Using Parallel Techniques to Accelerate PCFG-based Password Cracking Attacks

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

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

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

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

1 Introduction

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

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

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

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

*corresponding author

1

Using Parallel Techniques to Accelerate PCFG-based Password Cracking Attacks

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

摘要 — 文本密码在访问控制机制中发挥着重要作用，通常以密文形式存储在服务器上。然而，攻击者可能会尝试对大量候选密码进行哈希运算，以找到密码数据库中目标哈希值的匹配项。为了破解密码数据库，行业中的攻击者通常使用像 Hashcat 这样的破解软件。最近，学术研究人员提出了许多数据驱动的概率模型，其中概率上下文无关文法（PCFG，简称）尤为突出。尽管破解效率很高，但由于密码候选生成速度显著缓慢，数据驱动模型很少被工业实践采用。

为弥合差距并推动高效的数据驱动模型在工业中的实际应用，我们提出使用并行技术加速候选密码生成，从而实现 PCFG 模型与实际使用的 Hashcat 工具的集成。为此，我们主要提出了两种加速 PCFG 模型密码生成的算法：首先，我们设计了一种具有内存负载均衡策略的存储结构，以更均衡地存储用于生成密码的数据结构；其次，我们设计了一种通过不同线程并行生成候选密码的算法。基于这两种算法，我们提出了 Parallel_PCFG，并在 Hashcat 上基于其内置 GPU 内核实现了 Parallel_PCFG。我们对 Parallel_PCFG 与最先进的数据驱动模型进行了全面评估，发现 Parallel_PCFG 与性能最佳模型相比，仅耗时 14.33% 即可达到相同的破解率，为 PCFG 模型与 Hashcat 的集成铺平了道路。

Index Terms—密码安全，基于 PCFG 的模型，候选密码生成，GPU 并行性

1 引言

Textual 密码由于易用性和低成本的优势，在我们的日常生活中仍然被广泛用于保护账户和私人数据，并且通常以密文形式存储以防止恶意攻击。人类创建的密码通常不足以抵抗密码破解攻击。黑客通过尝试大量候选密码来哈希，以找到与目标密文密码数据库匹配的明文密码，这被称为通用密码破解攻击。

最近，研究人员提出了许多数据驱动的概率猜测模型 [9], [34],, 例如，概率上下文无关语法（ PCFG，简称） [38], ，他们通常根据训练集学习一个有学问的密码空间，以按降序概率 [16], [27] 生成密码候选。虽然 PCFG 成为许多数据驱动模型 [15], [38]，的有力竞争者，但基于 PCFG 的模型的生成方法显著受到生成速度慢的影响。在现实世界的工业案例中，黑客很少使用数据驱动模型，原因如下：(1) 数据驱动模型的生成速度很慢，例如，通常需要一天或更多天才能生成大约 10^14 个候选密码 [35], [38],，增加了时间成本。(2) 针对密文密码时，数据驱动模型应利用破解软件（如 Hashcat ）中的哈希函数来加密生成的明文密码候选。现有的工业实践通常使用管道模式，将每个生成的候选密码串行传输到破解软件，其效率受候选生成速度慢的显著限制。(3) 数据驱动模型通常根据猜测数量相等来评估破解率 [21], [38], [39], ，对猜测实际随时间如何表现提供有限见解（例如，在半小时内）。某些场景可能对时间非常敏感，例如，当多个攻击者同时针对同一系统时，第一个攻破系统的攻击者可能会窃取资金或窃取有价值的信息。

为了弥补数据驱动模型在工业猜测攻击中应用上的差距，丰富的文献 [6], [9],[12] 已经提出了加速数据驱动概率模型密码生成的有效技术。首先，它们优化用于加速密码生成的模型结构。典型的例子是 OMEN [9] ，它将候选密码的概率分为十个等级，以同时生成同一等级的密码，即以粗粒度的降序概率生成候选密码。其次，它们利用分布式计算机集群，即更多的计算机资源，来生成候选密码。典型的例子是 PCFG-manager 方法 [6], [12] ，它将密码生成任务分配给多台机器，并采用周密的通信协议。尽管如此，借助并行技术，它仍有显著潜力来加速生成速度，支持数据驱动模型在工业上的应用。

随着硬件的发展，GPU 可以成为加速密码生成任务的一种有前景的方式，因为其多核架构可以提供更高的计算能力

* 通讯作者

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

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

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

We summarize the main contributions as follows:

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

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

2 Background Knowledge

2.1 Password Cracking Attacks

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

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

2.2 Related Works

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

通过充分利用并行性。在本文中，我们致力于利用基于 GPU 的并行技术来提高密码生成速度。为此，挑战如下：1) 数据驱动生成方法的数据存储结构必须精心设计，以与 GPU 的并行生成算法相匹配。由于 GPU 运行时间通常受最慢线程的限制，不平衡的存储结构会严重阻碍性能。优化这些结构对于减少瓶颈和最大化效率至关重要。2) 确保输出候选密码按降序排列是另一个挑战。这一目标涉及将生成任务分配给不同的线程，并设计查找表以定位每个线程的候选密码，这变得具有挑战性。

为了应对上述挑战，我们提出使用基于 GPU 的并行技术来加速基于 PCFG 的密码破解攻击，称为 Parallel_PCFG。Parallel_PCFG 包括两个关键算法：1) 一种具有内存负载平衡的新型 PCFG 存储结构，以最大化内存使用。2) 一种并行算法，将密码候选生成分配给不同的线程，并确保每个线程按降序生成。具体来说，我们设计了一个查找表，以便线程可以同时定位和生成密码候选。由于 Hashcat [13] 具有内置的 GPU 内核，我们在流行的密码破解软件 Hashcat [13] 中实现了 Parallel_PCFG。

我们在同一时间范围内（例如半小时）对六个泄露的公共密码数据集上的 Parallel_PCFG 的破解率进行了评估，并与包括 OMEN 、 PCFGManager 和其他基于 PCFG 的模型进行了比较。我们发现 Parallel_PCFG 在六个数据集上仅需半小时即可达到 30 ~ 60% 的破解率。特别是，Parallel_PCFG 仅需要最先进模型总时间的 14.33% 即可达到相同的破解率；Parallel_PCFG 平均仅需 10 分钟即可破解与最先进模型相似百分比的密码，展示了减少密码破解时间开销的有效性。我们还比较发现，Parallel_PCFG 在各种规则集下可以显著优于默认的 Hashcat，甚至优于优化的规则集 [4]。我们认为这项工作可以作为将基于 PCFG 的模型集成到 Hashcat 工业工具中的有前景的解决方案，展示了其在工业应用中的潜力。

我们将主要贡献总结如下：

- 我们提出了两种算法，以使基于 PCFG 的模型能够在 GPU 上并行生成密码候选。首先，我们为 PCFG 密码生成中使用的数设计了一种具有内存负载均衡策略的存储。其次，我们设计了一种将生成任务分配给多个线程的算法。
- 我们基于这两种算法在 Hashcat 中实现了 Parallel_PCFG，提高了基于 PCFG 的模型的密码生成速度。我们通过实证评估表明，Parallel_PCFG 可以显著减少密码破解攻击中的时间开销，展示了将 PCFG 集成到实际猜测场景中使用的 Hashcat 的潜力。具体来说，Parallel_PCFG 仅需要 14.33% 的时间即可实现与先前同类方法相似的破解率。
- 我们从时间角度提供了关于评估密码猜测性能的见解，而不仅仅是关注猜测次数。我们采取了重大步骤来解锁这一点：30 ~ 60% 的密码可以在一定时间内（例如，半小时内）被猜测。

组织。 我们首先在第二节中介绍背景知识，然后在第三节中解释相关概念（例如，现有的 PCFG 生成方法和 Parallel_PCFG 的挑战）作为预备知识。在第四节中，我们详细介绍了我们 Parallel_PCFG 的设计，它包括两个组件。我们在第五节中展示了我们的评估。最后，我们在第六节中讨论了经济成本，并在第七节中总结本文。

2 背景知识

2.1 密码破解攻击

在最近几十年中，密码破解攻击是实际攻击场景和研究领域的热点。一方面，黑客可以利用破解软件（例如，Hashcat 或 JtR ）通过比较通过将一组规则应用于字典获得的候选猜测来破解目标密码，并进一步破坏数据集数据库或窃取用户的个人数据属性。

另一方面，研究人员提出了几种高效的数据驱动猜测模型 [17], [19], [34] ，这些模型学习一个有教育意义的密码空间以生成更多的候选密码，这些候选密码用于与目标密码进行比较。通常，这些数据驱动模型通过成功破解密码所需的努力（即猜测次数）来评估密码的强度，其中它们采用蒙特卡洛 [8] 等高效技术将概率映射到猜测次数。

2.2 相关工作

加速密码破解攻击。 丰富的文献已经提出了有效技术来加速数据驱动模型的密码生成方法，这通常涉及优化数据结构或使用分布式计算机集群。一方面，他们不断改进模型结构以加速密码生成过程。对于基于 PCFG 的方法，Weir 等人 [14] 发布了一个编译的 PCFG 密码猜测器，称为 *Compiled-PCFG*，用C 编程语言编写，以实现更快的破解和更容易与现有的破解软件互连。对于基于马尔可夫的方法，Durmuth 等人提出了 OMEN [9](有序马尔可夫枚举器) 作为一种更快的密码猜测方法，它将概率分为十个级别，并一次性生成同一级别的候选密码。在另一方面，研究人员提出利用分布式计算机集群（即更多计算机资源）同时生成候选密码。例如，Hranický 等人 [11] 基于分布式集群计算机，通过 Go 编程语言改进了 PCFG 的生成过程，称为 *PCFG-manager*。他们引入了一个 PCFG 管理器来处理多个客户端的

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

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

2.3 Password Cracking Scenarios

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

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

2.4 Threat Model

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

3 Preliminaries

3.1 Probabilistic Context-free Grammars

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

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

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

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

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

基于多台计算机同时生成候选密码。

数据驱动模型与破解软件的结合。 2020 年，Radek Hranick´y [12] 建议将数据驱动模型与猜测软件结合，特别是将高效的 PCFG 与基于 GPU 的 Hashcat 猜测软件结合，以发起密码破解攻击。由于破解软件通常依赖于专家总结的规则集 [22] 来达到更高的破解性能，这也限制了它们的更广泛应用。对于在现实世界破解中使用的数据驱动模型，攻击者可以将生成的候选密码传输到猜测软件中破解密码文本。不幸的是，通常需要一天或更多天才能生成大量候选密码（10^14）。然后，现有机制通常采用管道模式 作为通道，将每个候选密码串行传输到 Hashcat。瓶颈在于每个候选密码的生成速度。这种不理想的流程通常导致显著的时间浪费，限制了其在现实世界猜测中的实际应用。此外，之前在 2018 年，在 Hashcat 的官方论坛上，Weir （基于 PCFG 的猜测方法的发起人之一，其在论坛上的昵称为 lakiw ）和 Steube （ Hashcat 的作者之一，其在论坛上的昵称为 atom ）讨论了向 Hashcat 添加 PCFG 模型的主题。他们尝试按优先队列的顺序将候选密码分配给每个线程。然而，他们讨论说他们必须初始化优先队列并从头开始生成密码，以确定优先队列中特定位置的特定密码，这使得定位与每个线程相关的候选密码变得具有挑战性，最终结束了这个主题。

