Cost-Asymmetric Memory Hard Password Hashing

Wenjie Bai, Jeremiah Blocki(✉), and Mohammad Hassan Ameri

Purdue University, West Lafayette, IN 47907, USA
{bai104,jblocki,mameriek}@purdue.edu

Abstract. In the past decade billions of user passwords have been exposed to the dangerous threat of offline password cracking attacks. An offline attacker who has stolen the cryptographic hash of a user's password can check as many password guesses as s/he likes limited only by the resources that s/he is willing to invest to crack the password. Pepper and key-stretching are two techniques that have been proposed to deter an offline attacker by increasing guessing costs. Pepper ensures that the cost of rejecting an incorrect password guess is higher than the (expected) cost of verifying a correct password guess. This is useful because most of the offline attacker's guesses will be incorrect. Unfortunately, as we observe the traditional peppering defense seems to be incompatible with modern memory hard key-stretching algorithms such as Argon2 or Scrypt. We introduce an alternative to pepper which we call Cost-Asymmetric Memory Hard Password Authentication which benefits from the same cost-asymmetry as the classical peppering defense i.e., the cost of rejecting an incorrect password guess is larger than the expected cost to authenticate a correct password guess. When configured properly we prove that our mechanism can only reduce the percentage of user passwords that are cracked by a rational offline attacker whose goal is to maximize (expected) profit i.e., the total value of cracked passwords minus the total guessing costs. We evaluate the effectiveness of our mechanism on empirical password datasets against a rational offline attacker. Our empirical analysis shows that our mechanism can significantly reduce the percentage of user passwords that are cracked by a rational attacker by up to 10%.

Keywords: Memory Hard Functions · Password Authentication · Stackelberg Game

1 Introduction

In the past decade data-breaches have exposed billions of user passwords to the dangerous threat of offline password cracking. An offline attacker has stolen the cryptographic hash $h_u = H(pw_u, salt_u)$ of a target user (u) and can validate as many password guesses as s/he likes without getting locked out i.e., given h_u

© The Author(s), under exclusive license to Springer Nature Switzerland AG 2022
C. Galdi and S. Jarecki (Eds.): SCN 2022, LNCS 13409, pp. 21–44, 2022.
https://doi.org/10.1007/978-3-031-14791-3_2

and $salt_u$[1] the attacker can check if $pw_u = pw'$ by computing $h' = H(pw', salt_u)$ and comparing the hash value with h_u. Despite all of the security problems text passwords remain entrenched as the dominant form of authentication online and are unlikely to be replaced in the near future [17]. Thus, it is imperitive to develop tools to deter offline attackers.

An offline attacker is limited only by the resources s/he is willing to invest in cracking the password and a rational attacker will fix a guessing budget to optimally balance guessing costs with the expected value of the cracked passwords. Key-Stretching functions intentionally increase the cost of the hash function H to ensure that an offline attack is as expensive as possible. Hash iteration is a simple technique to increase guessing costs i.e., instead of storing $(u, salt_u, h_u = H(pw_u, salt_u))$ the authentication server would store $(u, salt_u, h_u = H^t(pw_u, salt_u))$ where $H^{i+1}(x) := H(H^i(x))$ and $H^1(x) := H(x)$. Hash iteration is the traditional key-stretching method which is used by password hashing algorithms such as PBKDF2 [27] and BCRYPT [36]. Intuitively, the cost of evaluating a function like PBKDF2 or BCRYPT scales linearly with the hash-iteration parameter t which, in turn, is directly correlated with authentication delay. Cryptocurrencies have hastened the development of Application Specific Integrated Circuits (ASICs) to rapidly evaluate cryptographic hash functions such as SHA2 and SHA3 since mining often involves repeated evaluation of a hash function $H(\cdot)$. In theory an offline attacker could use ASICs to substantially reduce the cost of checking password guesses. In fact, Blocki et al. [13] argued that functions like BCRYPT or PBKDF2 cannot provide adequate protection against an offline attacker without introducing an unacceptable authentication delay e.g., 2 min.

