \tag{3}$$

If the attacker employs a probability model that does not directly generate guesses in descending order, there exists a variant of the static guessing attack. The serial Phase SA can be divided into a serial Phase GS and a hash computation and matching Phase H, where the serial Phase GS is further divisible into a guess generation Phase G and a sorting Phase S. This implies that the attacker first generates a candidate dictionary and then sorts all guesses in descending order of probability, before carrying out hash computations and matching. According to Section 2.3, reordering guesses can reduce the average time for computing hash values for each guess when salted hashes are applied. Assuming the average time spent on Phase S per guess is t_S, and the optimized average time spent on Phase H per guess becomes t'_H after reordering guesses. The revised average time t'_{SA} spent on Phase SA per guess is:

$$t'_{SA} = t_G + t_S + t'_H \tag{4}$$

By comparing t'_{SA} with t_{SA}, attackers can judiciously decide whether reordering guesses is required when using a probability model.

Dynamic guessing attack. A dynamic guessing attack refers to an attack that utilizes a pipeline to *connect* guessing methods/models with hash computation and matching functions. The process of a dynamic guessing attack typically constitutes a parallel Phase DA, which can also be divided into a guess generation Phase G and a hash computation and matching Phase H. Since reading guesses from a pipeline is as fast as reading from a file [21], additional communication latency can be disregarded. Therefore, according to Eq. 2, the average time t_{DA} spent on Phase DA per guess, taking into account the average times t_G and t_H spent on Phase G and Phase H per guess respectively, can be approximated as:

$$t_{DA} = \mathtt{max}(t_G,\ t_H) \tag{5}$$

In cases where Phase G and Phase H of dynamic guessing attacks are identical to those of static guessing attacks, respectively, dynamic guessing attacks will be faster than their static counterparts. Hence, attackers are *recommended* to prioritize dynamic guessing attacks while employing non-probabilistic methods/models and probabilistic models if reordering is not required.

Utilizing parallelization techniques to accelerate dynamic guessing attacks may lead to a problem of *waste of computational resources*. If Phase G is accelerated by a factor of p and Phase H by a factor of q, the average time spent on Phase DA per guess becomes:

$$t_{DA} = \mathtt{max}(\frac{t_G}{p},\ \frac{t_H}{q}) \tag{6}$$

If $\frac{t_G}{p}$ is greater than $\frac{t_H}{q}$, allocating more computational resources to Phase H (i.e., increasing the value of q), will be futile in terms of accelerating Phase DA, leading to a complete waste of these computational resources. Consequently, it is imperative for attackers to allocate an appropriate amount of computational resources to each Phase to avoid the waste of computational resources.

4 PROBABILISTIC MODEL PARALLELIZATION FRAMEWORK

4.1 Framework application scenarios

In past efforts to accelerate dynamic guessing attacks, researchers focused on refining the guess generation Phase G and used an off-the-shelf hash computation and matching Phase H [5, 28, 32, 33]. This involves employing CPU-based optimization for password guessing models and connecting them to hashcat's GPU-based hash functions through a pipeline[33]. Such approaches are effective under slow hash or when there are many salts. For example, for a target set containing hashes with 10,000 different salts, pcfg-manager [32, 33], utilizing CPU parallelization, can generate approximately 2.2 million guesses per second, surpassing the speed of most salted hash and slow hash functions (see Section 2.3 and 2.4). Hence, directly connecting high-speed password guessing models

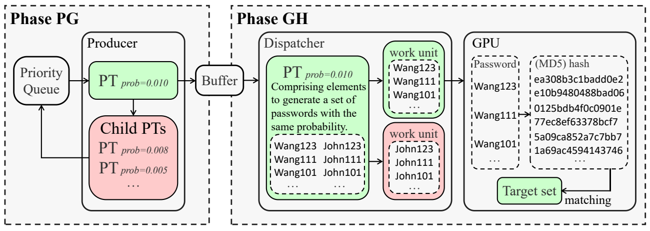


Figure 1: The workflow of probabilistic model parallelization framework. The pre-generation Phase PG generates the intermediate data structure pre-terminal (PT). The generation-hash Phase GH utilizes the PT to generate passwords within the GPU, followed by hash computation and matching.

with these hash functions in the dynamic guessing attacks does not result in significant computational resource wastage.

However, in scenarios that tackle fast hashes with no salt or few salts, GPU-based functions are nearly 10,000 times faster than the guess generation speed of pcfg-manager [32, 33]. This means that, even after CPU acceleration, the average time t_G/p spent on Phase G per guess is still significantly greater than the average time t_H/q spent on Phase H. Further, Eq. 6 reveals that this discrepancy leads to the waste of GPU computational resources in the Phase H, indicating that there is still considerable room for improvement. To address this issue, we propose a probabilistic model parallelization framework for *fast hashes with no salt or few salts*, a common scenario as shown in Section 2.4. The fundamental idea is to delegate a portion of the guess generation task to the GPU.

4.2 Framework architecture

This framework divides the offline guessing process into two Phases: a *pre-generation Phase PG* and a *generation-hash Phase GH*. These two Phases are interconnected using a producer-consumer model. Phase PG, functioning as the producer, continuously outputs an intermediate data structure, *pre-terminal* (PT), into a buffer. Phase GH, acting as the consumer, persistently extracts PTs from the buffer, utilizes them to generate final password guesses, and subsequently carries out hash computation and matching tasks.

Pre-generation Phase PG. The structure of Phase PG is illustrated on the left side of Figure 1. Phase PG employs a priority queue to generate PTs. Each PT is assigned a probability, and all password guesses derived from that PT are assigned this same probability. Within the priority queue, PTs are ordered in descending order of their corresponding probabilities. During a processing iteration, a *Producer* pops the PT with the highest probability from the priority queue, places it into the buffer, and then derives several PTs with lower probabilities to be returned to the priority queue.

Generation-hash Phase GH. The structure of Phase GH is depicted on the right side of Figure 1. This Phase comprises a *Dispatcher* and a GPU. Given that it is more typical to have a single GPU in a common computer, we take one GPU as a standard case study to show how it can be integrated into Phase GH, enabling the Dispatcher to allocate tasks to it concurrently. In cases where multiple GPUs or other computing devices, such as FPGAs, are available, we can easily add them into Phase GH. In each iteration of Phase GH, the Dispatcher retrieves a PT from the buffer and divides it into appropriately sized *work units* for allocation to the GPU based on the computing capacity. After acquiring a work unit, the GPU employs it to produce candidate password guesses, followed by hash computation and matching. The Dispatcher persistently assigns tasks to the GPU until the entire PT has been exhausted.

4.3 Framework speed analysis

The probabilistic model parallelization framework operates as a parallel Phase, with the pre-generation Phase PG and the generation-hash Phase GH connected through a low-latency buffer. According to Eq. 2, the additional delay is relatively low, so the average time t_F spent on the framework per guess can be estimated as the larger of the average times spent on these two sub-Phases per guess.

For Phase PG, assuming the average time for generating a PT and calculating its child PTs is t_{PG}. With the number of passwords that can be generated from one PT being x, the average time spent on Phase PG per guess is then $\frac{t_{PG}}{x}$. For Phase GH, it is assumed that the average time spent by the Dispatcher on each guess is t_D, and that the average time generating a final password, along with hash computation and matching, is t_{GH}. If the GPU can accelerate this process by a factor of q, then the average time spent on Phase GH per guess equates to $t_D + \frac{t_{GH}}{q}$. Therefore, the average time t_F spent on this framework per guess can be estimated as:

$$t_F = \mathtt{max}(\frac{t_{PG}}{x},\ t_D + \frac{t_{GH}}{q}) \tag{7}$$

To prevent the waste of computational resources, the speeds of Phase PG and Phase GH should be closely matched. However, this is challenging because it requires that each PT generates almost the same number of passwords. Fortunately, for attackers who utilize a GPU to accelerate password guessing models, the primary concern is to *prevent the waste of computational resources in the GPU*, rather than ensuring the CPU operates at full capacity. Attackers only need to ensure that each PT can generate a sufficient number of passwords, essentially *maximizing* x. This way, the average time spent on Phase PG per guess can be less than that spent on Phase GH, thereby effectively enhancing the utilization rate of the GPU.

4.4 Framework integrated with hashcat

In scenarios that tackle fast hashes, the *source of hashcat's acceleration is the amplifier algorithm* [20]. The amplifier algorithm divides the offline guessing attack process into two loops: a *base loop* and a *modifier loop*. Specifically, applying a *modifier* from the modifier loop to a *pre-password* from the base loop generates a password guess. During each iteration of the base loop, each GPU thread loads a pre-password from the GPU memory into GPU registers. Multiple modifiers are then applied to this pre-password to create guesses, followed by hash calculation and matching.

To illustrate the amplifier algorithm, we consider the Rule attack and Mask attack as examples. In a Rule attack, each rule from a rule set is applied to every word in a dictionary to generate candidate guesses, where words are considered as pre-passwords and the rules as modifiers. In each iteration of the base loop, each GPU thread receives a word from the dictionary and applies all rules to it. Similarly, in a Mask attack, suffixes with lengths of 1~4 are used as modifiers, and pre-passwords are the prefixes. Attaching a modifier to a pre-password completes a password guess.

Hashcat [18] does not use the amplifier in stdout mode (directly outputting the generated password guesses) but employs it when

Table 3: Advantages of the amplifier algorithm.

Methods	Speed in stdout mode (no amplifier)	Speed in MD5 mode (amplifier)
Rule [18]	2.3×10^7/s	2.2×10^9/s
Mask [18]	1.1×10^7/s	1.5×10^{10}/s

cracking fast hashes. Our experiment in Section 2.3 has measured the speed of the Rule attack and Mask attack in stdout mode. To demonstrate the advantage of the amplifier algorithm, we conduct an experiment to apply the Rule attack and Mask attack on MD5 hashed ClixSense password sets, using the same rule set and mask template. Table 3 shows that despite requiring additional hash computation and matching, the speed of the Rule attack and Mask attack increased by 100 times and 1,000 times, respectively.

One reason for this advantage is that password generation, hashing, and matching are accomplished at the register level, which keeps the memory access very low [19]. Another reason is attributed to the amplifier's ability to reduce the PCI-e latency [17, 20]. The host only needs to transfer the list of pre-passwords and modifiers to the GPU. The number of guesses attempted equals the product of the sizes of the two lists, while the transmission delay is related only to their sum. For *best practices*, since each GPU thread processes one pre-password, it relies on a large pre-password list to enhance thread utilization. Considering the relationship between the number of generated guesses and transmission delay, a large modifier list is also necessary, with optimal performance achieved when it closely matches the number of threads deployed.

The probabilistic model parallelization framework implements the amplifier algorithm in the generation-hash Phase GH. As shown in Figure 2, a pre-terminal (PT) contains a list of pre-passwords and a list of modifiers. Each pre-password can have an equal number of modifiers applied to generate the same number of password guesses. Assuming the GPU runs w threads, the Dispatcher will extract w pre-passwords and transfer them along with the modifier list to the GPU as a single work unit. The GPU then applies the entire modifier loop to each pre-password in individual threads to create guesses, subsequently computing and matching hashes.