2.3 密码破解场景

密码破解场景通常分为在线和离线猜测场景，其主要区别在于攻击者可以尝试不同的猜测。由于服务提供商的限制，攻击者在破解目标密码时尝试的猜测次数较少（例如，少于 1,000 次），因此在线猜测通常是基于个人信息的针对性猜测任务，以破解特定用户的密码。

离线破解攻击通常在降序概率下尝试大量候选猜测，以在密码服务器被攻破的情况下破解哈希的通用密码数据库。这些候选密码（基于数据驱动模型或规则转换的密码）随后使用哈希算法（例如，MD5 、 SHA-1 ）进行哈希处理，并与密文密码进行比较，其中匹配概率即为破解率。现有的离线猜测工作 [24], [38] 通常评估在 10^14 次猜测下有多少候选密码与目标密码匹配。然而，如果漏洞被公开，可能会出现多个攻击者尝试破解同一密码数据库的情况。在极端情况下（例如，数字钱包），只有第一个破解密码的攻击者才能获得奖励，而其他攻击者则浪费了所有资源支付。这种竞争使得攻击者需要考虑攻击的时间成本，而大多数现有的离线猜测工作都低估了这一点。

2.4 威胁模型

在本文中，我们主要关注通用的离线猜测场景 [32], [38] ，这些场景旨在通过候选密码的匹配来恢复密文和明文密码。我们认为目标猜测、掩码猜测或其他现实世界的猜测 [21], [31], [39] 超出了本文的工作范围。我们在有限的时间内（例如，半小时）生成密码候选，并测量匹配目标的密码百分比，作为攻击者可以在一定时间内破解的密码。具体来说，我们假设攻击者可以使用 Hashcat 软件和一个训练好的 PCFG 模型，其中他们可以并行执行密码生成任务，以展示 PCFG 模型在现实世界中的潜在破解机会。

3 预备知识

3.1 随机上下文无关文法

随机上下文无关文法（ PCFG ）最早由 Weir 等人于 2009 年提出（简称 PCFG-2009 ）。他们不断优化该方法，并提出了改进版的 PCFG （简称 PCFGv4.1 [33] ）。

PCFG 主要分为两个阶段：**训练**和**生成**。在训练阶段，PCFG 基于训练数据集统计信息及其概率。我们描述统计信息如下：基于训练集的文法（又名，结构）和特定字符串（与结构相关）及其概率。文法（如 L_4D_4 的结构）描述了构成密码的字符类别和长度，其中同构的 L_n 、 D_n 和 S_n 指结构中的子结构，每个子结构包含由字母、数字或特殊字符组成的特定字符串，并具有 n 个字符。例如，L_4 是一个包含特定字符串如 “pass” 的子结构。

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

基于 PCFG 的生成短语。 在生成候选密码的过程中，PCFG 使用优先队列来确保具有更高概率的候选密码的优先级。为了减少队列的内存开销，Aggarmal 等人 [1] 采用了预终端结构和 Deadbeat dad 算法。具体来说，这有助于阐明 PCFG-based 猜测模型中候选密码生成中的几个概念。

- 容器：容器指的是具有相同字符类别、长度和概率的所有特定字符串。通常，一个子结构（例如，L_4 ）可以包含具有不同概率的多个容器。
- 预终端（***PT***，简称）：预终端（ PT ）是用于密码生成的基本结构，指的是从语法中的所有子结构中收集的容器的集合。

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

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

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

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

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

PT-1 (L_4D_4)

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

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

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

3.2 Challenges of Parallel Password Generation

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

3.2.1 GPU parallelism logic.

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

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

3.2.2 Issue analysis

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

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

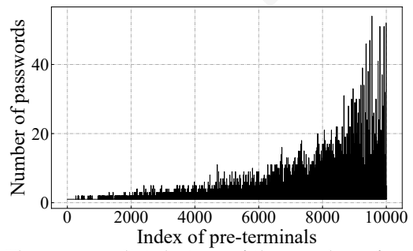


Fig. 2: The uneven distribution of the number of candidate passwords generated by different pre-terminals.

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

（结构）。因此，从同一 PT 生成的候选密码具有相同的概率。

图 1 展示了一个 PT 的示例，其中包含两个容器。该 PT 来自由两个子结构 L_4 和 D_4 组成的结构 L_4D_4，其顶部两个容器（来自 L_4 和 D_4）构成了第一个预终端 (PT-1)。例如，与子结构 L_4 相关的顶部容器是 "pass, word" 的集合，其概率均为 0.2。

PCFG 通过 PT 的结构生成候选密码，从同一 PT 生成的候选密码具有相同的概率。PCFG 使用特定字符串的笛卡尔积来生成候选密码。如图 1 所示，两个特定字符串的索引集均为 {1，2}，从而产生以下四个笛卡尔积对：{(1, 1), (1, 2), (2, 1), (2, 2)}

每对组合可以描述构成候选密码的特定字符串的组合，例如，(1, 1) 描述了候选密码 *pass1234*，它由来自 L_4 的 *pass*（索引为 1）和来自 D_4 的 1234（索引为 1）组成。

PT-1 (L_4D_4)

类型：	L	类型：	D
长度：	4	长度：	4
概率：	0.2	概率：	0.1
特定字符串：	1 个密码	特定字符串：	1 1234
	2 个单词		2 5678

图 1：来自 L_4D_4 结构的预终端示例。

在基于 PCFG 的密码生成过程中，特别是，优先队列最初包含每个结构中最可能的（概率较高）的预终端。假设图 1 中的预终端（ PT-1 ）是优先队列的顶部。在弹出预终端并根据其生成所有四个候选密码后，我们推导出优先级较低的 PT。例如，我们可以用概率为 0.1 的其他容器替换第一个容器（与 L_4 相关联）。然后，我们将新获得的预终端放回优先队列。优先队列将重新排列其中所有预终端的顺序。然后 PCFG 可以从优先队列中取出新的顶部预终端并重复上述生成步骤。

['3.2 密码生成并行挑战 ', ' 在这里，在理解了 PCFG 生成方法之后，我们介绍了将生成方法直接部署到不同线程的并行任务中时出现的问题，这也是挑战。我们先介绍 GPU 并行逻辑，然后分析在 GPU 上并行部署 PCFG 生成方法时出现的问题。', '3.2.1 GPU 并行逻辑。', ' 随着硬件的发展，GPU 作为一种最初设计用于加速 3D 图形渲染的电子电路，逐渐变得更加灵活和可编程，以解决更多并行问题。特别是，Hashcat 通常在 GPU 中执行猜测任务，这也激励我们基于其在数据驱动模型中卓越的猜测性能，开发基于 GPU 的并行方法用于基于 PCFG 的猜测模型。通常，我们将 GPU 在并行任务中的特性总结如下。', ' · GPU 以固定大小的空间（即 128 字节）从内存中读取数据，这也被称为缓存行策略。因此，当缓存行存储的数据较小时，GPU 需要多次读取数据，从而导致空间和传输时间成本。']

- GPU 以固定大小的空间（即 128 字节）从内存中读取数据，这被称为**缓存行**策略。因此，当缓存行存储的数据量较小时，GPU 需要多次读取数据，从而导致空间和传输时间成本。
- GPU 采用 **warp**(即，多线程) 来并行执行 SIMT(单指令多线程) 结构中的任务。warp 使用不同的数据资源执行相同的指令，其运行时间取决于最慢的线程。

3.2.2 问题分析

基于上述特性，在 PCFG 中并行生成候选密码仍然面临以下问题：

不平衡问题： 基于 PCFG 的猜测方法通过预终端为单位生成候选密码，即它们从顶层预终端开始，枚举所有可能的候选密码，然后转向下一个预终端。然而，不同预终端生成的候选密码显著不平衡，即概率较低的预终端可以生成更多候选密码。如图 2 所示，我们统计了 CSDN 数据集中前 10,000 个预终端生成的候选密码数量，发现最大值为 54，最小值仅为 1。这是因为容器的规模变得不平衡，特别是那些具有相同较高概率的特定字符串较少，而大多数特定字符串共享较低概率，导致这种现象。然后当我们直接将不同 PT 相关的生成任务分配给不同的线程时，生成速度可能会受到显著限制，因为 GPU 的运行时间取决于 warp 中最慢的线程 [11], ，从而促进到一种新的平衡存储结构。

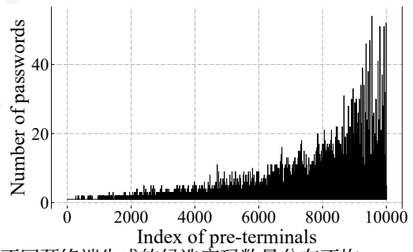


图 2：不同预终端生成的候选密码数量分布不均。

并行问题： 现有的基于 PCFG 的候选密码生成方法处于 PTs [11] 级别，并使用优先队列，但是优先顺序可能被破坏

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

4 PARALLEL_PCFG: DESIGN

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

4.1 Overview

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

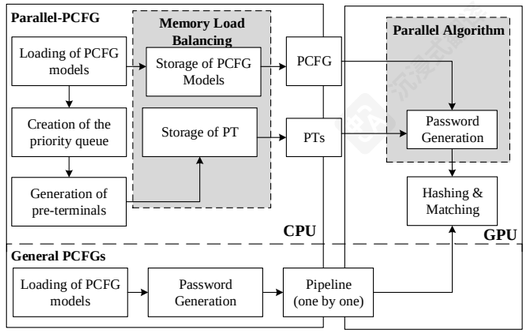


Fig. 3: The workflow of PCFG-based models in `Hashcat`, whose shaded part is our main modified module.

4.2 Memory Load Balancing

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

4.2.1 Storage of PCFG models.

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

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

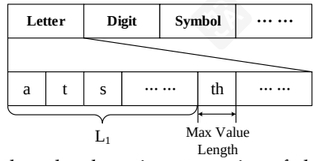


Fig. 4: The three-level sorting strategies of the dictionary storage for all the specific strings in PCFG models.

