Performance of Password Guessing Enumerators Under Cracking Conditions

Mathieu Valois, Patrick Lacharme, Jean-Marie Le Bars

Performance of Password Guessing Enumerators Under Cracking Conditions

Mathieu Valois[1], Patrick Lacharme[2], and Jean-Marie Le Bars[1]
Normandie Univ, UNICAEN, ENSICAEN, CNRS, GREYC, 14000 Caen, France

[1] {mathieu.valois, jean-marie.lebars}@unicaen.fr
[2] patrick.lacharme@ensicaen.fr

Abstract. In this work, we aim to measure the impact of hash functions on the password cracking process. This brings us to measure the performance of password enumerators, how many passwords they find in a given period of time. We propose a performance measurement methodology for enumerators, which integrates the success rate and the speed of the whole password cracking process. This performance measurement required us to develop advanced techniques to solve measurement challenges that were not mentioned before. The experiments we conduct show that software-optimized enumerators like John The Ripper-Markov and the bruteforce perform well when attacking fast hash functions like SHA-1. Whereas enumerators like OMEN and PCFG-based algorithm perform the best when attacking slow hash functions like bcrypt or Argon2. Using this approach, we realize a more in-depth measurement of the enumerators performance, considering quantitatively the trade-off between the enumerator choice and the speed of the hash function. We conclude that software-optimized enumerators and tools must implement academic methods in the future.

Keywords: password · hash function · cracking conditions

1 Introduction

Passwords are for a long time one of the weakest point in digital identity security. NIST guidelines [19] tells us that "Memorized secrets SHALL be salted and hashed using a suitable one-way key derivation function". Such functions are designed to be time and memory costly to hash a password. However, the impact of the hash function on an offline password cracking process has not been studied.

Before our work, even if the time influence has once been mentioned [27], we were not able to get and understand the performance of enumerators and password cracking software depending on the hash function. For example, what is the real impact on enumerators performances when switching from SHA-1 to bcrypt?

Contributions of this work are useful to choose a good hash function knowing the threat in terms of enumerators (defender side) and to choose a good set of

enumerators knowing the hash function used to protect passwords (attacker side, password crackers, pentesters).

To measure the impact of the hash function, we aim to measure the performance of the whole offline cracking process, which is composed of the candidate generation by an enumeration algorithm, the candidate hashing and the search for its fingerprint in the targeted dataset. The performance on a period of time is defined as the number of found passwords in that period.

The most popular academic enumerators categories are Probabilistic Context-Free Grammars (PCFG) [15, 28, 30] and Markov chains models [9, 16, 18]. These models are probabilistic, as they assign probabilities to passwords to measure their strength, whereas very fast enumerators are also implemented in free software like John The Ripper (JtR) [21] and Hashcat [24]. These enumerators are optimized for their runtime performance. The faster the hash function is, the more we can test candidates in a period of time. Slowest enumerators generate better quality candidates compared to the fastest ones. It makes sense since slowest enumerators are probabilistic, and propose candidates in decreasing order of probability. There is then a trade-off between the speed of the enumerator and the quality of candidates it generates. We think that the choice of the hash function affects the performance, however it is unclear how the speed of the hash function impacts the performance of the password cracking process. For instance, until which speed of the hash function a naive strategy like bruteforce is viable?

In previous works, probabilistic enumerators were usually compared using the guess number metric [9, 30, 27]. This metric makes sense as long as enumerators aim to measure the password strength. From the moment when they are used to crack passwords, it becomes inadequate because the candidate generation speed of the enumerator must be taken into account.

Our contributions:

- We propose a methodology to measure the performance of enumerators in a password cracking context by considering the cost of processing candidates. Since this measurement can not be directly done, we then need to measure separately the success rate and the frequency of the process.
- Measuring the success rate and the frequency of the process are challenging tasks. We present advanced techniques used to conduct these measurements, since there is a sort of Heisenberg effect: measuring a phenomenon has an influence on the value itself. Such techniques have never been presented before and will certainly be useful for future works.
- Experiments on leaked databases with this performance measurement show that fast cryptographic hash functions are unsuitable for password storage, because even the most naive strategies (like bruteforce) perform very well against them. They also show that academic enumerators are very useful against slow hash functions. To make them even more performing, developers should implement them in their password cracking tools.

This paper is organized as following. First, we present the background and related works 2. Then we introduce our password cracking modelling 3. In section 4

we explain how we compute the performance using success rate and frequency measurements. Section 5 contains the results of performance measurements on two publicly leaked passwords using two different hash functions. Then, in section 6, we expose different scenarios in where our work has concrete interests. We conclude in section 7.