Memory-Hard Functions (MHFs) [35] have been introduced to address the short-comings of hash-iteration based key-stretching algorithms like BCRYPT and PBKDF2. Candidate MHFs include SCRYPT [35], Argon2 (which was declared as the winner of Password Hashing Competition [2] in 2015) and DRSample [5]. Intuitively, a password hash function is memory hard if any algorithm evaluating this function must lock up large quantities of memory for the duration of computation. One advantage of this approach is that RAM is an expensive resource even on an ASIC leading to egalitarian costs i.e., the attacker cannot substantially reduce the cost of evaluating the hash function using customized hardware. The second advantage is that the Area-Time cost associated with a memory hard function can scale quadratically in the running time parameter t. Intuitively, the honest party can evaluate the hash function $\mathsf{MHF}(\cdot; t)$ in time t, while any attacker evaluating the function must lock up t blocks of memory for t steps i.e., the Area-Time cost is t^2. The running time parameter t is

[1] The salt value protects against pre-computation attacks such as rainbow tables and ensures that the attacker must crack each individual password separately. For example, even if Alice and Bob select the same password $pw_A = pw_B$ their password hashes will almost certainly be different i.e., $h_A = H(pw_A, salt_A) \neq H(pw_B, salt_B) = h_B$ due to the different choice of values and collision resistance of the cryptographic hash function H.

constrained by user patience as we wish to avoid introducing an unacceptably long delay while the honest authentication server evaluates the password hash function during user authentication. Thus, quadratic cost scaling is desireable as it allows an authentication server to increase password guessing costs rapidly without necessarily introducing an unacceptable authentication delay.

Peppering [32] is an alternative defense against an offline password attacker. Intuitively, the idea is for a server to store $(u, salt_u, h_u = H(pw_u, salt_u, x_u))$. Unlike the random salt value $salt_u$, the random pepper value $x_u \in [1, x_{max}]$ is *not* stored on the authentication server. Thus, to verify a password guess pw' the authentication server must compute $h_1 = H(pw', salt_u, 1), \ldots, h_{x_{max}} = H(pw', salt_u, x_{max})$. If $pw' = pw_u$ then we will have $h_{x_u} = h_u$ and authentication will succeed. On the other hand, if $pw' \neq pw_u$ then we will have $h_i \neq h_u$ for all $i \leq x_{max}$ and authentication will fail. In the first case (correct login) the authentication server will not need to compute $h_i = H(pw', salt_u, i)$ for any $i > x_u$, while in the second case (incorrect guess) the authentication server will need to evaluate h_i for every $i \leq x_{max}$. Thus, the expected cost to verify a correct password guess is lower than the cost of rejecting an incorrect password guess. This can be a desirable property as a password attacker will spend most of his time eliminating incorrect password guesses, while most of the login attempts sent to the authentication server will be correct.

A natural question is whether or not we can combine peppering with Memory Hard Functions to obtain both benefits: quadratic cost scaling and cost-asymmetry.

Question 1. Can we design a password authentication mechanism that incorporates **cost-asymmetry** into ASIC resistant Memory Hard Functions while having the benefits of **fully quadratic cost scaling** under the the constraints of authentication delay and expected workload?

Naive Approach: At first glance it seems trivial to integrate pepper with a memory hard function $\mathsf{MHF}(\cdot)$ e.g., when a new user u registers with password pw_u we can simply pick our random pepper $x_u \in [1, x_{max}]$, salt $salt_u$, compute $h_u = \mathsf{MHF}(pw_u, salt_u, x_u; t)$ and store the tuple $(u, salt_u, h_u)$. Unfortunately, the solution above is overly simplistic. How should the parameters be set? We first observe that the authentication delay for our above solution can be as large as $t \cdot x_{max}$ since we may need to compute $\mathsf{MHF}(pw, salt_u, x; t)$ for every value of $x \in [1, x_{max}]$ and this computation must be carried out sequentially to reap the cost-asymmetry benefits of pepper. Similarly, the Area-Time cost for the attacker to evaluate $\mathsf{MHF}(pw, salt_u, x; t)$ for every value of $x \in [1, x_{max}]$ would scale with $t^2 \cdot x_{max}$. This may seem reasonable at first glance, but what if the authentication server had not used pepper and instead stored $h_u = \mathsf{MHF}(pw_u, salt_u; t \cdot x_{max})$ using the running time parameter $t' = t \cdot x_{max}$? In this case the authentication delay is identical, but the attacker's Area-Time cost would be $t'^2 = t^2 \cdot x_{max}^2$—an increase of x_{max} in comparison to the naive solution. Thus, the naive approach to integrate pepper and memory hard functions loses much of the benefit of quadratic scaling.