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

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

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

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

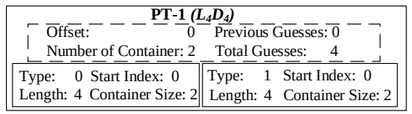


Fig. 5: Our storage structure of the pre-terminal.

4.2.2 Storage of PTs.

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

当我们并行生成 PT 时。也就是说，我们无法通过不同的线程直接并行生成 PT。因此，我们仔细确定并行粒度是在候选密码的级别（存储在训练好的 PCFG 模型中的特定字符串）。这样，它需要一种周密的并行算法来将候选密码生成任务分配给不同的线程，以及一种仔细的设计原则来定位每个特定的字符串。

4 PAR AL LE L_PCFG: 设计

在本节中，我们介绍了支持 PCFG 方法并行密码生成的设计，主要包括两个关键组件：具有内存负载均衡策略的存储和一个分配生成任务给不同线程的并行算法。

4.1 概述

我们在图 3 中展示了并行 PCFG 模型在 Hashcat 软件中部署的工作流程。主要区别在于设计的两个算法（阴影部分），这些算法能够在 Hashcat 的 GPU 内核上并行生成密码，而现有的通用模型通常采用 管道模式 将每个生成的候选密码按顺序传输到 GPU。Parallel_PCFG 首先加载 PCFG 模型并在 CPU 上创建预终端的优先队列。然后，我们使用一种具有负载均衡策略的设计存储结构。基本上，我们采用字典来存储和定位训练好的 PCFG 模型中的所有特定字符串和 PT，Parallel_PCFG 将 PCFG 模型和 PT 传输到 GPU。接下来，Parallel_PCFG 将密码生成任务分配给不同的线程以并行生成候选密码。最后，Parallel_PCFG 利用 Hashcat 的 GPU 内核中实现的哈希函数来加密这些密码，并将加密的密码与目标密码进行匹配。

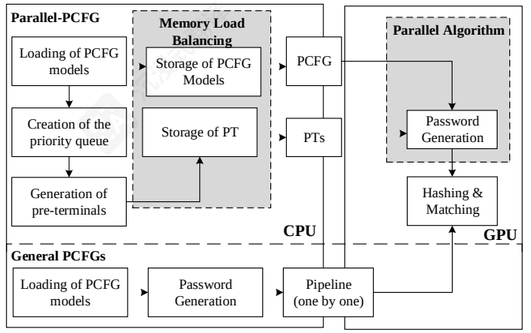


图 3：Hashcat 中基于 PCFG 的模型的流程图，其中阴影部分是我们主要修改的模块。

4.2 内存负载均衡

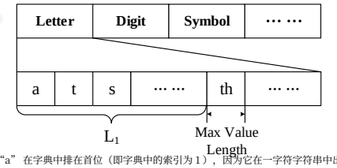
在这里，我们介绍我们的负载均衡机制，该机制将训练好的 PCFG 模型中的所有特定字符串均匀存储。

以及基于 PCFG 模型生成的 PT。首先，我们设计一个字典来存储 PCFG 模型中的所有特定字符串。其次，我们设计 PT 的存储结构以提高内存访问效率。

4.2.1 PCFG 模型的存储。

对于训练好的 PCFG 模型中的所有特定字符串，我们设计了一个一维数组，并使用 <type, length and probability> 的顺序的三级原则来存储它们。也就是说，我们首先按类型（即字母、数字或特殊符号等）对它们进行排序；对于具有相同字符类型的特定字符串，我们按长度（即从短到长）进行排序；对于具有相同类型和长度的特定字符串，我们按降序概率（即从高到低）进行排序。

以图 4 为例，字符 "*a*" 在字典中排在首位（即字典中的索引为 1），因为它在一字符字符串中出现的概率最高。



['以图 4 为例，字符 "a" 在字典中排在首位（即字典中的索引为 1），因为它在一字符字符串中出现的概率最高。','图 4：PCFG 模型中所有特定字符串的字典存储的三级排序策略。','然后，我们可以通过以下公式根据其在字典中的位置（即索引）定位特定字符串：','位置 = 起始位置 + 排名 × 长度','其中，起始位置指相同类型和长度的特定字符串的起始位置，排名指概率排序（即相同类型和长度字符串中概率降序排列的数字），长度指该字符串的长度。','为了在我们设计的字典中定位特定字符串，我们应该使用一个一维大小为 T × L 的数组来存储相同类型和长度的特定字符串的起始位置，其中 T 指类型数量，L 表示类型的最大长度。']

['以图 4 为例，字符 "a" 在字典中排在首位（即字典中的索引为 1），因为它在一字符字符串中出现的概率最高。','图 4：PCFG 模型中所有特定字符串的字典存储的三级排序策略。','然后，我们可以通过以下公式根据其在字典中的位置（即索引）定位特定字符串：','位置 = 起始位置 + 排名 × 长度','其中，起始位置指相同类型和长度的特定字符串的起始位置，排名指概率排序（即相同类型和长度字符串中概率降序排列的数字），长度指该字符串的长度。','为了在我们设计的字典中定位特定字符串，我们应该使用一个一维大小为 T × L 的数组来存储相同类型和长度的特定字符串的起始位置，其中 T 指类型数量，L 表示类型的最大长度。']

$$position = \text{起始位置} + rank \times \text{长度}$$

其中，起始位置指代相同类型和长度的特定字符串的起始位置，排名指代概率排名（即相同类型和长度的字符串中按概率降序排列的数值），而长度指代该字符串的长度。

为了在我们的设计词典中定位特定的字符串，我们应该使用一个一维大小为 $T \times L$ 的另一个数组来存储具有相同类型和长度的特定字符串的起始位置，其中 T 指代类型的数量，而 L 表示类型的最大长度。

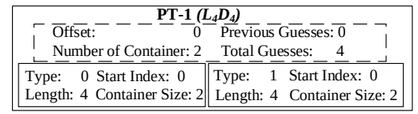


图 5：我们预终端的存储结构。

4.2.2 PTs 的存储。

基于 PCFG 模型的所有特定字符串的字典，我们设计了一种新的 PTs 存储结构，以使不同 PT 生成的密码均匀，支持并行生成算法。为了分配每个线程同时生成每个候选密码，

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

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

TABLE 1: Notations used in a PT.

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

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

24 bytes (head data) + 8×29 (body data) = 256

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

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

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

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

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

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

4.3 Parallel Algorithm

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

每个 warp 用于从相同的 PT 生成密码，减少线程之间的等待延迟。此外，现有的 PT 存储可能存储大量冗余信息，例如，"word" 可能既存在于与 L_4D_4 结构相关的 PT 中，也存在于与 L_4 结构相关的 PT 中。

首先，我们优化 PT 的存储，以一次存储特定的字符串。如图 5 所示，我们添加了四个值（即偏移量、容器数量、先前猜测和总猜测）。我们可以将预终端的数据结构分为头部数据（用虚线框表示），它存储预终端的统计信息，以及主体数据（用实线框表示），它存储 PT 中容器的详细信息。这样，原始结构的冗余存储被减少，也为后续并行算法的计算做好了准备。我们将这些值的详细含义总结如下，见表 1

表 1：PT 中使用的符号。

符号	描述
偏移量	它指的是最后生成的密码中线程的索引。PT 的最后一个密码。 PT-1 的偏移量为 0。当 PT-1 生成 4 个候选密码， PT-2 中的偏移量为 4，这描述了从上一个 PT 的移位值。
容器数量	当前容器中的数量 租用 PT。也就是说，PT 结构中的子结构数量， PT 结构中的结构数量。
以前的猜测	所有以前的 PT 生成的密码总数
总猜测次数	可破解的密码总数为 由当前 PT 生成。
类型	The t容器。of the specific strings in the
长度	The len th of the specific string in the g 当前容器。
起始索引	所有具有相同类型和长度的字符串中特定字符串的起始位置 相同类型和长度。
容器大小	当前容器中包含的特定字符串数量 当前容器中包含的特定字符串数量。

其次，我们将 PTs 的存储大小调整为缓存行大小的整数倍，以减少内存访问（即增加一次传输中的 PTs 数量）。这是因为当 PT 的存储大小大于缓存行时，GPU 会多次加载 PTs。具体来说，我们使用 8 字节、 8 字节、 1 字节和 7 字节分别用于偏移量、先前猜测、容器数量和总猜测量（ 8 + 8 + 1 + 7 = 24 字节）； 对于主体数据中的每个容器，我们使用 1 字节、 1 字节、 1 字节和 5 字节分别用于类型、起始索引、长度和容器大小（ 1 + 1 + 1 + 5 = 8 字节）。此外，我们将密码中的最大容器数量限制为 29，因为密码数据集中超过 29 个子结构的密码很少（ <0.1% ）。当我们使用 29 作为最大容器数量时，每个 PT 将占用 256 字节，具体计算如下：

24 字节（头部数据) + 8×29 （体数据） = 256 由于 GPU 的缓存行大小通常为 128 字节，

我们的 PTs 存储大小正好是缓存行大小的两倍，以便我们可以缓解由于不适当的存储而导致的 PTs 多次读取。

最后，我们使用一种平滑技术，重新分配具有相同类型和长度的特定字符串的概率，以便每个容器可以包含相同数量的特定字符串，进一步保证每个 PT 生成的候选密码数量相似。通过这种方式，我们可以使 warp 进程中的线程并行生成密码，从而减少 warp 中线程之间的等待延迟。

为此，我们按概率降序对具有相同类型和长度的特定字符串进行排序，将它们分成几个组，并重新分配同一组字符串的概率为该组所有字符串的平均概率。特别是，我们设置每个组中的特定字符串数量为 2^n ($n \geq 1$)，换句话说，调整后的容器大小为 2^n。我们选择 2^n 的大小，以确保 PTs 会产生 32 （ 25 ）个整数倍的候选密码。据我们所知，warp 通常包含 32 个线程。因此，有了 32 个整数倍的候选密码，warp 可以处理来自同一 PT 的候选密码。当 n 变得更大（即 ≥ 5 ）时，所有 PT 生成的候选密码数量可以是 32 的整数倍；当 n 变得更小，我们应该重复一些 PTs，以使生成的密码达到 32。以 $n = 3$ 为例，则每个容器包含 8 个特定字符串。因此，来自具有两个以上子结构（容器数量超过两个）的结构（当容器数量正好为两个时为 8 × 8 ）的 PTs 将直接生成 64 或更多候选密码。对于只有一个容器的 PTs，我们使用重复特定字符串 4 次的额外存储来满足 32 个线程，以确保 warp 中的线程可以处理来自同一 PT 的密码。