2 Background

2.1 Related Works

Password strength evaluation is an active research topic [6, 5, 7, 26]. Probabilistic passwords models have been studied by Ma et al. in [16] where they introduced a probability-threshold to plot graphs rather than guess-number graphs. However, it is not sufficient since it does not depend on the hash function and enumeration speed, which are required to compute the performance of passwords attacks.

Different cracking strategies have been studied by Ur et al. in [27], where they compare algorithms like PCFG, JtR, Hashcat, Markov from [16] and a strategy from password recovery professionals. They found that professionals are more efficient against complex password policies at a high number of guesses, while automatic approaches perform best at low numbers of guesses. They ended up by providing a Password Guessing Service [25] where they offer to analyze a list of plaintext passwords and return the score for each approach they support. Once again, they do not take care of the enumeration speed nor the hash function to compare enumerators.

2.2 Enumerators

Several enumerators have been proposed by academic researches, but few are currently implemented in the most well-known password cracking software. For example, John The Ripper and Hashcat include a Markov model-based enumerator that has been originally introduced in [18]. However, the remaining proposed enumerators are distributed in standalone versions, meaning they are not actually cracking passwords. Instead, they only generate candidates that might be actual passwords. These candidates should then be gathered by a password cracking software to do the rest of the process: hash them and search for a match.

Bruteforce: this is the most naive and the most known enumerator. Basically, bruteforce will output the words incrementing the characters one by one using a defined alphabet. In this paper, we run our own C implementation of the bruteforce algorithm.

Markov Models: they are probabilistic models that can be applied to words, for which the probability of a character at a given position depends on the previous characters. The software will learn probabilities on a dataset and will then output newly-created words according to these probabilities. In this paper, we consider two enumerators based on Markov models implementations: **John The Ripper's Markov mode** [1] and **OMEN** [8]. The main differences between

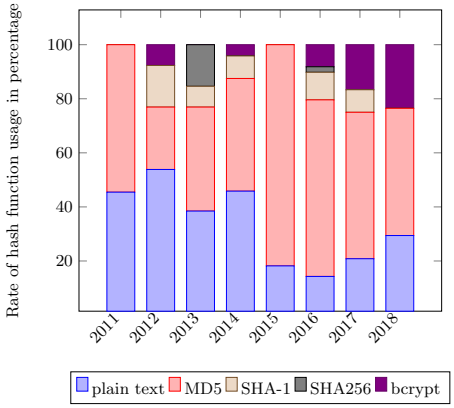

Fig. 1: Distribution of hash functions usage in password leaks since 2011. Source: haveibeenpwned.com

them is that OMEN outputs candidates in approximately decreasing probabilities order. Also, OMEN has an adaptive strategy which reorders the sets of next candidates based on the success rate of previous ones.

Probabilistic Context-Free Grammars: Weir et al. [30] are the first to use PCFGs for passcracking. Kelley et al. and Komanduri [13][15] have modified Weir's PCFGs by adding string tokenization and by assigning probabilities to terminals that have not been seen in the training dataset. The implementation used in this paper is the Weir's one and can be found on Github [29]. For easier reading, we will write "PCFG" referring as the Weir et al. algorithm.

In this paper, we make use of these four enumerators: bruteforce, JtR-Markov, OMEN and PCFG. They only generate candidates as they do not hash them or search for them in the targeted database. We don't make use of Hashcat and John The Ripper password cracking features, only the Markov-based enumerator of John The Ripper.

2.3 Hash Functions

Fast functions like MD5, SHA-1, SHA256 are often used in the leaks that happened the last 10 years. Figure 1 is an overview of the distribution of used hash functions in passwords leaks since 2011: MD5 and SHA-1 remain very used even if we observe an increase of the usage of bcrypt. These figures are not representative as it is more likely that poorly-protected services are also using fast hash functions. However, this provides a lower bound of the bcrypt usage.

Moreover, these hash functions are deterministic, which means that two same words have the same hash. It is unwanted since multiple users having the same password would have the same hash. A common counter-measure is to use a salt [17], a randomly-generated word which is appended to the passwords before hashing, and stored aside the hash value. Attackers are then forced for each

candidate to apply the salt value before hashing, slowing down the speed of the attack and making pre-computation-based attacks [11, 20] impracticable. The leaked databases on which we experiment in this paper are not salted. Nevertheless, our model still apply since we don't use pre-computed tables and usage of a salt multiply the cost of the attack by the number of users.