For the specific implementation of hash computation and matching functions, our framework reuses the highly optimized code from hashcat [18]. Particularly, hashcat optimizes the algorithm for matching hashes, enabling results to be obtained in a very short time, even when the target set contains millions of different hashes. Appendix D provides a detailed description of this optimization.

5 PROB-HASHCAT: A HIGH-SPEED PROBABILISTIC CRACKING TOOL

We now develop a new password cracking tool that implements the PCFG [55] and OMEN [28] models within the probabilistic model parallelization framework. This tool is named *Prob-hashcat.* We first analyze two main challenges in the implementation of probabilistic models within this framework, subsequently providing solutions for the PCFG and OMEN models. Then, we design a universal interface for the PCFG model, enabling the PCFG mode in Prob-hashcat to be applicable to various PCFG training models (e.g., [31, 58]).

5.1 Main challenges

Designs of the pre-terminals (PTs). As explained in Section 4.3, to enhance the utilization rate of the GPU, we need to ensure that

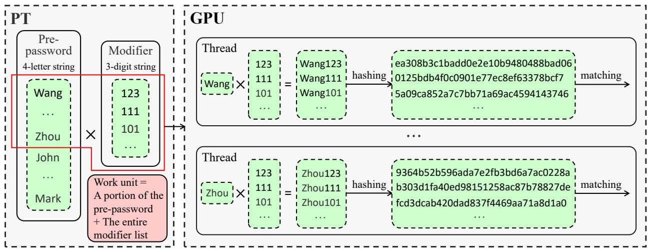

PT
Pre-password
4-letter string
Wang
Zhou
John
Mark
Modifier
3-digit string
123
111
101
Work unit = A portion of the pre-password + The entire modifier list
GPU
Thread
Wang
123
111
101
Wang123
Wang111
Wang101
hashing
ea308b3c1badd0e2e10b9480488bad06
0125bdb4f0c0901e77ec8ef63378bcf7
5a09ca852a7c7bb71a69ac4594143746
matching
Thread
Zhou
123
111
101
Zhou123
Zhou111
Zhou101
hashing
9364b52b596ada7e2fb3bd6a7ac0228a
b303d1fa40ed98151258ac87b78827de
fcd3dcab420dad837f4469aa71a8d1a0
matching

Figure 2: An example of the amplifier algorithm in our framework. Suppose the pre-password serves as the prefix of a password, and the modifier acts as the suffix. The GPU applies all modifiers to one pre-password in each thread to generate guesses, followed by hashing and matching.

each PT can generate a sufficient number of password guesses. Since the extensive number of passwords each PT generates share the same probability, and the passwords ultimately produced should follow an uneven distribution, we also need to control the size of each PT. Thus, designing PTs of appropriate size is challenging.

Implementations of the amplifier. As shown in Section 4.4, to implement the amplifier algorithm, we need to design a mechanism that applies modifiers to pre-passwords to generate password guesses. A key challenge is ensuring that each pre-password in a PT has the same list of modifiers that can be applied to generate passwords. This uniformity is crucial for dividing the PT's password generation process into a base loop and a modifier loop.

5.2 Designs of the pre-terminals

Since the passwords generated by the probabilistic model are expected to follow an uneven distribution, with fewer passwords in higher probability levels, the design of the pre-terminal (PT) should be adjusted accordingly. We allow PTs with higher probabilities to generate fewer passwords, and as the probability of a PT decreases, gradually increase the number of passwords each PT can generate. We introduce a hyperparameter, allowing more passwords to be generated from a PT as it increases. By incrementally raising this hyperparameter until an increase in PT size does not enhance the model's cracking rate, we can achieve a reasonable design.

PCFG. According to Section 2.1, the PCFG model produces passwords by replacing each *tag* in the *base structure* with a *substring*. The left side of Figure 3 shows the design of a PT for the PCFG model. To design the PT, a straightforward approach involves grouping passwords that share the same base structure and have substrings, each derived from a tag in the base structure, with identical probabilities, respectively. We design the PT to generate such group of passwords that share the same probability, and Appendix E provides the process of deriving child PTs from a parent PT.

This straightforward design encounters a problem: the number of substrings derived from a tag, each with a specific probability, is unpredictable. Thus, a PT may not be able to generate a sufficient number of passwords. We address this issue by reassigning the probabilities of consecutive substrings. The implementation details of the probability reassignment algorithm are provided in Appendix F. This algorithm ensures the adequacy of PT size and also guarantees that PTs with lower probabilities have the potential to generate a larger number of passwords. The hyperparameter x_1

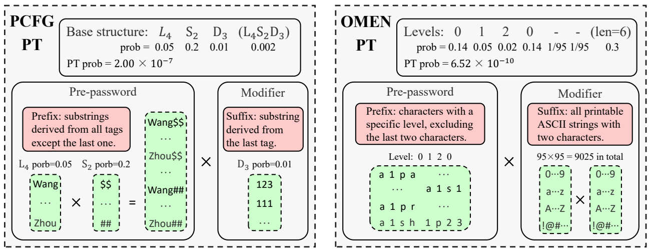

PCFG PT
Base structure: L_4 S_2 D_3 $(L_4S_2D_3)$
prob = 0.05 0.2 0.01 0.002
PT prob = 2.00×10^{-7}
Pre-password
Prefix: substrings derived from all tags except the last one.
L_4 porb=0.05
S_2 porb=0.2
Wang … Zhou
$$ … ##
Wang$$ … Zhou$$ … Wang## … Zhou##
Modifier
Suffix: substring derived from the last tag.
D_3 porb=0.01
123 111 …
OMEN PT
Levels: 0 1 2 0 - - (len=6)
prob = 0.14 0.05 0.02 0.14 1/95 1/95 0.3
PT prob = 6.52×10^{-10}
Pre-password
Prefix: characters with a specific level, excluding the last two characters.
Level: 0 1 2 0
a 1 p a … … a 1 s 1 a 1 p r … a 1 s h 1 p 2 3
Modifier
Suffix: all printable ASCII strings with two characters.
95×95 = 9025 in total
0…9 a…z A…Z !@#…

Figure 3: An example of pre-terminal (PT) designs and amplifier implementations for the PCFG and OMEN models.

in the probability reassignment algorithm, which represents the number of substrings to be reassigned in the first round, acts as a hyperparameter controlling the average size of a PT, with a larger x_1 enabling the generation of more passwords.

OMEN. According to Eq. 1, OMEN employs a negative logarithm function to map the conditional probabilities of characters into discrete *levels*. The right side of Figure 3 shows the design of a PT for the OMEN model. We can design a PT to generate passwords of a particular length, with each character having a specific *level* value. By mapping each character's *level* back to probability, the probability of the PT is calculated by multiplying the length probability with the probabilities of each individual character. Appendix E demonstrates how to derive child PTs from a parent PT.

Given that the maximum *level* is typically set to be small (e.g., 10 in [28]), each *level* contains a sufficient number of characters on average. The size of a PT is equal to the product of the number of characters belonging to a specific *level* at each position, so most PTs can generate an adequate number of passwords. Additionally, due to the use of a negative logarithm function, higher *level* values (corresponding to lower probabilities) may encompass more characters. This design inherently ensures that, as the probability diminishes, the PT can generate a greater number of passwords.

The two variables c_1 and c_2 in Eq. 1 are hyperparameters of the OMEN model. The value of variable c_2 has a negligible impact. Thus, we set it to 10^{-9}, consistent with [28]. The variable c_1 is the hyperparameter that can control the average size of a PT. Selecting a larger value for c_1 means that more characters are assigned low and medium *level*, which can increase the average size of a PT.

5.3 Implementations of the amplifier

To implement the amplifier algorithm, we need to determine the base loop and modifier loop. For both the PCFG and OMEN models, we treat the prefix generation of passwords as the base loop, with the addition of suffixes acting as the modifier loop. An example of amplifier implementations is illustrated in Figure 3.

PCFG. For the PCFG model, we assume a PT is defined based on a base structure of length l, i.e., comprising l tags. The prefix formed by the substrings derived from the first $l-1$ tags is utilized as a pre-password in the base loop, with the substring derived from the last tag acting as a modifier. Note that if the length of the base structure, l, equals one, the size of the base loop will decrease to one. This results in only one thread being utilized in the GPU to execute the modifier loop, leading to an insufficient utilization of computational resources. However, for users with high security awareness and websites that employ password policies requiring passwords to include more than one type of character [25], most passwords will possess a rich structure. The base structures of these passwords exceed a length of one, therefore our model is more effective when attacking them. Besides, we can also employ more sophisticated methods for dividing passwords, such as using the BPE algorithm to divide passwords into different chunks [58] (see Section 5.4).

OMEN. In the OMEN model, the process of generating suffixes, which involves appending the last few characters to the prefix, cannot be directly employed as the modifier loop. The reason is the number of characters with the same *level* value varies after different prefixes. Allowing each pre-password in the base loop to generate a varying number of passwords violates the principle of the amplifier algorithm, which requires applying the same modifier loop to each pre-password. This violation prevents the model from leveraging the amplifier's benefit in reducing PCI-e latency. Additionally, the requirement for each thread to process modifier loops of varying sizes leads to GPU threads being unable to synchronize.

To address this issue, we assert that the probabilities for the last two characters are equal across all characters in the character set. Assuming a PT is based on a password length of l, the first $l-2$ characters form the pre-password in the base loop and the modifier is the 2-character suffix. The number of modifiers in the modifier loop is the square of the character set size for all PTs. In this approach, if the character set includes 95 printable ASCII characters, the probability of the password `Wang**` (wildcard `*` represents any printable ASCII character) equals the product of three factors: the probability of its length, the probability of the pre-password `"Wang"`, and $(\frac{1}{95})^2$ for the last two characters as a modifier. Note that this approach will alter many passwords' probabilities, yet the passwords generated still adhere to an uneven distribution. Furthermore, we will demonstrate in Section 6 that such adjustments significantly enhance the cracking rate of the OMEN model, even though some low-probability passwords are reassigned higher probabilities.

5.4 Training interface of PCFG

Following the initial proposal of the PCFG model in 2009 [55], researchers have made major improvements to PCFG, which included integrating semantic patterns [47], incorporating keyboard patterns [31], and using the Byte Pair Encoding (BPE) algorithm to segment passwords into chunks [58]. In reality, most of these improvements affect only the training stage, involving modifications to the PCFG grammar tags through changes in how passwords are divided into substrings. Subsequent implementations of the PCFG model, such as pcfg_cracker [23], compiled_pcfg [5], and pcfg-manager [8], have also adopted such improvements to enhance cracking efficiency, which involve the division of password strings into substrings that represent years, email addresses, keyboard patterns, and so on.

As long as the mechanism of deriving substrings from tags remains unchanged, the generation process of the PCFG model will stay the same. Therefore, if our PCFG model in Prob-hashcat can directly take the probabilities of the base structures and the probabilities of substrings derived from each tag as inputs, it can conduct cracking attacks based on the input grammar. This allows it to adapt to various PCFG models. When researchers propose a new PCFG model, they only need to implement the training component of the new model for Prob-hashcat, enabling it to be integrated with Prob-hashcat for efficient high-speed cracking.