我们在表 2 中展示了不同 n 的重复次数。我们可以得出结论，当我们选择较小的 n 时，我们需要更多的重复次数，从而导致空间浪费。相反，当我们选择较大的 n 时，我们的平滑处理变得粗粒度，导致更多特定的字符串共享相同的概率，从而影响最终的猜测性能。此外，我们通过实证分析 n 的选择，并在第 5.3 节中推荐该参数为 3。

表 2：n 选择的重复次数。 "-" 表示相关容器大小中无重复。

容器 size	PT 中的容器数量			
	1	2	3	4
2 (n = 1)	16 times	8 times	4 times	2 times
4 (n = 2)	8 times	2 times	-	-
8 (n = 3)	**4 times**	-	-	-
16 (n = 4)	2 times	-	-	-

4.3 并行算法

然后，我们设计了一种并行算法，每个线程同时生成一个候选密码。此外，我们在 warp 中的线程生成密码来自

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

4.3.1 Candidate Password Identification

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

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

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

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

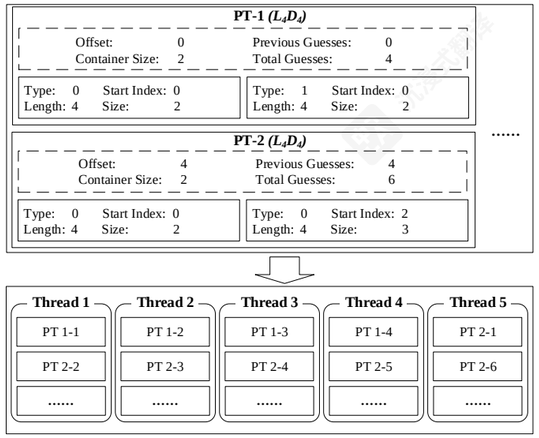


Fig. 6: The allocation of parallel generation tasks on GPU.

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

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

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

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

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

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

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

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

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

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

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

4.3.2 Specific String Identification

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

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

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

使用相同的 PT 来充分利用并行性。为此，我们应该为每个线程中的候选密码识别设计两个索引算法，以及构成候选密码的特定字符串识别。

4.3.1 候选密码识别

首先，我们设计一个二维查找表来定位每个线程中生成的候选密码。具体来说，我们使用 PT id-Guess id （例如，j-k ）来描述从 PT id （ j ）个预终端生成的 Guess id （ k ）个候选密码。也就是说，我们引入 *guess id* 来描述候选密码在 PT 生成顺序中的索引（即，第一维索引）。然后，我们可以总结 *Guess id* 可以通过公式 1 计算：

$$\begin{aligned} Guess\ \ id = (Thread\ \ id - Offset[PT\ \ id] + Thread\ \ size) \,\%\, Thread\ \ size \\ + n \times Thread\ \ size \\ \text{where } Guess\ \ id \le Total\ \ Guess[PT\ \ id] \text{ and } n \in N \end{aligned} \tag{1}$$

其中 *Offset* 和 *Total guesses* 描述了从最后一个 PT （ *Thread id* 在最后一个密码生成的最后一个 PT ）的偏移值以及当前 PT 生成的候选密码总数。

在公式 1 中，我们可以通过 线程 *id* 减去 偏移量 来获取当前 PT 生成的第一个密码的线程索引。在这里，为了避免出现负值，我们在加上 线程大小 后，还取模总线程数（即线程大小）。然后，我们向 猜测 *id* 添加线程总数的整数倍来标记下一个候选密码，同时保证 猜测 *id* 不会超过 总猜测次数，因为当前 PT 只能生成 总猜测次数 个密码。我们遍历线程 *id* 和 *PT id* 的值从 1 开始，并为每个 线程 *id* 和 *PT id* 计算相应的 猜测 *id* ，直到 猜测 *id* 大于 总猜测次数。然后，我们开始计算下一个 PT 生成的候选密码的 猜测 *id* 。

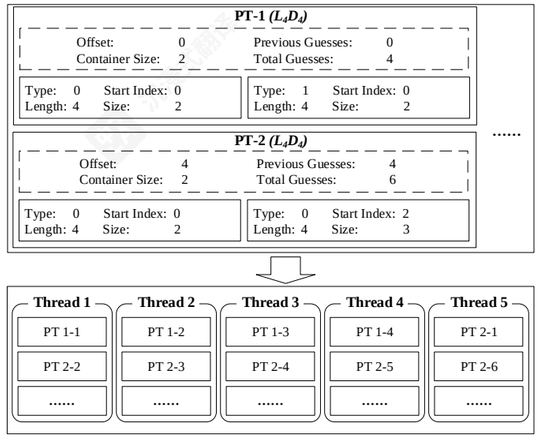


图 6：GPU 上并行生成任务的分配。

然后，我们以图 6 为例来展示将密码生成分配给不同的线程的过程，基于 猜测 *id* 索引算法。我们使用 "PT*j-k*" 作为二维查找表的结果来定位从 *j-th*(*PT id*) PT 生成的 *k-th* (猜测 *id*) 候选密码。

在这个例子中，我们假设线程大小 是 5。对于第一个 PT-1，偏移值是 0 （即，偏移 [1] 是 0 ），并且总猜测次数 [1] 是 4 ；对于 PT-2，偏移 [2] 是 4，并且总猜测次数 [2] 是 6。

然后，我们计算每个线程的猜测 *ID* 。对于第一个线程（线程 *ID* = 1 ），当 *PT ID* 是 1 时，我们可以得到

猜测 *ID* =(1 - 0 + 5) % 5 + $n \times 5$ = 1 + $n \times 5$ 考虑到猜测 *ID* 应该小于或等于总猜测次数 （即，这个例子中的 "4" ），所以猜测 *ID* 只能取 1。也就是说，我们可以从第一个线程（线程 *ID* = 1 ）生成的密码的第一个索引定位到 "PT 1-1"。

Similarly, we calculate the *Guess id* when the *PT id* is 2 for the first thread (*Thread id* = 1). When *PT id* is 2, the *Guess id* is calculated as:

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

同样地，我们取 2 作为猜测 *ID*，因为猜测 *ID* 应该小于或等于 6。也就是说，我们可以定位第一个线程（线程 *ID* = 1 ）生成的密码的下一个索引为 "PT 2-2" 。

对于每个线程，我们可以重复上述步骤以获取 $PT\,j-k$ 来定位来自 j th PT 的 k th 候选密码。通过更改 线程 *ID* 的值，我们可以计算其他线程生成的密码的索引。然后我们只需从 PT $j-k$ 中定位候选密码的特定字符串，然后每个线程就可以生成候选密码。

表 3：针对不同 <code>Guess id</code> （我们以图 6 中的 PT-1 为例）的特定字符串索引（即 <code>Str index</code> ）。

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

4.3.2 具体字符串识别

为了生成候选密码，我们仍然需要定位最终生成它们的特定字符串。当我们获得候选密码的 *Guessid* 时，我们可以利用 *Guessid* 来逆向推导出 PCFG 模型中存储的每个特定字符串的索引。首先，我们展示计算 *Guess id* 的过程。基于生成方法中的笛卡尔积，从 PT-1 （如图 6 所示）生成的四个候选密码在表 3 中推导出来（详细见第 3.1 节）。然后，我们可以将表 3 中 *Guess id* 与特定字符串索引（ $Str\ index[id]$ ）之间的关系总结为公式 2：

$$Guess\ \ id = (Str\ \ index[1] - 1) \times Container\ \ size[1] + Str\ \ index[2] \tag{2}$$

如公式 2 所示，对于具有 *Guess id* 为 1 的候选密码，其 *Str index*[1] 和*Str index*[2] 都为1。 $Container\ \ size[Container\ \ id]$ 指的是总数

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

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

We abstract Formula 2 into the algorithm 1.

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

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

We abstract the above steps into Algorithm 2.

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

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

5 Evaluation

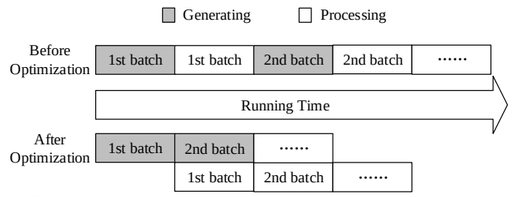


Fig. 7: Comparison of running process before and after the optimization of the running logic of `Parallel_PCFG`.

Algorithm 1 Forward algorithm for *Guess_id* calculation.

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

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

Algorithm 2 Backward algorithm of *Str_id* calculation.

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

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

5.1 Experiment Settings

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

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

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

*Container id*中的特定字符串在相应容器中的情况。然后，本公式第一行的示例是

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

我们将公式 2 抽象为算法 1。

基于公式 2，我们可以推导出通过模运算定位特定字符串索引的逆向算法。在表 3 中，给定 *Guess id* 为 3，计算每个容器中特定字符串索引的步骤如下：

- 初始状态：*Str index* 是 {*-1*, - 1}, *Guess id* 是 *3*, *Container size* 是 {2, 2}。
- 步骤 *1*： *Str index* 变为 {-1, 1}, *Guess id* 变为 2。*Str index* [2] = *Guess id* % *Container size* = 3 % 2 = 1。然后，*Guess id* = ⌈ *Guess id* ÷ 容器大小 [2] ⌉，即 *Guess id* = ⌈ 3 ÷ 2 ⌉ = 3 | 2 + 1 = 2。
- 步骤 *2*: *Str index* 变为 {2, 1}。计算结束。*Str index*[1] = *Guess id* % *Containersize*[1] = 2 % 2= 0。当取模结果为 0 时，我们需要加上当前容器中特定字符串的数量（即，*Container size*）。因此，*Str index*[1]⊨ *Guess id* % *Container size*[1] + *Container size*[1],也就是说，*Str index* [1] = 2 % 2 + 2 = 2。

我们将上述步骤抽象为算法 2。

完成设计。 我们使用 *Guess id* 计算方法来计算 *PT id-Guessid* ，以定位每个线程生成的候选密码。然后，我们通过 *Guess id*计算候选密码相关联的特定字符串的索引。起始索引存储在一个 PT 中，该 PT 描述了训练的 PCFG 模型中所有特定字符串字典中具有相同类型和长度的特定字符串的起始位置。偏移值可以通过 *Str id* 和 起始索引 的加法获得。我们利用具有所有相同类型和长度的特定字符串起始位置的一维数组来获取其位置，然后根据字典获取特定字符串。最后，我们将每个特定字符串连接起来以获得候选密码。

运行逻辑的实现优化。如图 7 所示，现有的生成逻辑是串行地实现密码生成和预终端处理，导致不必要的等待时间开销。在优化的实现中，我们异步地生成密码和处理预终端。当 GPU 处理前一批预终端时，CPU 开始生成下一批预终端。我们使用两个相同大小的空间和 CPU 中的两个线程，异步地执行预终端的生成和处理，其大小等于一次传输中可以容纳的预终端对应的总空间。为了避免不同线程访问相同空间时的冲突，我们通过锁定和解锁两个空间来使用信号量和互斥锁，并交替释放空信号和满信号。

5 评估

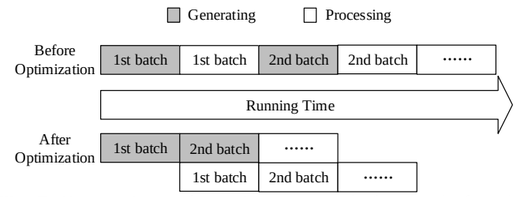


Fig. 7: Comparison of running process before and after the optimization of the running logic of `Parallel_PCFG`.