Password-specific hash functions like bcrypt [23], Scrypt [22] or Argon2 [4], winner of the Password Hashing Competition [3], have been proposed to replace fast hash functions. They are designed to break the usage of massive parallel processing units by requiring a big memory and time amounts to compute one hash. Moreover, they handle themselves the salt value, making easier for developers to manage and store passwords.

In this paper we consider two hash functions: the first, SHA-1, due to its wide usage according to publicly leaked passwords datasets (especially in Linkedin). SHA-1 will represent the set of cryptographic hash functions (MD5, SHA-1, SHA2, SHA-3, ...), since these functions are similar for our study. The second is bcrypt, the reference function to store passwords, with a cost factor of 10 which is the default cost factor for PHP's password_hash() function at the time of writing. Since Argon2, Scrypt and bcrypt are for our study very similar, we only consider bcrypt since it is more commonly used than Argon2 and Scrypt.

2.4 Datasets

Here are presented the publicly accessible datasets used in this paper. Note however that there exists many more datasets that can be used for such a study. The ones we used are enough to illustrate our modelling, as it does not aim to be exhaustive. Further works can be conducted on more datasets. A list of recent leaks can be found at [12].

Rockyou: The most used dataset, probably because in plaintext and easily accessible. Rockyou is a company providing services to social networks and video games. The leak happened in 2009. It contains more than 32 million of passwords (14 million unique).

LinkedIn: This leak comes from the LinkedIn social network. The hack happened in 2012. Four years later, the entire database were published, containing more than 160 million of passwords (60 million unique). The passwords were hashed with SHA-1, without salt.

It should be mentioned that the relevance of one password cracking strategy highly depends on the training and target datasets.

3 Cracking Process Modelling

3.1 Context

The cracking process, at the level of a single candidate, can be resumed in 3 steps:

(i) generate a candidate

(*ii*) hash it
(*iii*) search for a match in the target database

In practice, searching for a match in step (*iii*) is negligible, even when a fast hash function is used. Indeed, most password cracking tools use a probabilistic structure, similar to a Bloom filter with only one hash function, to store the hashed passwords list in memory. We assume that step (*iii*) takes no time. Then according to the hash function in step (*ii*), the bottleneck of the process is either step (*i*) or (*ii*).

3.2 Formalization of the Performance

Even if in practice we compute a discrete version of this password cracking process, it remains a continuous process since the time is continuous, which explains the usage of integrals. Let first consider the performance at the level of a single candidate. For any $i \in \mathbb{N}$, let t^i be the instant where we finish to process the i^{th} candidate of the enumeration, g^i the gain of this candidate, *i.e.* its number of occurrences in the dataset D and c^i the time to process it (steps (*i*), (*ii*), (*iii*)), *i.e.* $c^i = t^i - t^{i-1}$. The performance will be $P(t^{i-1}, t^i) = g^i$. Thus

$$P(t^{i-1}, t^i) = c^i \ \frac{g^i}{c^i} = \int_{t^{i-1}}^{t^i} \frac{g(t)}{c(t)} dt,$$

where $c(t) = c^i$ and $g(t) = g^i$.

Note 1. this model includes the parallelization of the process, since $g(t)$ and $c(t)$ could be measured on multiple cores.

Let t_1 and t_2 be any instants such that $t_1 < t_2$. The performance in the period $]t_1, t_2]$ will be $P(t_1, t_2) = \int_{t_1}^{t_2} \frac{g(t)}{c(t)} dt$, where $c(t) = c^i$ and $g(t) = g^i$, for any $t^{i-1} < t \leq t^i$.

The frequency is by definition the number of processed candidates between t_1 and t_2. We get the formula $F(t_1, t_2) = \int_{t_1}^{t_2} \frac{1}{c(t)} dt$. We also define the success rate as the ratio between the number of passwords found in the period $]t_1, t_2]$ and the number of processed candidates in the same period: $S(t_1, t_2) = \frac{P(t_1,t_2)}{F(t_1,t_2)}$.

Comparison between enumerators performance We show here how to compare two enumerators depending on the speed of the used hash function. For each candidate i, $c^i = c_g^i + c_h^i + c_d^i$ where each term corresponds respectively to one step of the cracking process (generate, hash and search). We show in our experiments that c_d^i is negligible for any enumerator, then $c^i \approx c_g^i + c_h^i$. Let E_1 and E_2 be two enumerators we want to compare. We have two cases:

- **case a) a slow hash function is used**. Then $c^i = c_g^i + c_h^i \approx c_h^i$ for any i and $F_1(t_1, t_2) \approx F_2(t_1, t_2)$. $P_1(t_1, t_2) > P_2(t_1, t_2)$ when $S_1(t_1, t_2) > S_2(t_1, t_2)$, the enumerator E_1 outperforms E_2 when it has a better success rate.