Table 4: Basic information about eight password datasets.

Dataset	Web service	Language	When leaked	Removed	Total passwords	Unique passwords
Dodonew	E-commerce & Gaming	Chinese	Dec. 2011	65,963	16,192,928	10,126,610
CSDN	Programmer forum	Chinese	Dec. 2011	739	6,427,538	4,037,153
Tianya	Social forum	Chinese	Dec. 2011	183,450	30,717,791	12,884,759
Taobao	E-commerce	Chinese	Feb. 2016	17,400	15,055,019	11,627,083
RockYou	Social forum	English	Dec. 2009	10,214	32,565,286	14,325,137
LinkedIn	Job hunting	English	Jan. 2012	44,279	54,612,336	34,315,848
000Webhost	Web hosting	English	Oct. 2015	52,427	15,247,163	10,580,118
ClixSense	Paid task platform	English	Sep. 2016	366	2,221,680	1,627,806

To ensure the performance of cracking, Prob-hashcat requires that a tag can only derive substrings of a fixed length. For example, the tag L_6 can only produce substrings that are 6 characters long. This restriction allows for saving host memory and GPU memory when storing the model's grammar, and ensures that threads within a single GPU warp have the same execution time. Under this restriction, we implement three different PCFG models.

Base PCFG. We adopt the method from [55], dividing passwords into substrings with tags belonging to three categories: L, D, S (letters, digits, special characters). The tag for each substring is determined by its category and length. For example, a substring "Wang" corresponds to the tag L_4, a substring "123" corresponds to the tag D_3, and a substring "!!" corresponds to the tag S_2.

Complex PCFG. Inspired by [31, 56], we expand the categories of L, D, S tags to include tags for years, URLs, popular words, and keyboard patterns. The method for dividing passwords and the priorities in handling conflicts are consistent with [56].

Chunk PCFG. The CKL_PCFG model proposed in [58] utilizes the BPE algorithm [45] to divide passwords, categorizing the substrings into seven types of tags: L, D, S, U, DM, TM, FM. Among these, L, D, S, U represent substrings that contain exclusively lowercase letters, digits, special characters, and uppercase letters, respectively. DM, TM, FM correspond to substrings that include a mix of 2, 3, and 4 types of characters, respectively. The hyperparameter *avg_len* of CKL_PCFG is chosen to be 4.5, as recommended in [58].

To further enhance the cracking capability of PCFG attacks, if a password corresponds to a base structure of length 1, we split it into two parts when counting its base structure. For instance, if the original base structure of the password "Wangpass" is L_8, we treat its base structure as L_4L_4. This design is implemented because a base structure of length 1 significantly reduces the speed of the PCFG attack (see Section 5.3), which we need to avoid.

6 EXPERIMENTS

6.1 Experimental setup

Datasets. We evaluate the speed and cracking efficiency of our Prob-hashcat tool based on eight large, real-world password datasets (see Table 4), containing over 170 million passwords in total. Four of these datasets are from Chinese sites, while the other four are from English sites. When cleaning password datasets, we remove passwords under 4 or over 32 characters, as well as those including symbols beyond the 95 printable ASCII characters. Passwords under 4 characters or containing non-printable ASCII characters are likely to be generated by errors, while those over 32 characters may be generated by programs rather than created by users. These removed passwords constitute a very small fraction of the total.

Ethical considerations. Although these datasets are publicly available and widely used in password security research [37, 49–51], they are considered private data. Hence, we report only aggregated statistical information to avoid causing additional harm to the victims. While these datasets could be utilized by attackers to crack users' passwords, our use aims to aid the academic community in understanding potential threats to password security, thereby facilitating the development of defense mechanisms. We have obtained approval from our center's IRB for this evaluation. Since these datasets are widely used and available from public sources, the results of this work are replicable.

Experimental equipment. The specific configuration of the computer used for the experiments is detailed as follows:

- **CPU:** We utilize an Intel Xeon Gold 6230R processor with 26 cores and 52 threads, operating at a base frequency of 2.1 GHz and a maximum turbo boost of 3.9 GHz.
- **GPU:** The GPU used in our experiments is the NVIDIA RTX 3090, featuring 24 GB of GDDR6X memory, 10,496 CUDA cores, and a boost clock of 1.7 GHz.
- **Memory:** We are equipped with a total of 256GB of RAM, utilizing DDR4 modules at a frequency of 3200MHz.

Attacking scenarios. We utilize eight scenarios to model real-world attacks. The first four scenarios involve using Dodonew and CSDN as training sets to attack Tianya and Taobao, respectively, representing cross-site attacks in a Chinese-language environment. The latter four scenarios use RockYou and LinkedIn as training sets to attack 000Webhost and ClixSense, respectively, representing cross-site attacks in an English-language environment.

Hyperparameters selection. We experimentally evaluate the impacts of the hyperparameters x_1 for PCFG and c_1 for OMEN within Prob-hashcat, which are described in Section 5.2. For the PCFG model, we conduct experiments with the base PCFG training model for x_1 values of 1,024, 2,048, 4,096, and 8,192, with the results shown in Figure 4. For the OMEN model, experiments are conducted for c_1 values of 5.5, 6.0, 6.5, 7.0, and 7.5, with the results shown in Figure 5. We present the cracking rate of the models within 20 minutes in two scenarios: CSDN→Taobao and LinkedIn→ClixSense ($A \rightarrow B$ means using A as the training set and B as the hashed target set), using MD5 hash functions. The outcomes in other scenarios or over longer durations are consistent with these two cases.

Figure 4 shows that, as x_1 increases, our PCFG model can crack 7.1%~8.4% more passwords within 20 minutes. However, the improvement in the cracking rate when x_1 is raised from 4,096 to 8,192 is minimal, especially in contrast to the boost observed between 1,024 and 4,096. As shown in Figure 5, increasing c_1 also generally enhances the OMEN model's cracking capability, although this improvement is not consistently stable. The cracking performances at c_1 values of 7.0 and 7.5 are very close, and at certain moments, the cracking rate of the model with c_1 equal to 7.0 may even exceed that of the model with c_1 equal to 7.5. Thus, acceptable ranges for the parameters are 4,096 to 8,192 for x_1 and 7.0 to 7.5 for c_1.

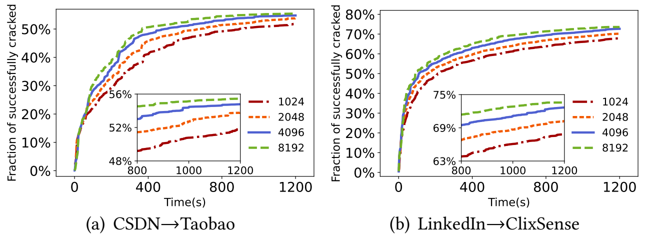


Figure 4: Impacts of the hyperparameter x_1 for PCFG model. The reasonable range for x_1 is between 4,096 and 8,192.

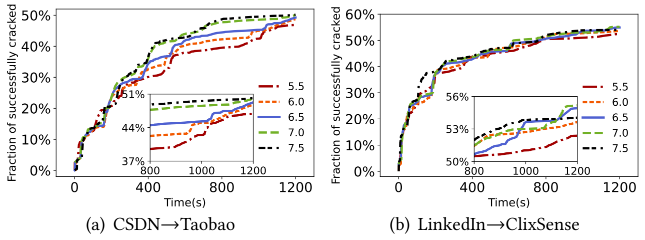


Figure 5: Impacts of the hyperparameter c_1 for OMEN model. The reasonable range for c_1 is between 7.0 and 7.5.

The increment in x_1 and c_1 leads more passwords have the same probability. To balance the cracking rate and the disruption to the uneven distribution, we set x_1 to 4,096 and c_1 to 7.0 for subsequent experiments. Although the optimal hyperparameters highly depend on the training model, training set, hash functions, and target set employed, suboptimal hyperparameters can still provide sufficient cracking capability for Prob-hashcat in real-world scenarios. Therefore, we recommend using the selected hyperparameter values in practice, as they generally do not need to be changed.

6.2 Comparison with dynamic guessing attacks

We compare Prob-hashcat's PCFG (including base, complex, and chunk PCFG models) and OMEN attacks with their main counterparts: attack processes using pcfg_cracker [23], compiled_pcfg [5], pcfg-manager [8], and the original OMEN model [6, 28]. The setups of these models are described in Appendix C. Among the 11 leading methods/models evaluated in Section 2.3, only these four are probabilistic models that generate guesses in descending order. They can directly connect with hashcat [18] to execute dynamic guessing attacks, and Prob-hashcat is proposed to address the limitations of these models in wasting GPU computational resources during dynamic guessing attacks (as discussed in Section 4.1).

The target sets are hashed using two fast hash functions: MD5 and SHA2-256. We evaluate the effectiveness of an attack process by measuring its speed and cracking rate within a certain timeframe. The runtime of each experiment is set to 20 minutes, which is sufficient for our models to crack nearly all popular passwords. Therefore, stopping at this point would increase the attacker's profit. Additionally, our models' speed far exceeds their counterparts, meaning that extending the duration would disproportionately amplify Prob-hashcat's advantage in cracking rates.

We first present the experimental results under the MD5 hash. For the speed experiments, given that the trend of speeds among the eight attack processes are consistent across all scenarios, we only

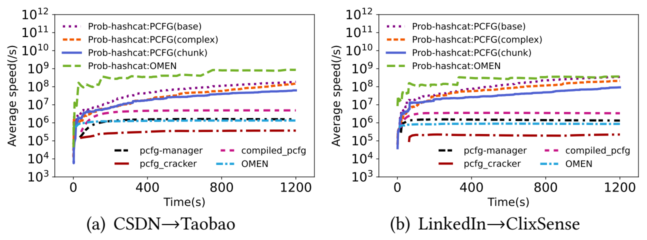


Figure 6: Comparison of speed by Prob-hashcat attack processes against dynamic guessing attack processes under MD5.

depict the results for the CSDN→Taobao and LinkedIn→ClixSense scenarios in Figure 6. Considering the possible thousands-fold differences in speed between various attack processes, the vertical axis is displayed on a logarithmic scale. The gradual increment of the size of the pre-terminal (PT) leads to a corresponding rise in the speed of four attack processes in Prob-hashcat. It can be observed that the OMEN attack stabilizes earlier, while the speeds of the three PCFG attacks are still increasing at 20 minutes. Conversely, the speeds of the four dynamic guessing attacks using CPU-based models remain consistently stable at lower levels.

Figure 10 and Table 5 in Appendix G present the results across all eight scenarios. After 20 minutes, the speed of the OMEN attack in Prob-hashcat reaches 210 to 865 million guesses per second, which represents 213 to 646 times faster than the speed of dynamic guessing attacks using the original OMEN model [28]. Likewise, the fastest PCFG attack in Prob-hashcat approaches 133 to 352 million guesses per second, achieving a speed 31 to 104 times greater than that of the dynamic guessing attack using the most efficient existing PCFG model (i.e., compiled_pcfg [5]). Given that the speeds of PCFG attacks in Prob-hashcat continue to increase beyond 20 minutes while the speeds of dynamic guessing attacks remain constant, their disparity will increase with longer cracking durations.