算法 1 用于猜测 *id* 计算的向前算法 n.

输入：字符串索引，容器大小，容器数量输出：猜测 *ID*

1: 容器 *id* ← 1 *2:* 猜测 *id* ← *-1 3: while* 容器 *id* ≤ 容器数量 *do 4: if* 猜测 *id* == *-1 then 5:* 猜测 *id* = 字符索引 [容器 *id*] *6: else 7:* 猜测 *id* = *(* 猜测 *id - 1)* × 容器数量 [容器 *id*] + 字符索引 [容器 *id*] *8: end if 9:* 容器 *id* += 1 *10: end while 11: return* 猜测 *id*

算法 2 反向算法的 *Str id* 计算 .

输入：猜测 *id*，容器大小，容器数量输出：字符串索引

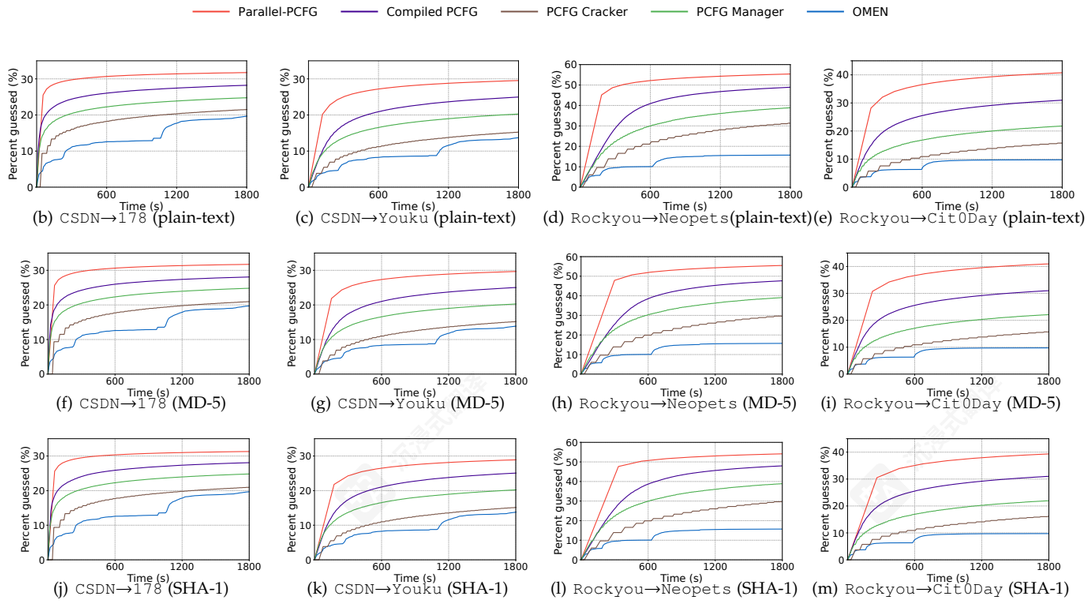
1: 容器 *id* ← 容器数量 2: 字符串索引 ← {}3: **while** 容器 *id* >0**do**4: *mod* = 猜测 *id*% 容器大小 [容器 *id*]5: **if** *mod* == 0**then**6: *mod* = *mod* + 容器大小 [容器 *id*]7: **endif**8: 猜测 *id* = ⌈ 猜测 *id* ÷ 容器大小 [容器 *id*] ⌉9: 字符串索引 . *addFirst*(*mod*)10: 容器 *id*-= 11: **endwhile**12: **return** 字符串索引

5.1 实验设置

系统环境。我们使用一台配备 Intel Xeon Silver 4210 (2.20GHz, 40 核) CPU 、 128 GB 内存和四块 GeForce RTX 2090Ti 显卡的计算机。

数据集。在评估实验中，我们使用了六个从真实网站泄露的大规模密码数据集：CSDN 、 178 、优酷、 Rockyou 、 Neopets 和 Cit0Day。这些数据集在地域范围和时间跨度上具有代表性，并在相关研究中被广泛使用 [2], [7], [20], [23], [25], [26], [28], [29], [30], [35], [36], [37]。我们在表 4 中展示了这些六个数据集的基本信息。

对于数据集清理 [17], [38] , , 我们清理包含非打印 ASCII 码的密码和长度超过 32 的密码。为了评估猜测方法（即破解新的、未见过的密码）的泛化能力，我们从训练集中移除重复密码，并获得了总共 120



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

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

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

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

TABLE 4: Information of six leaked real password datasets.

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

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

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

5.2 Experiment Comparison and Results

We empirically compare the cracking rate in terms of the comparison against the state-of-the-art data-driven models

TABLE 5: Baseline methods

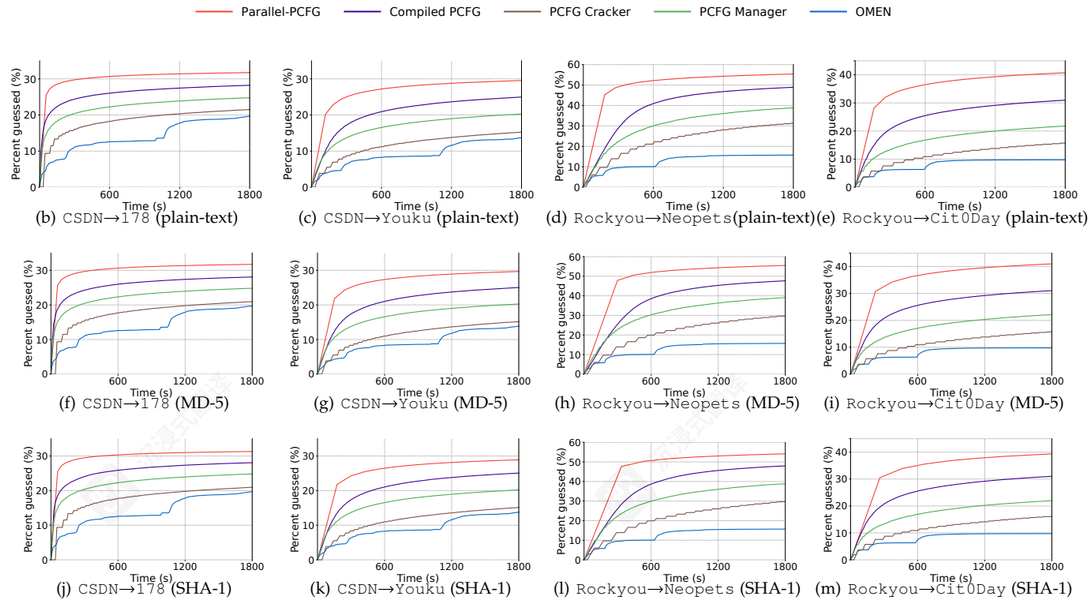
Method	Description
PCFG Cracker [1]	Python-based PCFG Generation
Compiled PCFG [2]	C-based PCFG Generation
PCFG Manager [3]	PCFG generation for Distributed Clustering
OMEN [4]	Markov-based Generation

and the Hashcat-based mask and rule-based attacks.

5.2.1 *Comparison against data-driven models.*

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

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



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

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

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

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

表 4：六个泄露的真实密码数据集的信息。

数据集	Language	Year	原始密码	使用密码
CSDN	中文	2011	6,425,243	6,422,884
178	中文	2011	9,071,979	3,461,974
优酷	中文	2016	47,607,615	25000000
Rockyou	English	2009	32584165	32582532
Neopets	English	2016	67,672,205	67,672,205
Cit0Day	英文	2020	86,835,796	25,000,000

一千万个密码。此外，Hashcat 通常在读取大规模数据集时会去重并调整附加参数，因此我们在表 4 的最后一列显示了测试数据集的唯一数量，同时从优酷和 Cit0Day 中随机选择了 2500 万个唯一密码以避免附加参数进行公平比较。我们主要使用三种测试用例：明文、 MD-5 哈希函数加密和 SHA-1 哈希函数加密来评估明文密码破解攻击和密文密码破解攻击。

伦理声明。我们声明，我们的研究专注于数据集的整体特征，而没有对个体特征进行分析，以满足伦理实践的要求。我们通过使用相应的哈希函数对泄露的明文数据集进行加密来获取加密数据集，而不是试图破坏泄露的加密数据集。也就是说，我们没有破坏现实世界中未泄露的数据集。我们相信我们的研究符合伦理关切。

5.2 实验比较与结果

我们通过对比最先进的数据驱动模型，对破解率进行了实证比较

表 5：基线方法

方法	描述
PCFG Cracker [1]	基于 Python 的 PCFG 生成 on
编译后的 PCFG [2]	基于 C 的 PCFG 生成
PCFG 管理器 [3]	PCFG eneration for Distributed g 聚类
OMEN [4]	基于马尔可夫的生成

以及基于 Hashcat 的掩码和规则攻击。

5.2.1 与数据驱动模型的比较。

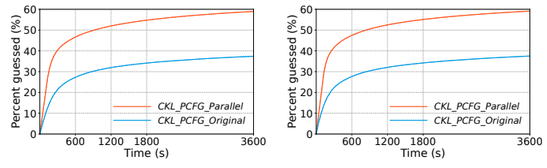
我们在 Hashcat 中部署 Parallel_PCFG，并与几种最先进的模型进行比较。我们在表 5 中展示了基准模型，对于基于 PCFG 的模型，我们将其与 PCFG 破解器、编译 PCFG 和 PCFG 管理器进行比较，因为它们在加速密码生成任务方面具有代表性。PCFG 管理器设计了一种分布式策略，以在分布式计算机集群中生成密码。此外，我们还与 OMEN [9] 进行了比较，因为 OMEN 是一种快速的密码生成模型，适用于基于马尔可夫的模型。由于基于神经网络的猜测模型 [35]（例如，生成超过 10^10 个密码需要 16 天）耗时较长，因此我们在本次工作中决定不评估这些模型。