– **case b) a fast hash function is used**. Then $c^i = c_g^i + c_h^i \approx c_g^i$ for any i and $P_1(t_1, t_2) > P_2(t_1, t_2)$ when $\frac{S_1(t_1,t_2)}{c_{g,1}} > \frac{S_2(t_1,t_2)}{c_{g,2}}$. The enumerator with the best ratio between the success rate and the enumerator speed has the best performance. We have a trade-off between success rate and speed.

Previous works consider the success rate from the beginning corresponds to the case a), we have a slow hash function. Indeed in this case, the enumerator speed is not very important and the best performance is obtained with the best success rate. In case b), when a fast hash function is used, the enumerator E_1 should have a best success rate than the enumerator E_2 but a worse performance if E_2 is a faster enumerator. For instance, if E_2 has twice better success rate than E_1 but a hundred times slower enumerator, the performance of E_2 is fifty times better than E_1, in a same period the enumerator E_2 finds fifty times more passwords than E_2. In that case, measuring only the success rate is clearly insufficient.

3.3 Estimating the Performance

In practice, we can't measure directly the performance, since it would be very expansive in time and uninteresting to compute the gain $g(t)$ for every t. One solution is to estimate the frequency and the success rate in small periods and derive an estimation of the performance. We will perform these measures for each interval of one second: $P(j, j+1) = F(j, j+1)\ S(j, j+1)$.

Firstly, as we already stated, the measurement of $c_g(t)$ has an impact on its value. We want then to have the less measurements possible while keeping a good enough accuracy. $c_g(t)$ does not vary much between j and $j+1$, then we suppose $c_g(t)$ to be constant in that period. For that, we note c_g its value, and take the mean of $c_g(t)$ as its value. Secondly, we can consider c_h to be constant given a hash function, because passwords size is almost always smaller than the input size of the compression function.

If we note $c = c_g + c_h$ in the interval $]j, j+1]$, then we have $F(j, j+1) = \frac{1}{c}$ the frequency of the enumerator during that period.

Let now consider a period of k seconds $]t_1, t_2 = t_1 + k]$, we have

$$P(t_1, t_2) = \sum_{l=0}^{k-1} P(t_1 + l, t_2 + l + 1)$$

as it is computed in the previous researches with the guess number comparisons, called the "Cumulative Distribution Function (CDF)".

4 How to Measure Performances of the Cracking Process

While the time cost to generate a candidate has too briefly been highlighted in a previous work [27], the computation of the performance has never been considered. We show that S and F must be computed separately. An estimation

Table 1: N value for measuring time for each enumerator

Enumerator	bruteforce	JtR-Markov	OMEN	PCFG
N	10^9	4×10^6	10^5	10^4

of these two values is enough. Furthermore, it makes possible the study of them separately, which is a work that can be done in a foreseeable future.

Nevertheless, as mentioned in [27], due to the fact that some enumerators are very fast, computing $S(j, j+1)$ and $F(j, j+1)$ in practice is very challenging. The measurements that have been performed are now presented. First of all, none of the enumerators implementations provide the measurement of the time between candidates. OMEN embeds the success rate measurement. For the remaining ones, we need to make our own measurements using different techniques depending on the implementation. Since time is not considered in the success rate measurement, we may use different techniques depending on the enumerator.

Number of candidates in a period. The 3 steps ((i), (ii) and (iii)) must be taken into account when measuring the number of candidates. However, steps (ii) and (iii) take constant times given a hash function, we then only need to analyze the generation step. Measuring times between each candidates until reaching one second is a very costly process. We instead estimate this number of candidates by measuring the time to generate a fixed number N of candidates. Then we compute the time spent to hash these candidates by multiplying the number of candidates with the time to hash one candidate. For example, in average, PCFG generates $\approx 8 \times 10^4$ candidates per second, OMEN 10^6, JtR-Markov 2×10^7 and bruteforce 1.6×10^9 (note however that these times are not constant during enumeration, hence are presented here to have a glance on enumerators speed). In our experiments, we chose N such that times required to generate N candidates are about 0.1 second (at beginning) to have both a good estimation of $C(t)$ and an acceptable number of floating point numbers to store. Our values of N for each enumerator can be found in Table 1.