For the experiments on the cracking success rate, we display the experimental results for all eight scenarios in Figure 7. Additionally, Table 6 in Appendix G provides specific cracking rates for each attack process in every scenario. Figure 7 reveals that within 20 minutes, across all eight scenarios, the attack processes from Prob-hashcat consistently show optimal performance. Specifically, Prob-hashcat's base PCFG attack is superior in five scenarios, while the complex PCFG attack and the chunk PCFG attack in Prob-hashcat excel in two scenarios and one scenario, respectively.

As shown in Figure 7, our improvements to the OMEN model are particularly significant. After 20 minutes, the cracking rate of the OMEN attack in Prob-hashcat increases by 16%~182% compared to the dynamic guessing attack using the original OMEN model [6, 28]. For PCFG, in each scenario, the best-performing PCFG attack in Prob-hashcat improves the cracking rate by 12%~42% compared to the best PCFG dynamic guessing attack. Furthermore, even the least effective PCFG attack in Prob-hashcat can crack 3%~42% more passwords compared to the best PCFG dynamic guessing attack.

To demonstrate the superiority of Prob-hashcat's attack processes under different hashes, we conduct the same speed and cracking rate experiments for eight scenarios under SHA2-256. Figure 12 in Appendix G shows the results. According to Table 2, the speed of SHA2-256 is slightly slower than that of MD5. However,

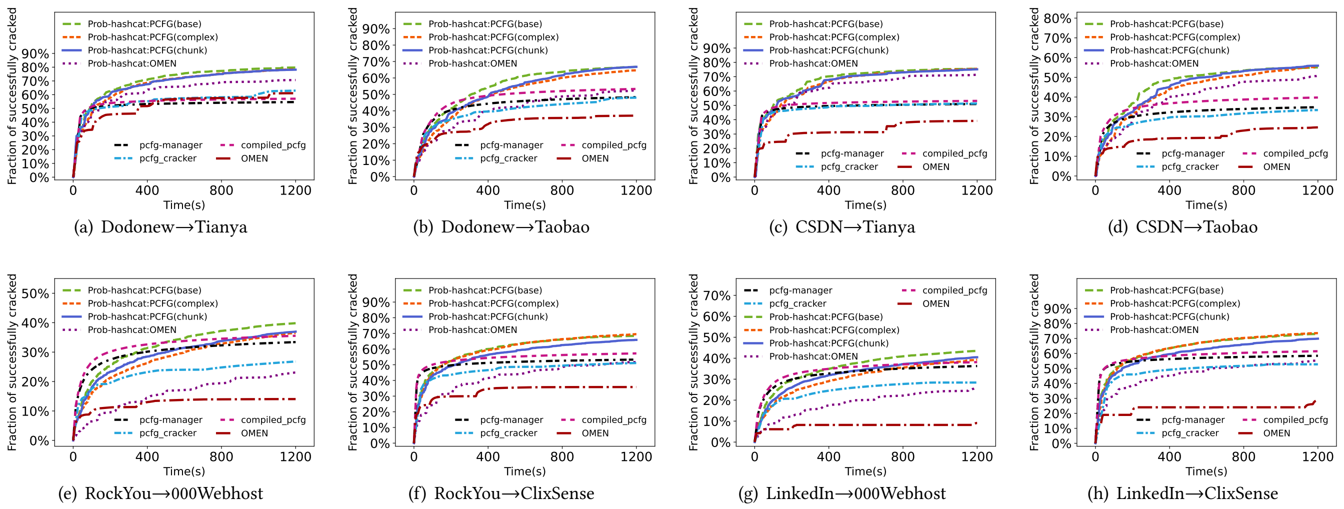


Figure 7: Fraction of passwords cracked by four Prob-hashcat attack processes compared with four dynamic guessing attack processes under MD5 across all eight scenarios. In every scenario, the most effective attack process originates from Prob-hashcat.

hashcat [18] can still compute $9.2*10^9$ SHA2-256 hashes per second, which is faster than the speed of all attack processes in experiments under MD5 hash (see Table 5). In theory, the adoption of SHA2-256 hash would not change the conclusions drawn from the experiments under MD5 hash. The experimental results confirm this, as using SHA2-256 hash has virtually no effect on the speed and cracking rate of each attack process in every attack scenario. The relative performances of the attack processes under SHA2-256 hash remain consistent with those observed under MD5 hash.

6.3 Comparison with the Rule attack

To demonstrate the application of Prob-hashcat, we compare Prob-hashcat with the Rule attack in hashcat [18]. The Rule attack uses two built-in rule sets in hashcat: generated.rule and generated2.rule, which are also used in [43]. Generated.rule and generated2.rule contain 14,350 and 55,349 rules respectively. Although generated.rule has fewer rules, its quality is higher than that of generated2.rule. We do not evaluate the Mask attack, as it is essentially a brute-force attack and less effective in practical scenarios. Additionally, we do not evaluate deep learning models because these models require the use of GPUs to generate passwords, making it impractical to execute dynamic guessing attacks and thus unsuitable for a fair comparison with Prob-hashcat. Furthermore, these models are slow and do not offer advantages in scenarios that tackle fast hashes.

When cracking MD5 hashes, the Rule attack can attempt 2.2 billion guesses per second (see Table 3), which is 10 times faster than Prob-hashcat. Therefore, Prob-hashcat struggles to gain an advantage in this scenario. As shown in Figure 8, the Rule attacks quickly exhaust all guesses and achieve a high cracking rate. Prob-hashcat, on the other hand, gradually increases its cracking rate over 20 minutes, eventually surpassing the Rule attack using generated.rule, but still not matching the Rule attack using generated2.rule.

According to Eq. 7, the speed of Prob-hashcat is t_D+t_{GH}/q, while the Rule attack's speed is only limited by t_{GH}/q. When tackling slightly slower fast hashes or target sets with a limited number of salts, their speeds become closer, allowing the probabilistic model's

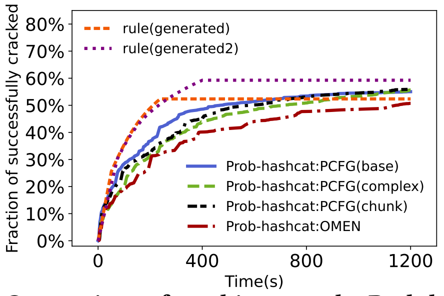
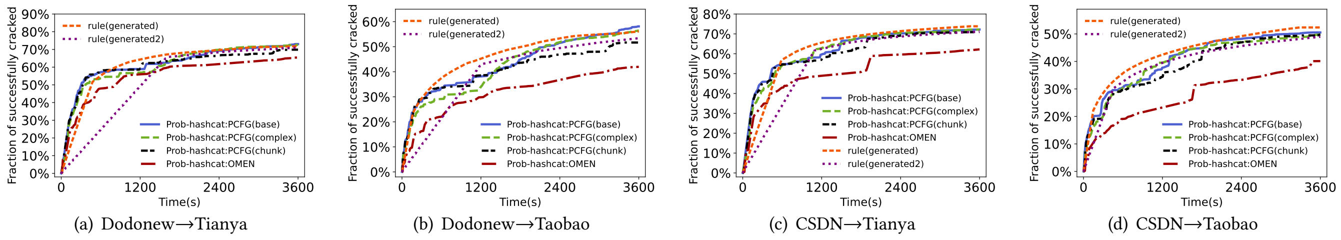


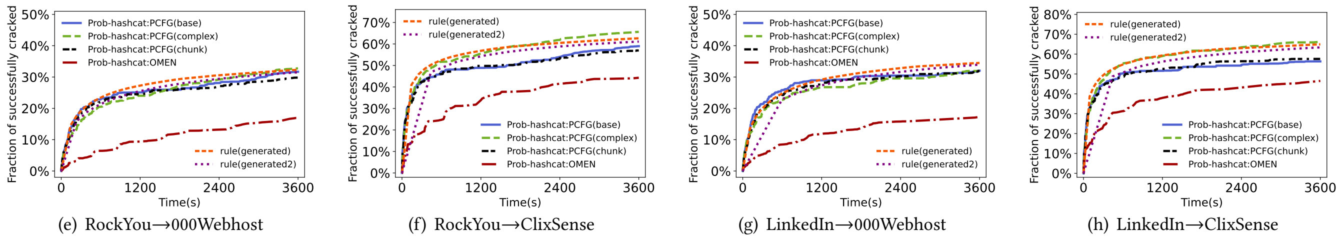
Figure 8: Comparison of cracking rates by Prob-hashcat attack processes against Rule attacks using generated.rule and generated2.rule under MD5 in the CSDN→Taobao scenario.

higher cracking rate under the same guess number to take effect. We use the salted MD5 to hash the target set, and the number of different salts is limited to 100. We do not sample 100 entries from the target set. Instead, we randomly choose one of the 100 salts when hashing each password to avoid sampling errors. Due to using a slower hash function, we extend the runtime to one hour, which aligns with the time it takes for the smallest combination (CSDN+generated.rule) to exhaust all possible guesses.

Figure 9 presents a comparison of the cracking rates between the four attack processes in Prob-hashcat and the two Rule attacks. Table 8 in Appendix G provides the cracking rates for each attack process. Compared to the Rule attack using generated2.rule, Prob-hashcat cracks 1.5%~8.9% more passwords in 7 out of 8 scenarios. Even with the higher quality generated.rule set, Prob-hashcat still achieves higher cracking rates in 5 out of 8 scenarios, with an increase of 1.4%~4.8%. This indicates that when dealing with slightly slower fast hashes or salted hashes with few different salts, Prob-hashcat is generally more effective than Rule attacks, with the base PCFG and complex PCFG attacks performing the best. Furthermore, due to the limited number of guesses that a Rule attack can attempt, probabilistic guessing models can continue cracking after the Rule attack has concluded, which provides an additional advantage.



(a) Dodonew→Tianya (b) Dodonew→Taobao (c) CSDN→Tianya (d) CSDN→Taobao



(e) RockYou→000Webhost (f) RockYou→ClixSense (g) LinkedIn→000Webhost (h) LinkedIn→ClixSense

Figure 9: Fraction of passwords cracked by four Prob-hashcat attack processes compared with Rule attacks under salted MD5 across all eight scenarios. The two Rule attacks use generated.rule and generated2.rule as their rule sets, respectively.

7 DISCUSSION

We now discuss the scalability of the probabilistic model parallelization framework to demonstrate how it can be applied to more practical cracking scenarios. Additionally, we explore potential directions for future improvements to the framework.

Scalability. In the design of the framework, we have provided it with significant scalability. On the one hand, it can utilize the optimized hash computation and matching functions within hashcat [18], making it applicable to nearly all scenarios that tackle fast hashes with no salt or few salts. On the other hand, our framework could also be adapted to more complex guessing models (e.g., RFGuess [52]), as long as it is possible to design sufficiently large pre-terminals (PTs) and divide the process of generating passwords into base loops and modifier loops (see Section 5.1).