Halting Puzzles: Boyen [18] introduced the notion of a halting puzzle where the "pepper" value is replaced with a random running time parameter. In particular, when a new user u registers with a password pw_u we can pick our random running time parameter $t_u \in [1, t_{max}]$ along with $salt_u$ and store $(u, salt_u, h_u)$ where $h_u = \mathsf{MHF}(pw_u, salt_u; t_u)$. Given a password guess pw' the authentication server will locate $salt_u, h_u$ and accept if and only if $h_u = \mathsf{MHF}(pw', salt_u; t)$ for some $t \in [1, t_{max}]$. All memory hard functions $\mathsf{MHF}(w; t)$ we are aware of generate a stream of data-labels $L_1, \ldots, L_t$ where $L_i = \mathsf{MHF}(w; i)$ and L_{i+1} can be computed quickly once the prior labels $L_1, \ldots, L_i$ are all stored in memory e.g., we might have $L_{i+1} = H(L_{i-1}, L_j)$ where $j < i-1$ and H is the underlying cryptographic hash function. Thus, whenever the user attempts to login with password pw'_u the honest server can simply start computing $\mathsf{MHF}(pw'_u, salt_u; t_{max})$ to generate a stream of labels $L'_1, L'_2, \ldots$ and immediately accept if the server finds some label $i \leq t$ which matches the password hash i.e., $L_i = h_u$. Observe that whenever the user enters the correct password $pw'_u = pw_u$ the honest authentication server will be able to halt early after just $t_u \leq t_{max}$ iterations. By constrast, the only way to definitely reject an incorrect password pw'_u is to finish computing $\mathsf{MHF}(pw'_u, salt_u; i)$. The authentication delay is at most t_{max} and it seems like the attacker's area-time cost will scale quadratically i.e., t^2_{max}. Thus, the solution ostensibly seems to benefit from quadratic cost scaling and cost-asymmetry.

However, we observe that an attacker might not choose to compute the entire function $\mathsf{MHF}(pw', salt_u; t)$ for each password guess. For example, suppose that, as proposed in [18], the running time parameter t_u is selected uniformly at random in the range $[1, t_{max}]$, but for each password guess pw' in the attacker's dictionary the attacker only computes $\mathsf{MHF}(pw', salt_u; t_{max}/3)$ to compare the stolen hash h_u with the first $t_{max}/3$ labels. The attacker's area-time cost per password guess $(t^2_{max}/9)$ would decrease by a factor of 9, but the attacker's success rate only diminishes by a factor of $1/3$—the probability that $t_u \in [1, t_{max}/3]$. Motivated by this observation there are several natural questions to ask. First, can we model how a rational offline attacker would adapt his approach to deal with halting puzzles? Second, if t_u is picked uniformly at random is it possible that the solution could have an adverse impact i.e., could we unintentionally *increase* the number of passwords cracked by a rational (profit-maximizing) attacker? Finally, can we find the optimal distribution over t_u which minimizes the success rate of a rational offline attacker subject to constraints on (amortized) server workload and maximum authentication delay.

1.1 Our Contributions

We introduce Cost-Asymmetric Memory Hard Password Hashing, an extention of Boyen's halting puzzles which can only *decrease* the number of passwords cracked by a rational password cracking attacker. Our key modification is to introduce cost-even breakpoints as random running time parameters i.e., we fix m values $t_1 \leq \ldots \leq t_m = t$ such that $t^2_m = t^2_i(m/i)$ for all $1 \leq i < m$. Now instead of selecting x_u randomly in the range $[1, t]$ (time-even breakpoints) we pick $x_u \in \{t_1, \ldots, t_m\}$. We can either select $x_u \in \{t_1, \ldots, t_m\}$ uniformly

at random or, if desired, we can optimize the distribution in an attempt to minimize the expected number of passwords that the adversary breaks. Then the authentication server computes $h_u = \mathsf{MHF}(pw_u, salt_u; x_u)$ and store the tuple $(u, salt_u, h_u)$ as the record for user u.