对于 PCFG-cracker 和 Compiled-PCFG，我们采用通常的做法，通过管道模式 将它们生成的密码串行传输到 Hashcat 进行密文匹配。对于 PCFG Manager，我们将其并行线程数设置为 50，因为它在我们的系统环境下表现最佳。

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

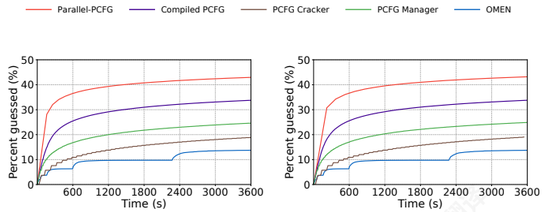
We show our results in Figure 8, in which we generally evaluate the cracking rates of several models within a fixed cracking time (i.e., half hour). We can find that Parallel_PCFG outperforms all of the baseline methods and achieves an average of 14.85% improvement than the best guessing performance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

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



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

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



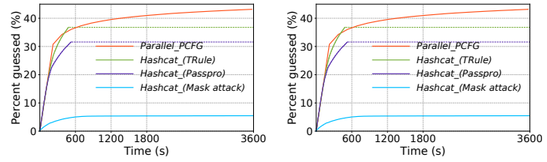
(b) Rockyou→Cit0Day (plain-text) (c) Rockyou→Cit0Day (MD-5)

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

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

5.2.2 Comparison against mask and rule-based attacks.

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



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

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

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

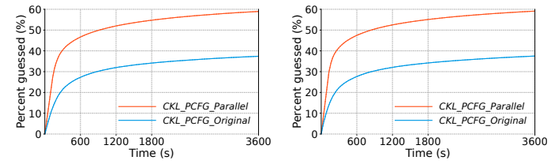
Reduced cracking time. We supplement to show how much cracking time our Parallel_PCFG can save in a longer cracking time (an hour) in Table 6, where we find that Parallel_PCFG achieves the similar cracking rate in an

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

对于 OMEN，我们将概率平滑的默认设置为 10 级以加快生成任务。对于 Parallel_PCFG，我们将容器大小设置为 8。对于 GPU 参数，我们将网格大小设置为 32，块大小设置为 1024。我们在第 5.3 节中通过经验评估了这些参数对 Parallel_PCFG 猜测性能的影响。

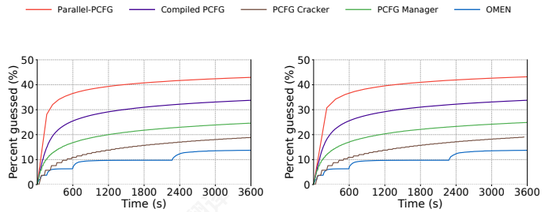
我们在图 8 中展示了我们的结果，其中我们通常评估在固定破解时间（即半小时）内几种模型的破解率。我们可以发现 Parallel_PCFG 优于所有基线方法，并比基线方法的最佳猜测性能平均提高了 14.85%（即 Compile-PCFG）。采用跨集群计算机（即大量计算机） CPU 的并行策略的 PCFG Manager，并不表现最佳，因为它们在更多 CPU 时更优越。然而，对于单个 CPU，PCFG Manager 会花费时间进行调度，从而实现中等猜测性能。因此，它成为 PCFG Manager 充分发挥其更好猜测能力的一个严格要求。

在更长的时间内进行破解。此外，我们可以观察到，半小时内的破解率几乎可以收敛，也就是说，在更长的时间内只有一点点的改进；另外，短时间（即半小时）在实际应用中比长时间更广泛使用。然而，我们仍在图 9 中补充了对较长时间的评估（即从半小时到一小时）。我们认为半小时的破解时间足以获得较大的收益，我们不必浪费更长的时间去追求微小的收益。



(a) Rockyou→Cit0Day(明文) (b) Rockyou→Cit0Day(MD-5)

图 10：CKL-PCFG 在并行设计前后的性能表现



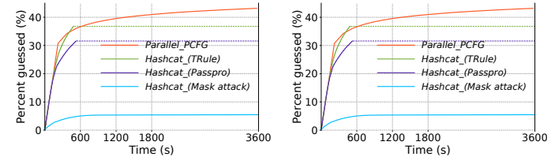
(b) Rockyou→Cit0Day (plain-text) (c) Rockyou→Cit0Day (MD-5)

图 9：与较长时间（例如一小时）设置的对比。

推广到其他 PCFG。我们的并行密码生成框架可以推广到其他基于 PCFG 的方法，其模式类型和长度（例如，"L 4"）的语法。然后我们将我们的并行密码生成框架应用于 GPU 上的优化 PCFG 模型（例如，CKL-PCFG [38]）。我们在两种猜测场景下进行实验，并将结果展示在图 10 中。我们观察到，*CKL PCFG* 并行 在明文和哈希测试集上均在一小时内实现了近 60% 的破解率，比 *CKL PCFG* 原版高约 22%，展示了我们并行框架在优化 CKL PCFG 上的生成能力。

5.2.2 Comparison against mask and rule-based attacks.

我们还将与掩码攻击和基于规则的攻击的效率进行比较，以便在比较时获得更多见解



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

图 11：与多种基于 Hashcat 的攻击进行比较。

我们在图 11 中展示了我们的结果，从中可以看出 Parallel_PCFG 实现了更高的破解率。掩码攻击模式通常效率最低，可能是因为训练的掩码密码与 Cit0Day 评估集存在不匹配，这是由于通用猜测的配置过于简单。总之，我们建议更广泛地结合数据驱动模型（例如 PCFG）和猜测软件（例如 Hashcat），因为纯猜测软件（例如 Hashcat）需要一个总结的规则集（例如 TRule）才能实现良好的性能。

减少破解时间。我们补充说明，在更长的破解时间（一小时）中，我们的 Parallel_PCFG 可以节省多少破解时间，如表 6 所示，我们发现 Parallel_PCFG 在类似的破解率下实现了

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

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

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

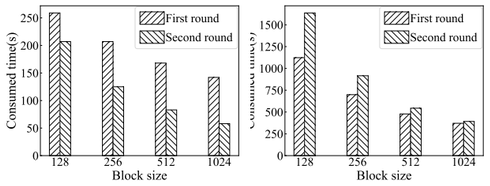
Experiments	Hash	Time	Reduction(%)
CSDN→178	Plain-text	13.58	77.37%
	MD-5	12.27	79.55%
	SHA-1	18.18	69.70%
CSDN→Youku	Plain-text	7.35	87.75%
	MD-5	7.95	86.75%
	SHA-1	9.80	83.67%
Rockyou→Neopets	Plain-text	5.00	91.67%
	MD-5	4.53	92.45%
	SHA-1	4.98	91.70%
Rockyou→Cit0Day	Plain-text	12.83	78.62%
	MD-5	10.30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

5.3 Influence of Parameters

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

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

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



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

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

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

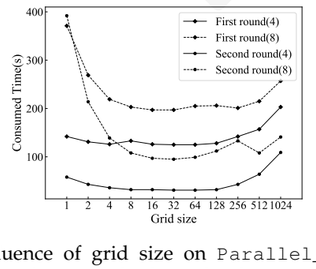
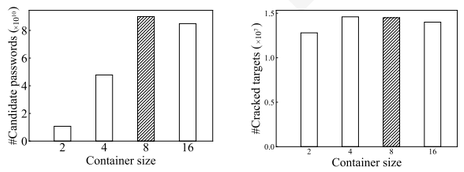


Fig. 13: Influence of grid size on Parallel_PCFG. The value in parentheses of the legend refers to the container size of each segment.

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

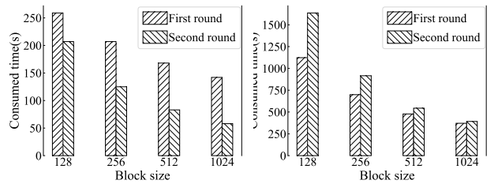


(a) Processed passwords (b) Cracked targets

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

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

平均需要 10 分钟，而 Compiled-PCFG 则需要更长时间，这表明 Parallel_PCFG 可以大幅减少时间开销，尤其是在测试集变得更大时。Parallel_PCFG 在 Youku 上的表现优于 178，并且同样在 Neopets 上比 Cit0Day 节省了更多的破解时间。Neopets 和 Youku 的测试集都显著更大，这表明 Parallel_PCFG 可以极大地受益于那些具有更大测试集的猜测场景。这可能是因为在更大的测试集上，模型需要更多 GPU 处理时间，而 Parallel_PCFG 可以减少这部分时间。当攻击者需要破解大量密码时，Parallel_PCFG 将非常有用。



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

图 12：块大小对并行 PCFG 密码处理速度的影响。

表 6：Parallel_PCFG 达到与 Compiled-PCFG 在更长时间（即一小时）内相同的破解率所需的时间（分钟）。

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

5.3 参数影响

在本节中，我们探讨了多个参数对 Parallel_PCFG 性能的影响。这里考虑了两个参数：GPU 的块大小（即单个块中的线程数）和 GPU 的网格大小（即网格中的块数）。我们还评估了容器大小对 Parallel_PCFG 性能的影响。

块大小的影响。我们将网格大小设置为 1，并使用 4 和 8 两种容器大小进行综合比较。我们考虑前两轮，因为第一轮涉及 GPU 内核的初始化，而第二轮及后续生成轮次没有初始化步骤。如图 12 所示，Parallel_PCFG 的消耗时间随着块大小的增加而减少。当块大小为 1024 时，我们获得了最小的开销。对于 Parallel_PCFG，块中线程数量的增加提高了并行处理能力，最终减少了消耗时间。因此，我们建议将块大小设置为 1024。

网格大小的影响。在探索网格大小的影响时，我们根据上一节的实验结果将块大小设置为 1024 以控制变量。实验结果如图 13 所示，其中 Parallel_PCFG 的消耗时间先减少后增加。我们可以观察到当网格大小在 16 到 64 之间时，时间消耗达到最小值。根据实验结果，我们建议将网格大小设置为 32，因为较小的网格大小会导致额外的计算开销。当网格大小过小时，GPU 线程的总数较少，因此每个线程在每次猜测时需要处理更多的密码。然而，当网格大小过大时，由于 GPU 中的流式多处理器数量有限，GPU 无法完全调度所有块，从而增加了整体的时间消耗。

大小过大或过小都会导致额外的时间开销。当网格大小过小时，GPU 线程的总数较少，因此每个线程在每次猜测时需要处理更多的密码。然而，当网格大小过大时，由于 GPU 中的流式多处理器数量有限，GPU 无法完全调度所有块，从而增加了整体的时间消耗。