Additionally, while we only implement two trawling guessing models within the probabilistic model parallelization framework in this paper, our framework is equally *applicable to targeted password guessing*. Wang et al. [50] in 2016 proposed the TarGuess-I and TarGuess-II models for cracking passwords in targeted guessing scenarios where the attacker has the victim's personally identifiable information (PII) and old passwords respectively. We can directly execute TarGuess-I [50] under our Prob-hashcat's PCFG mode through the PCFG training interface (see Section 5.4), simply by customizing a specific PCFG grammar for each target, where each PII corresponds to a tag. Although TarGuess-II [50] cannot be directly connected to our Prob-hashcat, it has a similar model architecture to PCFG [55] and can also be implemented within the probabilistic model parallelization framework after modifications.

Future improvements. Although our framework is general, there are still some models, such as deep learning models (e.g., FLA [37]), that are not suitable for this framework. Deep learning guessing models inevitably compete for GPU resources with the hash computation process during password generation. This observation suggests that we can further extend our framework by introducing more complex resource allocation mechanisms, allowing these accelerations to be applied to deep learning models as well.

Another direction for improvement of our framework is to make it suitable for multiple GPUs and to fully utilize the performance of all available GPUs. While we could simply allow the Dispatcher in the generation-hash Phase GH to assign tasks to multiple GPUs simultaneously (see Section 4.2), there are several limiting factors. Due to reasons such as thermal and power limits, as well as competition for bus bandwidth among multiple GPUs, it cannot guarantee that the cracking speed with multiple GPUs will be higher than that with a single GPU. Therefore, the framework may require more sophisticated scheduling methods to take advantage of all available GPUs. We leave the design of these methods to future work.

8 CONCLUSION

This paper applies GPU parallelization techniques to probabilistic password guessing models and, for the first time, introduces a high-speed probabilistic cracking tool called Prob-hashcat. This tool enables probabilistic password guessing models to attempt over 100 million guesses per second on a common computer. Our improvements are based on a universal probabilistic model parallelization framework, and we provide a theoretical explanation of the acceleration principles using an offline guessing speed theory. Extensive experiments with eight real-world datasets demonstrate the effectiveness of Prob-hashcat. Our results reveal that probabilistic password guessing models, once improved, can pose a more significant threat to password security than expected. This work provides a new perspective on password guessing research and opens up a new direction for improving password guessing models.

ACKNOWLEDGMENTS

The authors are grateful to the anonymous reviewers for their invaluable comments. Ding Wang is the corresponding author, and see more at https://wangdingg.weebly.com/ . This research was in part supported by the National Natural Science Foundation of China under Grants Nos. 62172240 and 62222208, and Natural Science Foundation of Tianjin, China under Grant No. 21JCZDJC00190.

REFERENCES

[1] 2017. *1 in 5 Websites Still Use SHA-1: Report.* https://bit.ly/3Y7ks6S.
[2] 2018. *neural_network_cracking.* https://bit.ly/4f9Xi5R.
[3] 2018. *PassGAN.* https://github.com/brannondorsey/PassGAN.
[4] 2019. *A quarter of major CMSs use outdated MD5 as the default password hashing scheme.* https://zd.net/46dltMI.
[5] 2019. *compiled_pcfg.* https://github.com/lakiw/compiled-pcfg.
[6] 2019. *OMEN.* https://github.com/RUB-SysSec/OMEN.
[7] 2021. *adams.* https://github.com/TheAdamProject/adams.
[8] 2021. *pcfg-manager.* https://github.com/Dasio/pcfg-manager.
[9] 2023. *7 Million Users Possibly Impacted by Freecycle Data Breach.* https://bit.ly/4cO86EZ.
[10] 2023. *Crack Me If You Can Contest.* https://contest-2023.korelogic.com/intro.html.
[11] 2023. *John the Ripper.* https://github.com/openwall/john.
[12] 2023. *Korelogic's CMIYC Write-up.* https://blog.cynosureprime.com/.
[13] 2023. *Team hashcat event writeups and tools.* https://bit.ly/3Ye5Jaf.
[14] 2023. *Topgolf Callaway Brands hacked, over a million golfers exposed.* https://bit.ly/3ShkK7A.
[15] 2024. *20 million Cutout.Pro user records leaked on data breach forum.* https://bit.ly/3y55ZOc.
[16] 2024. *CUDA C Programming Guide.* https://bit.ly/3A0rQqs.
[17] 2024. *Does the PCI-Express speed have any influence on cracking speed?* https://bit.ly/3J9X4wM.
[18] 2024. *hashcat.* https://github.com/hashcat/hashcat.
[19] 2024. *Hashcat Plugin Development Guide.* https://bit.ly/49rF8bD.
[20] 2024. *How to create more work for full speed?* https://bit.ly/3JcwGCw.
[21] 2024. *Is piping a wordlist slower than reading from file?* https://bit.ly/3xujoi4.
[22] 2024. *Mask Attack.* https://hashcat.net/wiki/doku.php?id=mask_attack.
[23] 2024. *pcfg_cracker.* https://github.com/lakiw/pcfg_cracker.
[24] 2024. *Rule-based Attack.* https://bit.ly/4cRj8tf.
[25] Suood Alroomi and Frank Li. 2023. Measuring Website Password Creation Policies At Scale. In *Proc. ACM CCS 2023.* 3108–3122.
[26] Joseph Bonneau, Cormac Herley, Paul C Van Oorschot, et al. 2015. Passwords and the evolution of imperfect authentication. *Commun. ACM* 58, 7 (2015), 78–87.
[27] Alessia Michela Di Campi, Riccardo Focardi, et al. 2022. The revenge of password crackers: Automated training of password cracking tools. In *Proc. ESORICS 2022.* 317–336.
[28] Markus Dürmuth, Fabian Angelstorf, et al. 2015. OMEN: Faster Password Guessing Using an Ordered Markov Enumerator. In *Proc. ESSoS 2015.* 119–132.
[29] David Freeman, Sakshi Jain, Markus Dürmuth, et al. 2016. Who are you? A statistical approach to measuring user authenticity. In *Proc. NDSS 2016.* 1–15.
[30] Briland Hitaj, Paolo Gasti, Giuseppe Ateniese, et al. 2019. Passgan: A deep learning approach for password guessing. In *Proc. ACNS 2019.* 217–237.
[31] Shiva Houshmand, Sudhir Aggarwal, and Randy Flood. 2015. Next gen PCFG password cracking. *IEEE Trans. Inf. Forensics Secur.* 10, 8 (2015), 1776–1791.
[32] Radek Hranický, Filip Lištiak, Dávid Mikuš, et al. 2019. On practical aspects of pcfg password cracking. In *Proc. DBSec 2019.* 43–60.
[33] Radek Hranický, Lukáš Zobal, Ondřej Ryšavý, et al. 2020. Distributed pcfg password cracking. In *Proc. ESORICS 2020.* 701–719.
[34] Shunbin Li, Zhiyu Wang, Ruyun Zhang, et al. 2022. Mangling Rules Generation with Density-based Clustering for Password Guessing. *IEEE Trans. Dependable Secur. Comput.* 20, 5 (2022), 3588–3600.
[35] Enze Liu, Amanda Nakanishi, Maximilian Golla, et al. 2019. Reasoning analytically about password-cracking software. In *Proc. IEEE S&P 2019.* 380–397.
[36] Jerry Ma, Weining Yang, Min Luo, et al. 2014. A Study of Probabilistic Password Models. In *Proc. IEEE S&P 2014.* 689–704.
[37] William Melicher, Blase Ur, et al. 2016. Fast, lean and accurate: Modeling password guessability using neural networks. In *Proc. USENIX SEC 2016.* 175–191.
[38] Junko Nakajima and Mitsuru Matsui. 2002. Performance analysis and parallel implementation of dedicated hash functions. In *Proc. EUROCRYPT 2002.* 165–180.
[39] Arvind Narayanan and Vitaly Shmatikov. 2005. Fast dictionary attacks on passwords using time-space tradeoff. In *Proc. ACM CCS 2005.* 364–372.
[40] Christoforos Ntantogian, Stefanos Malliaros, et al. 2019. Evaluation of password hashing schemes in open source web platforms. *Comput. Secur.* 84 (2019), 206–224.
[41] John D Owens, Mike Houston, David Luebke, et al. 2008. GPU computing. *Proc. IEEE* 96, 5 (2008), 879–899.
[42] Bijeeta Pal, Tal Daniel, et al. 2019. Beyond credential stuffing: Password similarity models using neural networks. In *Proc. IEEE S&P 2019.* 417–434.
[43] Dario Pasquini, Marco Cianfriglia, Giuseppe Ateniese, et al. 2021. Reducing Bias in Modeling Real-world Password Strength via Deep Learning and Dynamic Dictionaries. In *Proc. USENIX SEC 2021.* 821–838.
[44] Dario Pasquini, Ankit Gangwal, Giuseppe Ateniese, et al. 2021. Improving password guessing via representation learning. In *Proc. IEEE S&P 2021.* 1382–1399.
[45] Rico Sennrich, Barry Haddow, and Alexandra Birch. 2016. Neural machine translation of rare words with subword units. In *Proc. ACL 2016.* 1715–1725.
[46] Marc Stevens, Elie Bursztein, Pierre Karpman, et al. 2017. The first collision for full SHA-1. In *Proc. CRYPTO 2017.* 570–596.
[47] Rafael Veras, Christopher Collins, and Julie Thorpe. 2014. On semantic patterns of passwords and their security impact. In *Proc. NDSS 2014.*
[48] Ding Wang, Haibo Cheng, Ping Wang, et al. 2017. Zipf's Law in Passwords. *IEEE Trans. Inf. Forensics Secur.* 12, 11 (2017), 2776–2791.
[49] Ding Wang, Ping Wang, Debiao He, et al. 2019. Birthday, name and bifacial-security: understanding passwords of Chinese web users. In *Proc. USENIX SEC 2019.* 1537–1555.
[50] Ding Wang, Zijian Zhang, Ping Wang, et al. 2016. Targeted online password guessing: An underestimated threat. In *Proc. ACM CCS 2016.* 1242–1254.
[51] Ding Wang, Yunkai Zou, Yuan-An Xiao, et al. 2023. Pass2Edit: A Multi-Step Generative Model for Guessing Edited Passwords. In *Proc. USENIX SEC 2023.* 983–1000.
[52] Ding Wang, Yunkai Zou, Zijian Zhang, et al. 2023. Password guessing using random forest. In *Proc. USENIX SEC 2023.* 965–982.
[53] Xiaoyun Wang and Hongbo Yu. 2005. How to break MD5 and other hash functions. In *Proc. EUROCRYPT 2005.* 19–35.
[54] Charles Matthew Weir. 2010. *Using Probabilistic Techniques to Aid in Password Cracking Attacks.* Ph. D. Dissertation. Florida State University.
[55] Matt Weir, Sudhir Aggarwal, Breno De Medeiros, et al. 2009. Password cracking using probabilistic context-free grammars. In *Proc. IEEE S&P 2009.* 391–405.
[56] Yuxuan Wu, Ding Wang, Yunkai Zou, et al. 2022. Improving Deep Learning Based Password Guessing Models Using Pre-processing. In *Proc. ICICS 2022.* 163–183.
[57] Zhiyang Xia, Ping Yi, Yunyu Liu, et al. 2019. GENPass: A multi-source deep learning model for password guessing. *IEEE Trans. Multim.* 22, 5 (2019), 1323–1332.
[58] Ming Xu, Chuanwang Wang, Jitao Yu, et al. 2021. Chunk-level password guessing: Towards modeling refined password composition representations. In *Proc. ACM CCS 2021.* 5–20.
[59] Ming Xu, Jitao Yu, Xinyi Zhang, et al. 2023. Improving real-world password guessing attacks via bi-directional transformers. In *Proc. USENIX SEC 2023.* 1001–1018.
[60] Verena Zimmermann. 2021. *From the Quest to Replace Passwords towards Supporting Secure and Usable Password Creation.* Ph. D. Dissertation. Technical University of Darmstadt.
[61] Verena Zimmermann and Nina Gerber. 2020. The password is dead, long live the password–A laboratory study on user perceptions of authentication schemes. *Int. J. Hum. Comput. Stud.* 133 (2020), 26–44.