We adapt the Stackelberg game theoretic framework of Blocki and Datta [12] to model the behavior of a rational password cracking attacker when the authentication server uses Cost-Asymmetric Memory Hard Password Hashing. In this model the attacker obtains a reward v for every cracked password and will choose a strategy which maximizes its expected utility—expected reward minus expected guessing costs. One of the main challenges in our setting is that the attacker's action space is exponential in the size of the support of the password distribution. For each password pw the attacker can chose to ignore the password, partially check the password or completely check the password. We design efficient algorithms to find a locally optimal strategy for the attacker and identify conditions under which the strategy is also a global optimum (these conditions are satisfied in almost all of our empirical experiments). We can then use black-box optimization to search for a distribution over x_u which minimizes the number of passwords cracked by our utility maximizing attacker.

When $x_u \in \{t_1, \ldots, t_m\}$ is selected uniformly at random we prove that cost-even breakpoints will only reduce the number of passwords cracked by a rational attacker. By contrast, we provide examples where time-even breakpoints increases the number of passwords that are cracked—some of these examples are based on empirical password distributions.

We empirically evaluate the effectiveness of our mechanism with 8 large password datasets. Our analysis shows that we can reduce the fraction of cracked passwords by up to 10% by adopting cost-asymmetric memory hard password hashing with cost-even breakpoints sampled from uniform distribution. In addition, our analysis demonstrates that the benefit of optimizing the distribution over x_u is marginal. Optimizing the distribution over the breakpoints $t_1, \ldots, t_m$ requires us to accurately estimate many key parameters such as the attacker's value v for cracked passwords and the probability of each password in the user password distribution. If our estimates are inaccurate then we could unintentionally increase the number of cracked passwords. Thus, we recommend instantiating Cost-Asymmetric Memory Hard Password Hashing with the uniform distribution over our cost-even breakpoints $t_1, \ldots, t_m$ as a *prior independent* password authentication mechanism.

1.2 Related Work

Trade-off between usability and security lie in the core of mechanism design of password authentication. Users tend to pick low-entropy passwords [16], leaving their accounts insecure. Convincing them to select stronger passwords is a difficult task [19,29]. Password strength meters are commonly embedded in website in the hope that users would select stronger passwords after the strength of their original passwords being displayed. However, it is found that users are often not persuaded by the suggestion of password strength meters [20,39]. In order to

encourge users to pick high-entropy passwords some sites mandate users to follow stringent guidelines when users create their passwords. However, it has been shown that these policies can incur undesirable usability costs [3,24,26,37], and in some cases can even lead to users selecting weaker passwords [14,29].

Password offline attacks have been a concern since the Unix system was devised [34]. Various approaches are developed to expedite the cracking process by the adversary or model password guessability by the hoesty party. Tools like Hashcat [1] and John the Ripper [23] enumerate combinations of tokens as dictionary candidates and are widely used by real-world attackers. Liu et al. [30] analyzed these tools using techniques of rule inversion and guess counting to retrive guessing number without explicit enumeration. Probabilistic models like Probabilistic Context-Free Grammars [28,43], Markov models [21,22,31] have been applied and analyzed in password cracking. Character-level text generation with Long-Short Term Memory (LSTM) recurrent neural networks is fast, lean and accurate in modeling password guessability [33].