Success rate measurement. To measure the success rate, we want to have for each candidate, its rank and its gain (how many times it appears in the dataset). There are two ways of measuring the success rate: by running the enumerator and counting occurrences in the dataset, or backwardly by computing for each password of the dataset, its rank in the enumeration if the enumerator has an index function. The former can be applied to every enumerators. However, it requires to run them, which is a long process, especially if we want to benchmark them for a long time. The latter however is doable only for few enumerators: bruteforce and JtR-Markov since the rank of a word is predictable and easily computable (index function). Since the time is not considered, using different techniques to measure the success rate of enumerators is not an issue. Once we get all the gains, we can compute the success rate for any period of time using the number of candidates for that period. For that, we search the

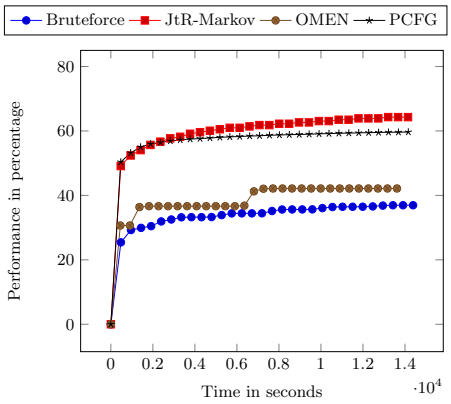

Fig. 2: LinkedIn dataset using SHA-1

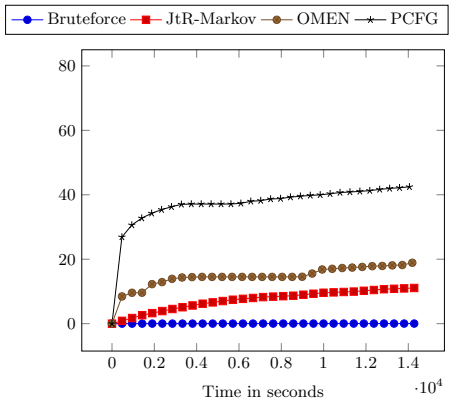

Fig. 3: LinkedIn dataset using bcrypt with a cost factor of 10

rank of the first generated candidate of the given period, and compute the sum of gains corresponding to candidates of this period. The rank can be computed by summing the number of candidates for all previous periods of one second.

Performance Computation. Once we get the success rate and the number of candidates for every period, the computation of the performance is straight forward $P = F \times S$.

5 Experimental Results

In this section we present the experiments of the performance computation for four enumerators, the bruteforce, John The Ripper-Markov mode, OMEN and the PCFG-based enumerator, over the datasets LinkedIn and Rockyou, and using two hash functions, SHA-1 and bcrypt (cost 10). To be able to compare plots with previous researches and across datasets, rather than plotting the performance, we plot the percentage of cracked passwords from the beginning. We can then compare how they perform between each other over time and across datasets. We still took [10] for hash functions benchmarks. For SHA-1, $1/c_h \approx 12.5 \times 10^9$, and for bcrypt with a cost of 10, which is the default cost factor for PHP's password_hash() function at the time of writing, $1/c_h \approx 700$. Note that the benchmarks in Hashcat uses a cost 5 bcrypt, which means 2^5 rounds of the internal key-derivation function. Thus, c_h for a cost-10 bcrypt is 2^5 times smaller, giving $23 \times 10^3/2^5 \approx 700$.

On Figure 2, showing how enumerator perform over LinkedIn with the SHA-1 function, JtR-Markov and PCFG cracked about 60% of passwords after four hours, while OMEN cracked around 43% and the bruteforce around 36%. Bruteforce, even though it is the most naive method, still has good results. PCFG is surprisingly good since it exploits the high number of passwords sharing the same grammatical structure in this dataset: 37% of passwords share the top