APPENDIX

A Implementation of PCFG model

We now provide a detailed explanation of the password generation process of the PCFG model [55]. For each base structure, replacing each tag with its most probable substring forms the most likely password for that base structure. When initializing the priority queue, the password with the highest probability generated by each base structure is pushed into the priority queue. In each iteration, the highest-probability password is popped from the priority queue. This password is then modified by replacing one of its substrings with another substring of lower probability under the same tag, thus deriving a new, less probable password from the original. This procedure, when applied to multiple substrings, yields several distinct new passwords that are pushed back into the priority queue.

Next function algorithm [55] ensures that no duplicates or omissions occur when deriving new passwords. It assigns a variable named *pivot* to each password within the priority queue. For a password comprising l tags in its base structure, sequentially from 0 to $l-1$, once the password is popped from the priority queue, replacing the substrings corresponding to tags from *pivot* to $l-1$ with less probable alternatives generates $l - pivot$ less probable passwords. When the most probable password of each base structure is pushed into the priority queue, its *pivot* value is set to 0. Later, for a password that is derived through the replacement of the substring corresponding to tag i, its *pivot* value is set to i.

We use a simple example to demonstrate the process of generating guesses using the Next function algorithm. Suppose in the grammar of the PCFG model there is only one base structure, L_4D_3. L_4 can derive three substrings "`pass`", "`love`", and "`life`", while D_3 can derive "`123`", "`101`", and "`007`", with all substring probabilities ordered in descending sequence. The highest-probability password of the base structure L_4D_3 is `pass123` ($pivot = 0$). It can derive `love123` ($pivot = 0$) by replacing "`pass`" with "`love`" and `pass101` ($pivot = 1$) by replacing "`123`" with "`101`". Subsequently, `love123` can further derive `life123` and `love101`. However, given that its $pivot$ value is 1, `pass101` is limited to deriving `pass007` but not `love101`, which is exclusively derivable from `love123`. This example can be easily extended to the case where the PCFG model's grammar contains multiple base structures, as the process of generating passwords using each base structure is independent.

B Introduction of Markov n-gram model

The core assumption of the Markov n-gram model is that each character in a string is related only to the preceding $n-1$ consecutive characters, and the probability of the string equals the product of each character's conditional probability. The probability of the string $s = c_1c_2\cdots c_m$ can be calculated as:

$$\Pr(s) = \Pr(c_1)\Pr(c_2|c_1)\cdots\Pr(c_m|c_{m-n+1}\cdots c_{m-1})$$

where $\Pr(c_i|c_{i-n+1}\cdots c_{i-1})$ represents the conditional probability of character c_i appearing after the string $c_{i-n+1}\cdots c_{i-1}$, which can be obtained from a training dataset. Assuming the character set used in the password dataset for training is Σ, the frequency with which the string $c_{i-n+1}\cdots c_{i-1}c_i$ appears is given by $\texttt{Count}(c_{i-n+1}\cdots c_{i-1}c_i)$, and the count of occurrences of the string $c_{i-n+1}\cdots c_{i-1}$ followed by any character from the character set is denoted as $\texttt{Count}(c_{i-n+1}\cdots c_{i-1}\cdot)$. Under the Laplace smoothing method, every occurrence of the string $c_{i-n+1}\cdots c_{i-1}c_i$ is incremented by a δ value (recommended 0.01 in [36]). The probability value for $\Pr(c_i|c_{i-n+1}\cdots c_{i-1})$ is therefore determined as:

$$\Pr(c_i|c_{i-n+1}\cdots c_{i-1}) = \frac{\texttt{Count}(c_{i-n+1}\cdots c_{i-1}c_i) + \delta}{\texttt{Count}(c_{i-n+1}\cdots c_{i-1}\cdot) + |\Sigma|\cdot\delta}$$

To ensure that the sum of probabilities for all passwords equals 1, a normalization method is required [36]. One normalization method is distribution-based normalization. It calculates the relative frequency for passwords of each length in the training set and uses it as the probability. The probability of a password is the product of its length's probability and the conditional probabilities of all its characters. Another normalization method is end-symbol normalization. This approach appends an end-symbol c_e to each password and includes it in the character set. The probability of the password $c_1c_2\cdots c_mc_e$ is calculated as the probability of the string $c_1c_2\cdots c_m$ multiplied by the conditional probability of c_e succeeding it.

C Guessing methods/models setups

Rule & Mask. The Rule and Mask modes in hashcat [18] are used, outputting guesses to files through the `-stdout` and `-outfile` options. In Rule mode, we combine the Rockyou dataset as the dictionary with generated.rule, a built-in rule set in hashcat [18] that includes 14,350 rules. For Mask mode, we enumerate all 8-character passwords comprised of printable ASCII characters.
PCFG. For the PCFG model, we utilize the improved open-source pcfg_cracker model [23] developed by the original creators. For its two variants, compiled_pcfg and pcfg-manager, we also employ source code from the original developers [5, 8]. Due to these two models only implementing the generation component, they require the training module of the pcfg_cracker model [23] to generate the PCFG grammar. The pcfg-manager model [8], which supports parallel guess generation, is configured to use 32 CPU threads.
Markov & OMEN. We implement the Markov 4-gram model and adopt the Laplace smoothing and End-symbol normalization methods as described in [36]. For its variant, the OMEN model [28], we use the source code [6], with the default configuration, where the gram is set to 4 and the maximum level is set to 10.
FLA, PassGAN and AdaMs. The three models are implemented using the open-source codes [2, 3, 7] and their default configurations. More specifically, the FLA model employs three LSTM layers and two fully connected layers (termed the "small" model in [37]), and is trained for a total of 20 epochs. The PassGAN model is trained for 200,000 iterations and it generates password guesses in batches of size 1,024. For the AdaMs model, we utilize the neural network trained on the *generated* rule-set in [43] and employ the ClixSense password dataset as the target set.
RFGuess. We implement a 6-gram Random Forest model, with parameters consistent with [52]. More specifically, its number of decision trees is set to 30, minimum number of nodes is set to 10, and the maximum ratio of features is set to 80%.

D Hashes matching algorithm in hashcat

After generating a password and computing its hash (a hash includes a digest and a salt), a search for a match in the target hash set is required. If matching each hash necessitates traversing the entire target set, it would result in a significant waste of time. Hashcat [18] has developed an algorithm to manage the target set, aimed at reducing the time complexity associated with a single match.

In detail, hashcat [18] sorts all the target hashes by their salts (considering unsalted hashes have empty string salts). If two hashes share the same salt, they are further sorted by their digests. Afterwards, hashcat [18] employs a Bloom filter to store all digests, which is a space-efficient data structure designed for quick querying an element's membership in a set, with a potential for false positives. Upon computing the hash for each password-salt pair, hashcat [18] first queries the digest in the Bloom filter. If the digest does not exist within the Bloom filter, it indicates that the hash cannot match. Conversely, if the digest is found, verification is then required using the binary search method in the sorted list of hashes.

Therefore, for a password-salt pair, after hashcat [18] computes its hash, it only requires constant time to query in the Bloom filter. If the digest is found within the Bloom filter, an additional logarithmic time is needed for verification. Since the usual number of guesses substantially surpasses the volume of the target set especially under fast hash scenarios, the chances of locating the digest of a candidate password in the Bloom filter are minimal. Consequently, the time complexity of hash matching can largely be considered constant in theory. However, in practice, due to the high success rate when

Algorithm 1: Process of deriving child PTs.

```
Input: Priority queue $PQ$, Threshold $\mathcal{T}$, Buffer $\mathcal{B}$
$PQ.init()$ /* At the time of PQ initialization, the pivot value of the PTs pushed within it is set to 0. */
while $!PQ.empty()$ do
    $current\_PT$=$PQ.pop()$
    $\mathcal{B}.add(current\_PT)$
    $l$=$current\_PT.child\_number()$
    for $i = current\_PT.pivot$ to $l$-1 do
        $child\_PT$=$current\_PT.find\_child(i)$
        if $child\_PT$!=$NULL$ and $child\_PT.prob$>=$\mathcal{T}$ then
            $child\_PT.pivot$=$i$
            $PQ.push(child\_PT)$
```

guessing popular passwords, the number of binary searches conducted could positively correlate with the size of the target set. Thus, when there are more distinct hashes in the target set, the average time spent on a single match might increase.

E Process of deriving child pre-terminals

When deriving child pre-terminals (PTs), we employed the Next function algorithm, just as in the PCFG model, to prevent duplication and omission[55]. Algorithm 1 briefly illustrates the process of deriving PTs. Each PT is assigned a *pivot* variable. When initializing the priority queue PQ, the *pivot* values of the initially pushed PTs are set to 0. In the generation loop, if a PT can derive l different child PTs, numbered from 0 to $l-1$, then only child PTs numbered from *pivot* to $l-1$ are pushed back into the priority queue. This process can generate all PTs with probabilities greater than a specific threshold $\mathcal{T}$ and ends after generating all possible PTs.

Specifically, for PCFG, a child PT can be derived from a parent PT by simply having one tag in the base structure derive substrings of a lower probability level. This is equivalent to replacing the passwords used in the algorithm of original PCFG with PTs. For OMEN, the number of child PTs is equal to the length of the parent PT. The *level* value of the i-th character of the i-th child PT is one higher than that of the parent PT, while the *level* values of all other characters remain equal to those of the parent PT.

Note that although Weir [54] introduced the *Deadbeat dad* algorithm as an alternative to the Next function algorithm to mitigate the issue of excessive memory consumption by the priority queue, we still opt for the Next function algorithm here. One reason is that, as discussed in Section 4.3, each PT is designed to generate a large number of passwords, resulting in a generally low number of PTs in the priority queue. This prevents the excessive memory usage issue associated with the original PCFG model [55]. Another reason is that the Deadbeat dad algorithm is more complex when deriving each child PT, requiring more time. Given our framework's high demand for speed, trading time for space is unacceptable.