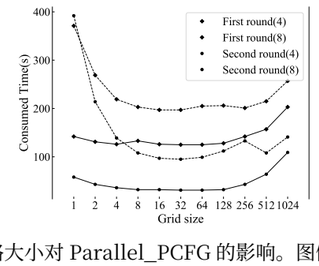
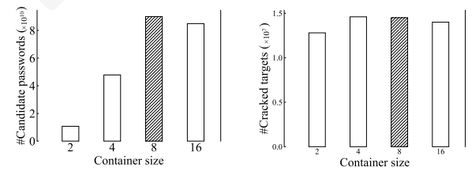


图 13：网格大小对 Parallel_PCFG 的影响。图例中括号内的值表示每个段的容器大小。

容器大小的影响。我们通过比较处理的候选密码数量和破解的目标密码数量，探索了容器大小对 Parallel_PCFG 性能的影响。我们选择 15 分钟作为固定的时间段，将块大小设置为 1024，将网格大小设置为 32。我们在图 14 中展示了实验结果。



(a) Processed passwords (b) Cracked targets

图 14：Parallel_PCFG 在同时处理密码数量和破解目标数量方面，容器大小的影响。符号 "#" 代表 " 数量 "。

通过观察图 14，我们可以发现，随着容器大小的增加，处理的

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

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

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

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

6 Discussion

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

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

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

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

7 Conclusion

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

表 7：Parallel_PCFG 使用 1 块 GPU 卡和 4 块 GPU 卡的破解率，表明 Parallel_PCFG 在密码生成任务中已达到 1 块 GPU 卡的潜力。Parallel_PCFG 达到其潜力并充分利用单块 GPU 卡，避免了多 GPU 设置的复杂性和成本。

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

密码数量和破解数量先增加后减少。考虑到将容器大小增加到 8 并不会显著提高处理速度和破解目标，我们在实验中确定容器大小为 8。以下分析原因。当容器较小时，一个预终端可以生成的候选密码数量较少，因此生成所消耗的时间比例较小。更多时间将用于数据传输。当容器较大时，GPU 处理所消耗的时间比例会更大。然而，由于单次传输时间不变，候选密码的处理速度会达到瓶颈。类似地，当容器较小时，Parallel_PCFG 必须重复某些预终端的内容，以确保一个 warp 中的所有线程处理相同的预终端，从而导致内存空间浪费。当片段项数量较多时，项在片段中的位置与其原始位置相比变化较大，导致生成的候选密码顺序发生较大变化，从而减少破解目标。

更多 GPU 卡的影响。 Parallel_PCFG 主要设计为充分利用单 GPU 的强大功能，通过高效的工作负载分配和线程管理，确保单 GPU 始终以最高潜力运行。通过专注于单 GPU 效率，Parallel_PCFG 避免了多 GPU 设置带来的复杂性和成本。通过表 7 中 1 张 GPU 卡和 4 张 GPU 卡的比较，我们证明了 Parallel_PCFG 在密码生成任务中达到了其全部潜力，解锁了 Parallel_PCFG 在用一张 GPU 卡生成候选密码时几乎达到其全部潜力的内在特性。我们相信，探索如何加速多卡场景是我们的未来工作，这需要周密和精心设计的方案。

6 讨论

经济成本： 我们使用 GPU 来加速基于 PCFG 的方法生成候选密码的速度，这显著减少了猜测攻击的时间成本。所用计算机的价格为 8000 美元，实验中使用的 GPU 卡价格为 1000 美元。在使用 GPU 之前，基于 PCFG 的方法每秒可以成功破解 2102 个密码，相当于每 1000 美元为 1 秒；使用 GPU 后，基于 PCFG 的方法每秒可以成功破解 2437 个密码，折合为每 1000 美元每秒 271 个密码。即使每秒可以以同样的经济成本（1000 美元）多破解 8 个密码，更不用说半小时或更长时间的密码破解了。总之，Parallel_PCFG 在 10 分钟内可以达到与传统基于 PCFG 的方法相似的破解速率。当经济成本增加 12.5% 时，时间成本缩短了 85.67%。

全 GPU 加速。运行逻辑 f Parallel_PCFG 已精心优化以实现 CPU-GPU 协作，这是通过异步生成（在 CPU 中）和处理（在 GPU 中）预终端来实现的，确保 GPU 核心被充分利用，最小化浪费时间并最大化吞吐量。通过上述措施，Parallel_PCFG 实现了更好的调度工作（例如，在 CPU 和 GPU 之间、平衡内存加载或并行威胁），以全速运行破解器。**与实际启示和防御的相关性**。本文通过关注基于时间的指标而不是简单地关注猜测次数，为密码猜测性能提供了宝贵的见解。我们认为，时间指标更准确地反映了现实世界的工业场景，而猜测次数在学术研究中更为常见。仅依赖猜测次数可能会产生误导，因为每次猜测尝试的成本不同可能会引入偏差。鉴于强调的漏洞，强制执行严格的密码创建策略至关重要。例如，我们可以鼓励用户使用密码管理器创建强密码。我们还鼓励使用双因素认证机制来加强认证。

未来工作： 提高生成密码猜测的速度是该领域的一个关键挑战。我们鼓励更多有针对性的研究，并强调理解密码猜测随时间演变的重要性。在未来的工作中，我们将探索针对特定场景（如掩码或目标猜测）的加速技术，因为它们具有独特的结构和定制需求。此外，我们计划研究如何更好地组织多张 GPU 卡以增强协作并显著加快密码候选生成。

7 结论

在本文中，我们利用并行技术加速了基于 PCFG 的方法的密码生成。我们主要提出了两种算法：一种具有内存平衡机制的结构存储和一种将生成任务分配给不同线程的并行算法。基于上述两种算法，我们在流行的密码破解软件 Hashcat 中提出了并实现了 Parallel_PCFG。我们从破解时间周期的角度对 Parallel_PCFG 的破解率进行了实证评估，发现 Parallel_PCFG 可以显著节省破解时间，并与其他几个竞争对手相比。特别是，Parallel_PCFG 仅用了 14.33% 的时间就达到了最佳模型实现的相同破解率，证明了 Parallel_PCFG 可以显著提高候选密码的生成速度并减少密码破解攻击的时间成本。

ACKNOWLEDGMENT

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

REFERENCES

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

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

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

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

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

[6] Dasio. Pcfg-manager.

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

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

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

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

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

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

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

[14] lakiw. Compiled-pcfg.

[15] lakiw. Pcfg-cracker.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

ACKNOWLEDGMENT

本文由国家自然科学基金（项目编号：62172100，U1836207 ）资助。

REFERENCES

[1] Sudhir Aggarmal 和 Charles Matthew Weir. *Using probabilistic techniques to aid in password cracking attacks*. 博士论文，佛罗里达州立大学，2010 年。

[2] Jeremiah Blocki 、 Benjamin Harsha 和 Samson Zhou. 关于离线密码破解的经济性。收录于 *2018* 年 *IEEE* 安全与隐私研讨会，*SP 2018*，会议论文集，*2018* 年 *5* 月 *21-23* 日，美国加利福尼亚州旧金山，第 853-871 页。IEEE 计算机协会，2018 年。

[3] Joseph Bonneau, Cormac Herley, Paul C. van Oorschot, and Frank Stajano. 寻求替代密码：Web 身份验证方案的比较评估框架。在 *IEEE* 安全与隐私研讨会，*SP 2012*，*2012* 年 *5* 月 *21-23* 日，美国加利福尼亚州旧金山，第 553-567 页。IEEE 计算机协会，2012 年。

[4] Alessia Michela Di Campi, Riccardo Focardi, and Flaminia L. Luccio. 密码破解者的复仇：密码破解工具的自动训练。在 Vijayalakshmi Atluri, Roberto Di Pietro, Christian Damsgaard Jensen, and Weizhi Meng 编辑的 计算机安全 - *ESORICS 2022* - 第 *27* 届欧洲计算机安全研讨会，丹麦哥本哈根，*2022* 年 *9* 月 *26-30* 日，会议录，第二部分，第 13555 卷的 计算机科学讲座笔记，第 317-336 页。Springer，2022 年。

[5] Alessia Michela Di Campi, Riccardo Focardi, and Flaminia L. Luccio. 密码破解者的复仇：密码破解工具的自动训练。在 Vijayalakshmi Atluri, Roberto Di Pietro, Christian Damsgaard Jensen, 和 Weizhi Meng 编辑的《计算机安全 - ESORICS 2022 - 第 27 届计算机安全研究欧洲研讨会，丹麦哥本哈根，2022 年 9 月 26-30 日，论文集，第二部分》中，第 13555 卷《计算机科学讲座笔记》，第 317-336 页。Springer，2022。D i P f as o. "c g-manager.

[6] Dasio. Pcfg-manager.

[7] Xavier de Carn´e de Carnavalet and Mohammad Mannan. 从非常弱到非常强：分析密码强度测量工具。在《第 21 届年度网络和分布式系统安全研讨会，NDSS 2014，美国加利福尼亚州圣地亚哥，2014 年 2 月 23-26 日》中。互联网协会，2014 年。

[8] Matteo Dell' Amico and Maurizio Filippone. 蒙特卡洛强度评估：快速可靠的密码检查。在 Indrajit Ray 、 Ninghui Li 和 Christopher Kruegel 编辑的《第 22 届 ACM SIGSAC 计算机与通信安全会议论文集，丹佛，美国，2015 年 10 月 12-16 日》中，第 158-169 页。ACM，2015。

[9] Markus D¨urmuth, Fabian Angelstorf, Claude Castelluccia, Daniele Perito, 和 Chaabane Abdelberi. OMEN：使用有序马尔可夫枚举器进行更快的密码猜测。在 Frank Piessens 、 Juan Caballero 和 Nataliia Bielova 编辑的《工程安全软件与系统 - 第 7 届国际研讨会，ESSoS 2015，米兰，意大利，2015 年 3 月 4-6 日。论文集》中，第 8978 卷的《计算机科学讲座笔记》中，第 119-132 页。Springer，2015。

[10] Cormac Herley 和 Paul C. van Oorschot. 一份承认密码持续存在的研究议程。 *IEEE Secur. Priv.*, 10(1):28–36, 2012。

[11] Radek Hranick´y, Filip Listiak, D´avid Mikus, 和 Ondrej Rysav´y. PCFG 密码破解的实际方面。 在 Simon N. Foley 编辑的 数据和应用安全与隐私 *XXXIII* - 第 *33* 届 *IFIP WG 11.3* 会议，*DBSec 2019*，美国查尔斯顿，*SC*，*2019* 年 *7* 月 *15-17* 日，会议记录，第 11559 卷的 计算机科学讲座笔记，第 43–60 页。Springer，2019。

[12] Radek Hranick´y, Luk´as Zobal, Ondrej Rysav´y, Dusan Kol´ar, 和 D´avid Mikus. 分布式 PCFG 密码破解。 在 Liqun Chen, Ninghui Li, Kaitai Liang, 和 Steve A. Schneider 编辑的 计算机安全 - *ESORICS 2020* - 第 *25* 届欧洲计算机安全研究研讨会，*ESORICS 2020*，英国吉尔福德，*2020* 年 *9* 月 *14-18* 日，会议记录，第一部分，第 12308 卷的 计算机科学讲座笔记，第 701–719 页。Springer，2020。

[13] Gabriele Gristina Jens Steube. Hashcat: 一个密码恢复工具 .