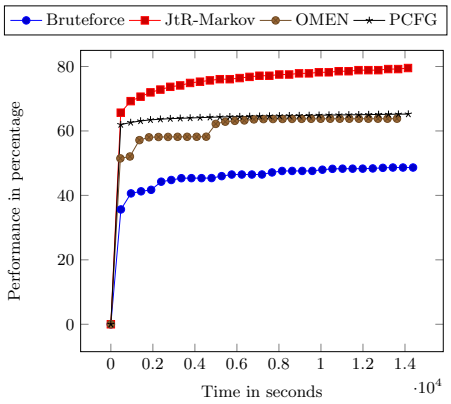

Fig. 4: Rockyou dataset using SHA-1

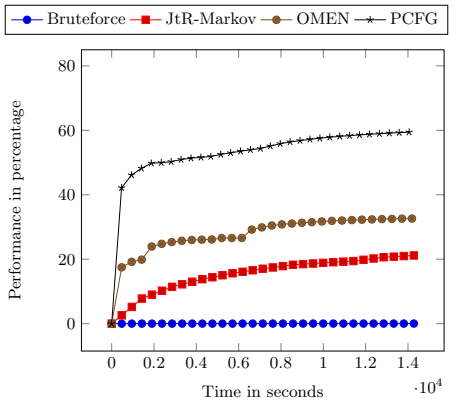

Fig. 5: Rockyou dataset using bcrypt with a cost factor of 10

five most frequent structures. However using bcrypt (Fig. 3), enumerators performances are completely disrupted. Because of the poor quality candidates of the bruteforce, the whole process spends its time to hash candidates which give mediocre success rates. After four hours, bruteforce cracked less than 1% of the dataset, while PCFG still performed great by cracking 43% of passwords. OMEN is better placed than before, cracking more than 18% of the dataset. Finally, JtR-Markov is worse than OMEN, finding only 11% of passwords after four hours. On Rockyou using SHA-1 (Fig. 4), JtR-Markov cracked 15% passwords more than on LinkedIn after four hours, while OMEN cracked 23% more than on LinkedIn, and the bruteforce 12% more. However, PCFG perform similar than on LinkedIn using SHA-1. On Rockyou using bcrypt (Fig. 5), results are similar than with LinkedIn using bcrypt: even though enumerators cracked not the same number of passwords, their performances are in the same order. The "steps" of the OMEN curves on all figures is due to the fact that it generates sets of candidates of same length. When OMEN switches the length of the candidates, it suddenly cracks many more passwords.

These four graphics highlight the impact of the hash function concerning the performance of the cracking process. They confirm our hypothesis that hacker enumerators are well-performing using fast hash functions while academic enumerators perform better using slow hash functions. One special mention to the PCFG-based enumerator which performs quite well on LinkedIn using SHA-1. Finally, experiments on each dataset can be easily adapted to other hash functions or to other parameters, like the cost of bcrypt. We could also choose the algorithm depending on the used hash function from the beginning. With the rise of memory-hard functions usage like bcrypt or Argon2, it will be even more interesting to have software-optimized academic enumerators implemented in password cracking tools.

It is important to note that the performance of enumerators highly depends on the dataset, and that OMEN becomes better than PCFG when attacking more complex datasets where passwords are longer and having more complex structures, like when keeping only strong passwords (longer than 8 characters, include all four characters classes). We also observed that enumerators perform differently on particular password composition policies, like the basic, complex, longbasic and longcomplex as defined in [27]. Results suggest that the beginning of the cracking session is decisive enough to determine which enumerator will perform the best on the targeted dataset.

6 Impacts of our Contributions

We present here the usages that can be made of our contribution and results, for different actors of the computer security community.

Password guessing researchers. Imagine a scenario where you are a password guessing researcher who wants to build a new enumerator. Since you aim to attack passwords, you have to take care of the time cost of the enumeration algorithms you build. Concretely, you have to measure both their success rate and their frequencies in password cracking conditions, as we did in section 4.

Then, you have a concrete proof that your enumerators are worth implementing them since you have measured their performances as if they were integrated in password cracking software. You could bring that proof in your future publications to encourage password cracking tools developers to implement your solution.

Furthermore, since you independently measured success rate and frequencies during a cracking session, you can independently analyze the behaviors of the success rate or the frequency during the enumeration. Thanks to that, you have a clearer understanding on your enumerator performance.

Then you are able to propose different enumerators settings regarding the attacked dataset. If your measurements show that a set of parameters provides good results against a given dataset, you can be pretty confident that this set of parameters will also provides good results against a similar dataset (for example where the password composition policiy is similar). Moreover, you can implement a strategy in your enumerator that adapt the enumerator settings depending on the found passwords. For example, if you found a lot of passwords of length 6, your enumerator can for a while only generate passwords of length 6.

Password cracking tools developers. Imagine once again a scenario where you are a developper of a password cracking tool and want to implement and optimize better enumerators in your tool. Using our comparison methodology, you are able to compare existing enumerators as if they were integrated in your software. For example, on both attacked datasets in section 5, PCFG is not really impacted when using bcrypt instead of SHA-1. Implementing it directly in your cracking tool would make it even more efficient in such contexts.