F Probability reassignment algorithm

To address the issue in the design of PCFG PTs where a tag may not be able to derive a sufficient number of substrings with the same probability, we propose the probability reassignment algorithm, which manually adjusts the probability of each substring. As shown in Algorithm 2, for each tag, we first arrange all possible substrings it can derive in descending order of probability, followed by multiple rounds of probability reassignment. In the first round, the probabilities of the top x_1 substrings are reassigned to their average value, and for the i-th round, the next x_i substrings' probabilities are reassigned to their average value, with $x_{i+1} = 2x_i$, continuing until all substrings derived from the tag have been reassigned.

Algorithm 2: Probability reassignment algorithm.

```
Input: PCFG Grammar $\mathcal{G}$, Tag $t$, hyperparameter $x_1$
$substrings = \mathcal{G}.get\_substrings\_list(t)$
$substrings.sort()$ /* Sort substrings in descending order. */
$round = 0$
$tail_{round} = 0$
while $tail_{round} < substrings.length$ do
    $round = round + 1$
    if $round > 1$ then
        $x_{round} = 2 \times x_{round-1}$
    $head_{round} = tail_{round-1}$
    $tail_{round} = MIN(substrings.length, head_{round} + x_{round})$
    $substrings.reassign(head_{round}, tail_{round})$
```

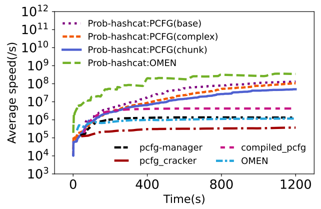
We employ a simple example to illustrate the algorithm for reassigning probabilities to substrings. Suppose the tag L_6 can derive 10,000 substrings, arranged in descending order of probability. If the value of x_1 for the first round is 2,048, then the probabilities of the first 2,048 substrings are reassigned to their average value. For the second round, the probabilities of the next 4,096 substrings (from the 2049th to the 6144th) are similarly reassigned. The probabilities of the remaining substrings (from the 6145th to the 10000th) are reassigned in the final round. Thus, for a base structure containing only one tag, L_6, it can produce three PTs capable of generating 2,048, 4,096, and 3,856 password guesses, respectively.

G Supplementary experiment results

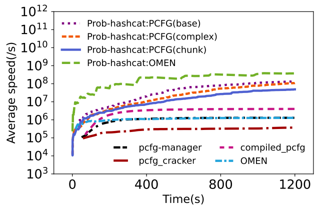
In this section, we present the results of experiments across the total eight attack scenarios (Dodonew/CSDN→Tianya/Taobao and RockYou/LinkedIn→000Webhost/ClixSense). The first experiment involves eight different attack processes, four from Prob-hashcat (base PCFG, complex PCFG, chunk PCFG, and OMEN) and four dynamic guessing processes using pcfg_cracker [23], compiled_pcfg [5], pcfg-manager [8], and the original OMEN model [6, 28]. Figures 10 shows the comprehensive results for speeds experiments under MD5 hash across the total eight scenarios. Tables 5 and 6 detail the speeds and cracking rates of these experiments at 20 minutes, respectively. To show the consistency of experimental results across different fast hashes, we provide the results for speed and cracking rate experiments under SHA2-256 in Figure 12.

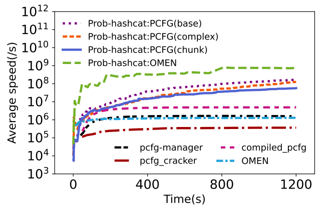
In the second experiment, we compare the four attack processes in Prob-hashcat with Rule attacks in hashcat [18] to demonstrate the application of Prob-hashcat. The Rule attacks use two rule sets: generated.rule and generated2.rule. While generated.rule is of higher quality, generated2.rule contains more rules. We hash the target sets using salted MD5, with the number of different salts set to 100. Table 7 and Table 8 show the speeds and the cracking rates of each attack process within one hour in each scenario.

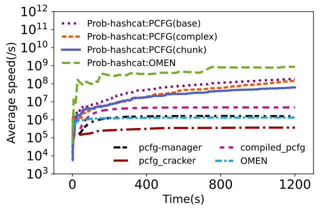
Impact of dataset size on speed. Table 5 shows that the speed of the four dynamic guessing processes is primarily related to the training set and independent of the target set. This is because the training set determines the speed of the guess generation Phase G, while the target set only affects the hash computation and matching

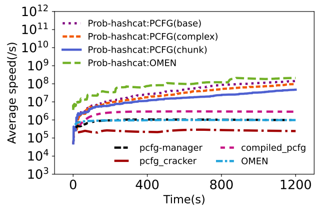


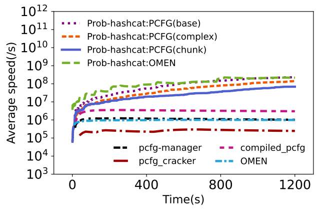
(a) Dodonew→Tianya

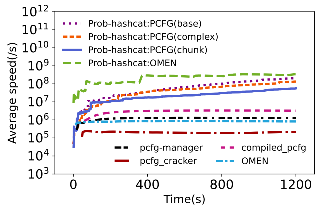

(b) Dodonew→Taobao


(c) CSDN→Tianya


(d) CSDN→Taobao


(e) RockYou→000Webhost


(f) RockYou→ClixSense


(g) LinkedIn→000Webhost

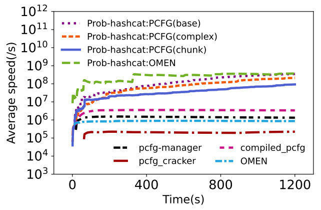

(h) LinkedIn→ClixSense

Figure 10: Comparison of the speeds of attack processes in Prob-hashcat against those of dynamic guessing attack processes using pcfg_cracker [23], compiled_pcfg [5], pcfg-manager [8], and the original OMEN model [6, 28] under MD5 across eight scenarios. Prob-hashcat's attack processes consistently outperform the compared dynamic guessing processes in all scenarios.

Table 5: Comparison of the average speed of Prob-hashcat and dynamic guessing attacks within 20 minutes.*

Scenario	Prob-hashcat attacks				Dynamic guessing attacks			
	PCFG (base)	PCFG (complex)	PCFG (chunk)	OMEN	pcfg_ cracker [23]	compiled_ pcfg [5]	pcfg- manager [8]	OMEN [6, 28]
Dodonew→Tianya	1.3×10^8	1.1×10^8	4.9×10^7	$\mathbf{3.5 \times 10^8}$	3.7×10^5	$\mathbf{4.2 \times 10^6}$	1.3×10^6	1.2×10^6
Dodonew→Taobao	1.4×10^8	1.1×10^8	4.9×10^7	$\mathbf{3.8 \times 10^8}$	3.7×10^5	$\mathbf{4.0 \times 10^6}$	1.3×10^6	1.3×10^6
CSDN→Tianya	1.7×10^8	1.3×10^8	5.7×10^7	$\mathbf{7.7 \times 10^8}$	3.7×10^5	$\mathbf{4.9 \times 10^6}$	1.6×10^6	1.3×10^6
CSDN→Taobao	1.9×10^8	1.5×10^8	6.3×10^7	$\mathbf{8.6 \times 10^8}$	3.7×10^5	$\mathbf{4.9 \times 10^6}$	1.6×10^6	1.3×10^6
RockYou→000Webhost	1.4×10^8	9.6×10^7	4.7×10^7	$\mathbf{2.1 \times 10^8}$	2.4×10^5	$\mathbf{2.9 \times 10^6}$	9.9×10^5	9.9×10^5
RockYou→ClixSense	$\mathbf{2.3 \times 10^8}$	1.4×10^8	6.9×10^7	2.2×10^8	2.5×10^5	$\mathbf{3.0 \times 10^6}$	1.0×10^6	9.9×10^5
LinkedIn→000Webhost	2.0×10^8	1.3×10^8	5.7×10^7	$\mathbf{3.4 \times 10^8}$	2.1×10^5	$\mathbf{3.3 \times 10^6}$	1.3×10^6	8.2×10^5
LinkedIn→ClixSense	3.5×10^8	2.1×10^8	9.3×10^7	$\mathbf{3.7 \times 10^8}$	2.2×10^5	$\mathbf{3.4 \times 10^6}$	1.3×10^6	8.7×10^5

*All numerical values in this table represent the average number of guesses attempted per second by each attack process under MD5. In each row, two **bold** values indicate the highest speeds achieved by Prob-hashcat attacks and dynamic guessing attacks, respectively.

Table 6: Comparison of the cracking rate of Prob-hashcat and dynamic guessing attacks within 20 minutes.*

Scenario	Prob-hashcat attacks				Dynamic guessing attacks			
	PCFG (base)	PCFG (complex)	PCFG (chunk)	OMEN	pcfg_ cracker [23]	compiled_ pcfg [5]	pcfg- manager [8]	OMEN [6, 28]
Dodonew→Tianya	**79.74%**	78.48%	78.17%	70.64%	**62.93%**	57.06%	54.62%	61.06%
Dodonew→Taobao	**66.76%**	64.69%	66.76%	52.50%	48.01%	**53.32%**	48.33%	37.09%
CSDN→Tianya	**75.54%**	75.52%	75.15%	71.39%	51.33%	**53.11%**	50.93%	39.17%
CSDN→Taobao	55.03%	55.67%	**55.89%**	50.72%	33.39%	**39.76%**	34.86%	24.62%
RockYou→000Webhost	**39.83%**	36.50%	36.96%	23.06%	26.79%	**35.55%**	33.41%	14.04%
RockYou→ClixSense	68.43%	**69.52%**	65.86%	51.70%	51.10%	**57.24%**	53.30%	35.78%
LinkedIn→000Webhost	**43.48%**	39.27%	40.46%	25.66%	28.39%	**38.13%**	36.27%	9.08%
LinkedIn→ClixSense	72.99%	**73.57%**	69.85%	55.17%	52.72%	**61.47%**	58.42%	28.65%

*All numerical values in this table represent the fraction of passwords successfully cracked by each attack process under MD5. In each row, two **bold** values indicate the highest fractions of passwords successfully cracked by Prob-hashcat attacks and dynamic guessing attacks, respectively.

Phase H. Since the speed of Phase H is significantly higher than that of Phase G, and the speed of dynamic guessing processes is dictated by the slower sub-Phase, the target set has little impact. This further validates the theories discussed in Section 3.3.

Conversely, the size of the target set affects the speed of the four attack processes in Prob-hashcat. This impact is particularly evident in the four scenarios using English datasets, as there is a significant difference in the number of unique passwords between 000Webhost and ClixSense (the former has about ten times as many as the latter, see Table 4). Since a larger target set typically leads to more time spent on hash matching in practice (see Appendix D), the cracking speed of the process with 000Webhost as the target set is significantly lower than that with ClixSense as the target set when using the same training set, as shown in Table 5.