[14] lakiw. 编译后的 pcfg.

[15] lakiw. Pcfg-cracker.

[16] Enze Liu, Amanda Nakanishi, Maximilian Golla, David Cash 和 Blase Ur. 对密码破解软件进行解析性推理 . 在 *2019 IEEE* 安全与隐私研讨会 , *SP 2019*, 加利福尼亚州旧金山 , *USA*, *2019* 年 *5* 月 *19-23* 日 , 第 380-397 页 . IEEE, 2019.

[17] Jerry Ma, Weining Yang, Min Luo 和 Ninghui Li. 对概率密码模型的研究 . 在 *2014 IEEE* 安全与隐私研讨会 , *SP 2014*, 加利福尼亚州伯克利 , *USA*, *2014* 年 *5* 月 *18-21* 日 , 第 689-704 页 . IEEE 计算机协会 , 2014.

[18] Jens Steube Matt Weir. 向 Hashcat 的 " 大脑 " 中添加 pcfg.

[19] William Melicher, Blase Ur, Sean M. Segreti, Saranga Komanduri, Lujo Bauer, Nicolas Christin 和 Lorrie Faith Cranor. 快速、精简且准确：使用神经网络建模密码猜测性 . 在 Thorsten Holz 和 Stefan Savage 编辑的 第 *25* 届 *USENIX* 安全研讨会 , *USENIX Security 16*, 德克萨斯州奥斯汀 , *USA*, *2016* 年 *8* 月 *10-12* 日 , 第 175-191 页 . USENIX 协会 , 2016.

[20] Bijeeta Pal, Tal Daniel, Rahul Chatterjee, 和 Thomas Ristenpart. 超越凭证注入：使用神经网络的密码相似度模型。在 *2019 IEEE Security and Privacy Symposium, SP 2019*, 加利福尼亚州旧金山 , *USA*, *2019* 年 *5* 月 *19* 日至 *23* 日 , 第 417-434 页。IEEE, 2019。

[21] Dario Pasquini, Giuseppe Ateniese, 和 Carmela Troncoso. 通用神经破解机：从辅助数据中自配置的密码模型。 *CoRR*, abs/2301.07628, 2023。

[22] Dario Pasquini, Marco Cianfriglia, Giuseppe Ateniese, 和 Massimo Bernaschi. 通过深度学习和动态词典减少建模真实世界密码强度的偏差。在 Michael Bailey 和 Rachel Greenstadt 编辑的 第 *30* 届 *USENIX Security Symposium, USENIX Security 2021, 2021* 年 *8* 月 *11* 日至 *13* 日 , 第 821-838 页。USENIX Association, 2021。

[23] Dario Pasquini, Ankit Gangwal, Giuseppe Ateniese, Massimo Bernaschi, 和 Mauro Conti. 通过表示学习改进密码猜测 . 在 第 *42* 届 *IEEE* 安全与隐私研讨会，*SP 2021*，美国加利福尼亚州旧金山，*2021* 年 *5* 月 *24-27* 日 , 第 1382-1399 页 . IEEE, 2021.

[24] Joshua Tan, Lujo Bauer, Nicolas Christin, 和 Lorrie Faith Cranor. 对更强、更易用的密码的实用建议，结合最小强度、最小长度和黑名单要求 . 在 Jay Ligatti, Xinming Ou, Jonathan Katz, 和 GiovanniVigna 编的 *CCS'20: 2020ACMSIGSAC* 计算机与通信安全会议，虚拟活动，美国，*2020* 年 *11* 月 *9-13* 日 , 第 1407-1426 页 . ACM, 2020.

[25] Blase Ur, Patrick Gage Kelley, Saranga Komanduri, Joel Lee, Michael Maass, Michelle L. Mazurek, Timothy Passaro, Richard Shay, Timothy Vidas, Lujo Bauer, Nicolas Christin, 和 Lor- rie Faith Cranor. 你的密码怎么样？强度指示器对密码创建的影响 . 在 Tadayoshi Kohno 编的 第 *21* 届 *USENIX* 安全研讨会论文集，美国华盛顿州贝尔维尤，*2012* 年 *8* 月 *8-10* 日 , 第 65-80 页 . USENIX 协会 , 2012.

[26] Blase Ur, Fumiko Noma, Jonathan Bees, Sean M. Segreti, Richard Shay, Lujo Bauer, Nicolas Christin, 和 Lorrie Faith Cranor. " 我在末尾添加了 '!' 使其更安全 "：在实验室观察密码创建。在 Lorrie Faith Cranor, Robert Biddle, 和 Sunny Consolvo 编辑的 第 *11* 届可用隐私与安全研讨会，*SOUPS 2015*，加拿大渥太华，*2015* 年 *7* 月 *22-24* 日，第 123-140 页。USENIX 协会，2015 年。

[27] Blase Ur, Sean M. Segreti, Lujo Bauer, Nicolas Christin, Lor- rie FaithCranor, Saranga Komanduri,DaryaKurilova,MichelleL. Mazurek, William Melicher, 和 Richard Shay. 衡量建模密码可猜测性的真实世界准确性和偏差。在 Jaeyeon Jung 和 Thorsten Holz 编辑的 第 *24* 届 *USENIX* 安全研讨会，*USENIX Security 15*，美国华盛顿特区，*2015* 年 *8* 月 *12-14* 日，第 463-481 页。USENIX 协会，2015 年。

[28] 王丹 , 邹宇 , 董启 , 宋宇 , 和 黄翔 . 如何攻击和生成蜜罐词 . 在 *2022 2022 IEEE* 安全与隐私研讨会 *(SP) (SP)*, 第 489-506 页 , 美国加利福尼亚州洛斯阿尔 amos, 2022 年 5 月 . IEEE 计算机协会 .

[29] 丁王 , 程海波 , 王平 , 黄新怡 , 和 建高鹏 . 密码中的齐夫定律 . *IEEE* 信息取证与安全汇刊 , 12(11):2776-2791, 2017.

[30] 丁王 , 王平 , 何德宝 , 和 田元 . 生日、姓名和双面安全：理解中国网络用户的密码 . 在 Nadia Heninger 和 Patrick Traynor 编辑的 第 *28* 届 *USENIX* 安全研讨会 , *USENIX* 安全 *2019*, 美国加利福尼亚州圣克拉拉 , *2019* 年 *8* 月 *14-16* 日 , 第 1537-1555 页 . USENIX 协会 , 2019.

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

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

[33] Matt Weir. Practical pcfg password cracking.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

[33] Matt Weir. Practical pcfg password cracking.

[34] Matt Weir, Sudhir Aggarwal, Breno de Medeiros, and Bill Glodek. 基于概率上下文无关文法的密码破解 . 在第 *30* 届 *IEEE* 安全与隐私研讨会（ *S&P 2009* ） , *2009* 年 *5* 月 *17* 日至 *20* 日 , 加利福尼亚州奥克兰 , 美国 , 第 391-405 页 . IEEE 计算机协会 , 2009.

[35] Daniel Lowe Wheeler. zxcvbn: 低预算密码强度评估 . 在 Thorsten Holz 和 Stefan Savage 编辑的第 *25* 届 *USENIX* 安全研讨会 , *USENIX Security 16*, 德克萨斯州奥斯汀 , 美国 , *2016* 年 *8* 月 *10* 日至 *12* 日 , 第 157-173 页 . USENIX 协会 , 2016.

[36] Simon S. Woo. 我们如何创建一个绝妙的密码？ 在 Yennun Huang, Irwin King, Tie-Yan Liu 和 Maarten van Steen 编辑的 *WWW '20: 2020* 年网络大会 , 台湾台北 , *2020* 年 *4* 月 *20* 日至 *24* 日 , 第 1491-1501 页 . ACM/IW3C2, 2020.

[37] 杨晓和曾建平 . 基于 Zipf 分布动态生成密码策略 . *IEEETransInf . ForensicsSecur.*, 17:835– 848, 2022.

[38] 徐明 , 王传旺 , 余继涛 , 张俊杰 , 张凯 , 和韩伟利 . 块级密码猜测：迈向精细密码组成表示的建模 . Yongdae Kim, Jong Kim, Giovanni Vigna, 和 Elaine Shi 编 , *CCS '21: 2021 ACM SIGSAC* 计算机与通信安全会议 , 虚拟活动 , 大韩民国 , *2021* 年 *11* 月 *15 - 19* 日 . ACM, 2021.

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

张凯 是复旦大学的副教授。他在 2016 年在中国科学技术大学获得博士学位。他目前是数据分析与安全实验室的成员。他的研究领域包括数据库系统、网络系统、并行和分布式计算。

张浩东 是复旦大学的研究生。他目前是数据分析与安全实验室的成员。他的主要研究方向包括密码安全和系统安全。

张俊杰 是复旦大学的研究生。他目前是数据分析与安全实验室的成员。他的主要研究方向包括密码安全和系统安全。

Jitao Yu 是复旦大学的研究生。他目前是数据分析与安全实验室的成员。他的主要研究方向包括密码安全和系统安全。

徐明 是新加坡国立大学的研究员。她的研究兴趣在于追求混沌数据在可用安全和隐私方面的美丽收敛，将其转化为有见地的洞察。她在 2023 年获得复旦大学博士学位。她的研究兴趣主要包括可用安全和隐私，包括密码安全。

Luwei Cheng是复旦大学的研究生。他目前是数据分析与安全实验室的成员。他的主要研究方向包括密码安全和系统安全。

张圣浩 是复旦大学的研究生。他的主要研究方向包括密码安全和系统安全。他目前是数据分析与安全实验室的成员。他的主要研究方向包括密码安全和系统安全。

Weili Han 是复旦大学软件学院的正教授。他在 2003 年获得浙江大学博士学位。之后，他加入复旦大学软件学院任教。2008 年至 2009 年，他作为中国国家留学基金委和普渡大学的访问教授访问了普渡大学。他的主要研究方向包括数据系统安全、访问控制和密码安全。他是中国计算机学会（ CCF ）的杰出会员，以及 IEEE 、ACM 、 Sigsac 的会员。他在多个顶尖会议和期刊中担任 PC 成员、审稿人和副主编。