Therefore, based on the hash function used in the targeted dataset, your tool can select different enumerators at the beginning of the cracking session. You

can select those which are known to be more efficient against fast hash functions when you detect such functions, and similarly for slow hash functions. That way, the first steps of a cracking session can be run without user interaction. You can then crack a non-negligible part of weakest passwords automatically. Moreover, you can implement an adaptive strategy to select enumerators during the same single cracking session: based on previous performances of the running enumerator, your tool can change the enumerator for a one that is more likely to be efficient. For example, if a lot of found passwords share the same base structure, your tool benefits to switch to a PCFG-based algorithm since its performance is high on such passwords (provided that this algorithm performs well on the used hash function).

Security community & CISO. Imagine a scenario where your are a CISO, system administrator of a company or an university, and that you want to improve the security of your infrastructure and users. The results of our research emphasize on the importance of a good hash function to protect passwords. Cryptography-oriented hash functions are unsuitable for password hashing since they allow to most of attacks to be very efficient even if their success rate is low. Therefore, it is essential for you to protect passwords using a dedicated hash function that has been designed for it, like bcrypt or Argon2d. Nevertheless speed performance of the hash function also depends on the hardware [14]. Dedicated hardware (ASIC or FPGA) focuses particularly on hash functions used in cryptocurrencies mining (typically SHA-256 but also Scrypt) [2].

Nonetheless, the results also show that a slow hash function is not enough to offer a very good protection for passwords. For example, PCFG still perform well against bcrypt-protected datasets. The only remaining protection to such attacks is by ensuring a good password strength before registering it in the database. That is why it is important for you to provide, when users register, a satisfying password composition policy that aims to increase the spread of passwords in their universe. Our work can also be used as a leverage to recommend or force the usage of slow hash functions and the usage of password strength meters in organizations services.

7 Conclusion

Our study proves the importance of the speed of the enumerator and the hash function in the performance measurement of a password cracking process. Thanks to that, it becomes possible to evaluate how efficient a slow memory-hard function is against the different enumeration strategies of the literature.

Even if we observe an increase of the bcrypt and Argon2d usage, recent passwords database leaks still confirm the high usage of cryptographic hash functions like SHA-1 and its siblings. We recommend the usage of dedicated slow and memory-hard hash functions to protect passwords.

We bring the technical challenges out when computing the performance of enumerators in password cracking context. Firstly, we highlight the impossibility to compute both the number of found passwords and the time to generate

the required candidates without altering their values. Even if we only measure the latter it remains a complicated task. Then we provided a methodology to estimate these values accurately enough.

When the enumerator has an index function, as in [18], we have a big advantage since it becomes possible to compute the success rate of the enumerator without running it. However it has not been considered when recent enumerators have been designed. In the other case, we need to run the algorithm and measure its success rate along the enumeration.

Our experiments over two publicly leaked passwords lists show unexpected results. First, we observe that bruteforce is still useful against fast hash functions. Secondly, the PCFG-based algorithm is nearly as good as JtR-Markov against LinkedIn using SHA-1, meaning that it would be better than JtR-Markov if it was optimized. In OMEN paper [9], authors showed that it was better than PCFG on the Rockyou dataset. However we show that considering the cost of generating candidates, PCFG becomes better in all presented experiments.

We showed that JtR-Markov is really relevant when using a fast hash function like SHA-1, while probabilistic enumerators like OMEN and PCFG-based algorithms perform better than others using slow hash function like bcrypt.

If PCFG and OMEN were optimized for password cracking, results would change only against fast hash functions, where OMEN, PCFG and JtR-Markov would be more distinguishable. We encourage Hashcat and John The Ripper developers to implement such algorithms in further versions of their tools. Hashcat developers have already took a step in that way since the version *v5.0.0* by introducing the feature "slow candidates" which aims to facilitate the integration of slow enumerators proposed by academics.

More generally, the security community, especially in the field of password protection, lack of researches on password enumerator performances. Nowadays, the usage of fast hash functions remains too high. However, slow hash functions are more likely to be used in the near future. Therefore, our study is relevant and should be extended with future works that take into account the password cracking context in order to be closer to the attacker environment.

Thanks

We would like to thank the Région Normandie for supporting this research.