Table 7: Comparison of the average speed of Prob-hashcat and Rule attacks within one hour.*

Scenario	Prob-hashcat attacks				Rule attacks in hashcat [18]	
	PCFG (base)	PCFG (complex)	PCFG (chunk)	OMEN	generated	generated2
Dodonew→Tianya	9.5×10^6	9.4×10^6	4.1×10^6	$\mathbf{3.9 \times 10^7}$	1.8×10^7	$\mathbf{1.8 \times 10^7}$
Dodonew→Taobao	9.1×10^6	8.3×10^6	4.1×10^6	$\mathbf{3.6 \times 10^7}$	$\mathbf{1.7 \times 10^7}$	1.6×10^7
CSDN→Tianya	4.2×10^6	5.2×10^6	1.0×10^6	$\mathbf{2.9 \times 10^7}$	$\mathbf{1.8 \times 10^7}$	1.5×10^7
CSDN→Taobao	3.7×10^6	4.0×10^6	1.0×10^6	$\mathbf{2.7 \times 10^7}$	$\mathbf{1.6 \times 10^7}$	1.5×10^7
RockYou→000Webhost	1.9×10^6	7.0×10^6	2.0×10^6	$\mathbf{3.2 \times 10^7}$	$\mathbf{1.9 \times 10^7}$	1.8×10^7
RockYou→ClixSense	2.1×10^6	1.0×10^7	2.1×10^6	$\mathbf{3.8 \times 10^7}$	$\mathbf{2.4 \times 10^7}$	2.3×10^7
LinkedIn→000Webhost	3.3×10^6	1.1×10^7	2.0×10^6	$\mathbf{1.5 \times 10^7}$	$\mathbf{2.0 \times 10^7}$	1.8×10^7
LinkedIn→ClixSense	4.7×10^6	1.5×10^7	2.3×10^6	$\mathbf{1.8 \times 10^7}$	$\mathbf{2.4 \times 10^7}$	2.3×10^7

*All numerical values in this table represent the average number of guesses attempted per second by each attack process under salted MD5. In each row, two **bold** values indicate the highest speeds achieved by Prob-hashcat attacks and Rule attacks, respectively.

Table 8: Comparison of the cracking rate of Prob-hashcat and Rule attacks within one hour.*

Scenario	Prob-hashcat attacks				Rule attacks in hashcat [18]	
	PCFG (base)	PCFG (complex)	PCFG (chunk)	OMEN	generated	generated2
Dodonew→Tianya	**73.03%**	72.61%	69.87%	65.41%	**71.88%**	70.97%
Dodonew→Taobao	**58.14%**	56.47%	51.68%	41.96%	**56.12%**	53.41%
CSDN→Tianya	**72.21%**	72.19%	70.95%	62.22%	**73.83%**	71.15%
CSDN→Taobao	**50.54%**	49.81%	49.73%	40.13%	**52.36%**	48.95%
RockYou→000Webhost	31.68%	**32.78%**	29.81%	16.92%	**32.32%**	31.50%
RockYou→ClixSense	58.97%	**65.61%**	56.97%	44.28%	**62.63%**	61.10%
LinkedIn→000Webhost	**32.03%**	31.93%	31.96%	17.25%	**34.50%**	33.97%
LinkedIn→ClixSense	56.33%	**66.13%**	57.58%	46.50%	**64.92%**	63.38%

*All numerical values in this table represent the fraction of passwords successfully cracked by each attack process under salted MD5. In each row, two **bold** values indicate the highest fractions of passwords successfully cracked by Prob-hashcat attacks and Rule attacks, respectively.

Impact of language on cracking rates. We conduct the cracking rate experiments in eight attack scenarios. The first four scenarios utilize Chinese datasets, representing cross-site attacks in Chinese environments, while the latter four use English datasets, representing attacks in English environments. Some attack processes perform differently under various languages, requiring attackers to select the appropriate password cracking process based on the language environment of the actual cracking target.

From Tables 6 and 8, it can be observed that the performance of the OMEN model is better in Chinese environments compared to English environments. In Chinese environments, the OMEN attack in Prob-hashcat surpasses the dynamic guessing attack processes using PCFG. However, in English environments, its cracking rate is still lower than that of the PCFG attacks. This is because, compared to Chinese users who prefer digit-based patterns, English users tend to use patterns related to letters [49], resulting in the OMEN model learning a higher probability for letters. Given that the number of letters in the character set significantly exceeds that of digits, the OMEN model may generate more letter-dominated passwords that do not hit when guessing, leading to its underperformance in English environments compared to the Chinese environments.

Based on our experimental results, the base PCFG attack may be more suitable for attacks in Chinese environments. In contrast, the complex PCFG attack is more suited in English environments. Table 6 shows that under MD5, in Chinese environments, the base PCFG attack has an advantage in three out of four scenarios; in English environments, the base PCFG attack and the complex PCFG attack each dominate in two scenarios. Table 8 illustrates that under salted MD5, in Chinese environments, the base PCFG attack holds an advantage in all four scenarios; while in English environments, the complex PCFG attack leads in three out of four scenarios. Therefore,

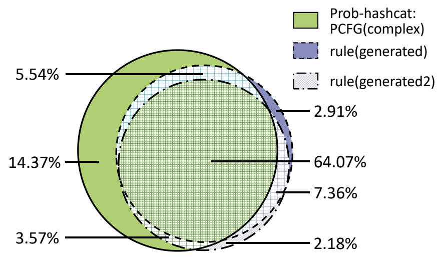
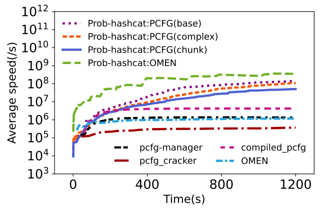


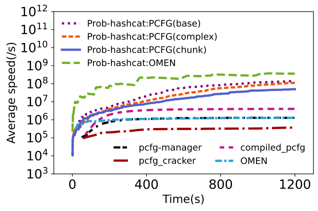
Figure 11: The overlap ratios of passwords cracked by complex PCFG attack in the Prob-hashcat and two Rule attacks.

attackers may opt for the base PCFG attack in Chinese environments and the complex PCFG attack in English environments. If feasible, they could also customize new training models through the PCFG training interface (see Section 5.4 for details).

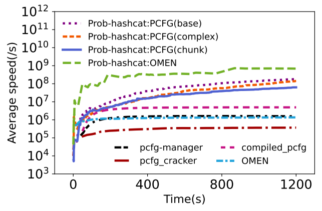
Overlaps. A practical application of Prob-hashcat is its complementary relationship with Rule attacks. Due to the different architectures of probabilistic models and Rule attacks, the passwords they attempt may vary. As shown in Figure 11, in the Rockyou→ClixSense scenario, the number of passwords cracked by the complex PCFG attack is similar to those cracked by the Rule attacks using generated.rule and generated2.rule. However, 14.37% of the passwords cracked by the complex PCFG attack are not hit by any Rule attacks, while the non-overlapping portion between the two Rule attacks is only approximately 7%. Therefore, since both Prob-hashcat and Rule attacks are fast, attackers can opt to run multiple guessing attack processes for limited durations each, which may be more effective than running a single attack process for an extended period. Moreover, when selecting a different attack process, using Prob-hashcat may also be more effective than switching rule sets.



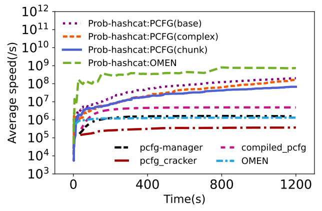
(a) Dodonew⟶Tianya



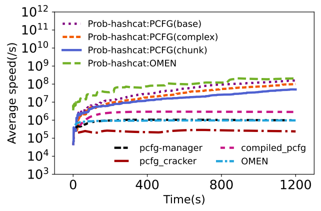
(b) Dodonew⟶Taobao



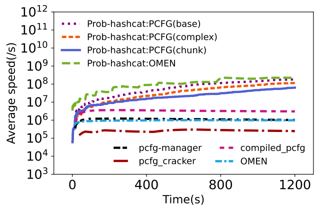
(c) CSDN⟶Tianya



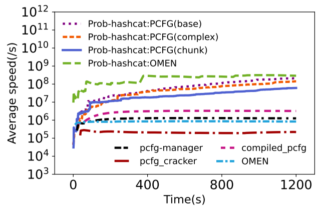
(d) CSDN⟶Taobao



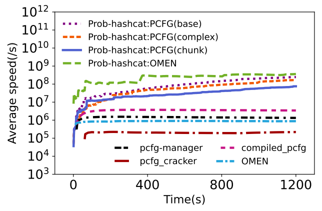
(e) RockYou⟶000Webhost



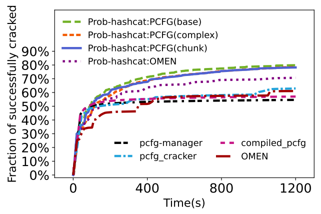
(f) RockYou⟶ClixSense



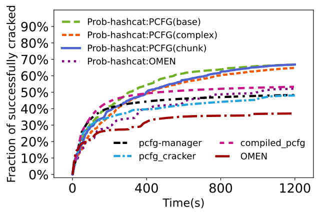
(g) LinkedIn⟶000Webhost



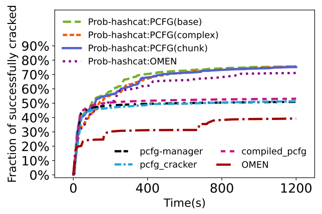
(h) LinkedIn⟶ClixSense



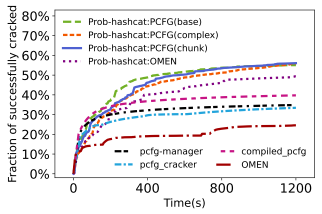
(i) Dodonew⟶Tianya



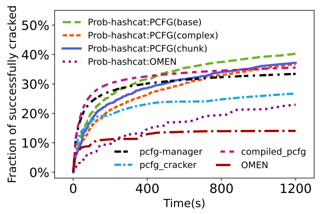
(j) Dodonew⟶Taobao



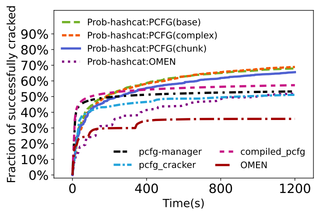
(k) CSDN⟶Tianya



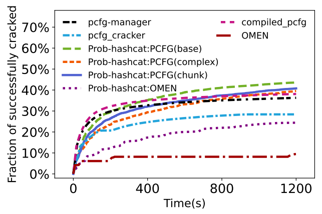
(l) CSDN⟶Taobao



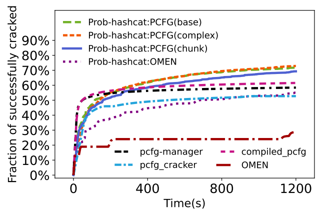
(m) RockYou⟶000Webhost



(n) RockYou⟶ClixSense



(o) LinkedIn⟶000Webhost



(p) LinkedIn⟶ClixSense

Figure 12: Average speeds and fractions of passwords successfully cracked by four attack processes in Prob-hashcat compared with dynamic guessing attack processes using pcfg_cracker [23], compiled_pcfg [5], pcfg-manager [8], and the original OMEN model [6, 28] under SHA2-256 hash across all eight scenarios. Sub-figures (a) to (h) show the speed experiment results, and sub-figures (i) to (p) display the cracking rate experiment results. The experimental outcomes under SHA2-256 hash closely mirror those under MD5 hash, as depicted in Figures 7 and 10. The differences in cracking rates between experiments conducted under MD5 and SHA2-256 for each scenario and each model are less than 1%, and in most cases, less than 0.1%.