References

1. John the ripper implementation. https://github.com/magnumripper/JohnTheRipper
2. Mining hardware comparison. https://en.bitcoin.it/wiki/Mining_hardware_comparison
3. Aumasson, J.P.: Password hashing competition. https://password-hashing.net/
4. Biryukov, A., Dinu, D., Khovratovich, D.: Argon2: new generation of memory-hard functions for password hashing and other applications. In: Security and Privacy (EuroS&P), 2016 IEEE European Symposium on. pp. 292–302. IEEE (2016)
5. de Carné de Carnavalet, X., Mannan, M.: A large-scale evaluation of high-impact password strength meters. ACM Transactions on Information and System Security (2015)

6. Delahaye, J.P., Zenil, H.: Numerical evaluation of algorithmic complexity for short strings: A glance into the innermost structure of randomness. No. 1, Elsevier (2012)
7. Dell'Amico, M., Filippone, M.: Monte carlo strength evaluation: Fast and reliable password checking. In: Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security. pp. 158–169. ACM (2015)
8. Dürmuth, M., Angelstorf, F., Castelluccia, C., Perito, D., Chaabane, A.: OMEN implementation. https://github.com/RUB-SysSec/OMEN
9. Dürmuth, M., Angelstorf, F., Castelluccia, C., Perito, D., Chaabane, A.: OMEN: Faster password guessing using an Ordered Markov ENumerator. In: International Symposium on Engineering Secure Software and Systems. Springer (2015)
10. Gosney, J.: 8x Nvidia GTX 1080 Hashcat benchmarks. https://gist.github.com/epixoip/ace60d09981be09544fdd35005051505
11. Hellman, M.: A cryptanalytic time-memory trade-off. IEEE transactions on Information Theory **26**(4), 401–406 (1980)
12. Hunt, T.: Have I been pwned. https://haveibeenpwned.com (2017)
13. Kelley, P.G., Komanduri, S., Mazurek, M.L., Shay, R., Vidas, T., Bauer, L., Christin, N., Cranor, L.F., Lopez, J.: Guess again (and again and again): Measuring password strength by simulating password-cracking algorithms. In: 2012 IEEE Symposium on Security and Privacy. pp. 523–537. IEEE (2012)
14. Khalil, G.: Password security - thirty-five years later (2014)
15. Komanduri, S.: Modeling The Adversary To Evaluate Password Strength With Limited Samples. Ph.D. thesis, Microsoft Research (2016)
16. Ma, J., Yang, W., Luo, M., Li, N.: A study of probabilistic password models. In: 2014 IEEE Symposium on Security and Privacy. pp. 689–704. IEEE (2014)
17. Morris, R., Thompson, K.: Password security: A case history. vol. 22. ACM (1979)
18. Narayanan, A., Shmatikov, V.: Fast dictionary attacks on passwords using time-space tradeoff. In: ACM conference on Computer and communications security (CCS). pp. 364–372. ACM (2005)
19. NIST: Nist special publication 800-63b. https://pages.nist.gov/800-63-3/
20. Oechslin, P.: Making a faster cryptanalytic time-memory trade-off. In: Annual International Cryptology Conference (CRYPTO). pp. 617–630. Springer (2003)
21. OpenWall: John the ripper. http://www.openwall.com/john (2017)
22. Percival, C.: Stronger key derivation via sequential memory-hard functions (2009)
23. Provos, N., Mazieres, D.: A future-adaptable password scheme. In: USENIX Annual Technical Conference, FREENIX Track. pp. 81–91 (1999)
24. Steube, J.: Hashcat implementation. https://github.com/hashcat/hashcat
25. Ur, B.: Password guessability service. https://pgs.ece.cmu.edu/ (2015)
26. Ur, B., Alfieri, F., Aung, M., Bauer, L., Christin, N., Colnago, J., Cranor, L.F., Dixon, H., Naeini, P.E., Habib, H., et al.: Design and evaluation of a data-driven password meter. In: Proc. CHI (2017)
27. Ur, B., Segreti, S.M., Bauer, L., Christin, N., Cranor, L.F., Komanduri, S., Kurilova, D., Mazurek, M.L., Melicher, W., Shay, R.: Measuring real-world accuracies and biases in modeling password guessability. In: 24th USENIX Security Symposium (USENIX Security 15). pp. 463–481 (2015)
28. Weir, C.M.: Using probabilistic techniques to aid in password cracking attacks. In: ACM conference on Computer and communications security (CCS) (2010)
29. Weir, M.: PCFG implementation. https://github.com/lakiw/pcfg_cracker
30. Weir, M., Aggarwal, S., De Medeiros, B., Glodek, B.: Password cracking using probabilistic context-free grammars. In: 2009 30th IEEE Symposium on Security and Privacy. pp. 391–405. IEEE (2009)