Memory-Hard Functions (MHF) is a key cryptographic primitive. Evaluation of MHF requires large amount of memory in addition to longer computation time, making parallel computation and customized hardware futile to speed up computation process. Candidate MHFs include SCRYPT [35], Balloon hashing [15], and Argon2 [11] (the winner of the Password Hashing Competition[2]). MHFs can be classified into two distinct categories or modes of operation—data-independent MHFs (iMHFs) and data-dependent MHFs(dMHFs) (along with the hybrid idMHF, which runs in both modes). dMHFs like SCRYPT are maximally memory hard [7], but they have the issue of possible side-channel attacks. iMHFs, on the other hand, can resist side-channel attakcs but the aAT (amortized Area Time) complexity is at most $\mathcal{O}(N^2 \log\log N/\log N)$ [4]—a combinatorial graph property called depth-robustness is both necessarily [4] and sufficient [6] for constructing iMHFs with large aAT complexity. Ameri et al. [8] introduced the notion of a computationally data-independent MHF (ciMHF) which protects against side-channel leakage as long as the adversary is computationally bounded and constructed a ciMHF with optimal aAT complexity $\Omega(N^2)$.

2 Background and Notations

Password Dataset. We use $\mathbb{P}$ to denote the set of all possible passwords, the corresponding distribution is $\mathcal{P}$. The process of a user u choosing a password for his/her account can be viewed as a random sampling from the underlying distribution $pw_u \xleftarrow{\$} \mathcal{P}$. It will be convenient to assume that the passwords in $\mathbb{P}$ are sorted such that $\Pr[pw_1] \geq \Pr[pw_2] \geq \ldots$. Given a password dataset D of n_a accounts, we can obtain empirical distribution $\mathcal{D}_e$ by approximating $\Pr_{pw_i \sim \mathcal{D}_e}[pw_i] = \frac{f_i}{n_a}$, where f_i is the frequency of pw_i and n_a is the number of accounts present in D. Often the empirical distribution can be represented in compact form by grouping passwords with the same frequency into an *equivalence set* i.e., $D_{es} = \{(f_1, s_1), \ldots, (f_i, s_i), \ldots, (f_{n_e}, s_{n_e})\}$, where s_i is the number of

passwords which appear with frequency f_i in D and n_e is the total number of equivalence sets and, for convenience, we assume $f_1 > f_2 > \ldots > f_{n_e}$. We use $es_i = (f_i, s_i)$ to describe the ith equivalence set. In empirical experiments it is often more convenient to work with the compact representation D_{es} of password distribution. In addition, we use n_p to denote the number of distinct passwords in our dataset D. Observe that for any dataset we have $n_a \geq n_p \geq n_e$. In fact, we will typically have $n_a \gg n_p \gg n_e$.

Computation Cost of an MHF. The evaluation of $\mathsf{MHF}(x;t)$ produces a sequence of labels $L_1, L_2, \ldots, L_t$ where the last label generated L_t is the final output. Once $L_1, \ldots, L_{i-1}$ are all stored in memory it is possible to compute label i by making a single call to an underlying cryptographic hash function H e.g., we might have $L_i = H(L_j, L_k)$ where $j, k < i$ denote prior labels. We can also define $\mathsf{MHF}(x;i) = L_i$ for $i < t$. Thus, we can obtain all of the values $\mathsf{MHF}(x;1), \ldots, \mathsf{MHF}(x;t)$ in time t. We model the (amortized) Area-Time cost of evaluating $\mathsf{MHF}(\cdot;t)$ as $c_H t + c_M t^2$, where c_H and c_M are constants. Intuitively, c_H denotes the area of a core implementing the hash function H and c_M represents the area of an individual cell with the capacity to hold one data-label (hash output). Since the memory cost tend to dominant, we ignore the hash cost as simply model the cost as $c_M t^2$.

3 Defender's Model

In this section, we present the model of the defender. In particular, we describe how passwords are stored and verified on the authentication server.

Account Registration. When a user u registers for a new account with a password pw_u the authentication server randomly generates a $salt_u$ value, samples a running time parameter $t_u \in T$ from our set of possible running time breakpoints $T = \{t_1, t_2, \ldots, t_m\}$ (we let $q_i = \Pr[t_i]$ to denote the probability that $t_u = t_i$) and stores the tuple $(u, salt_u, h_u)$ where $h_u = \mathsf{MHF}(pw_u, salt_u; t_u)$. Note that the salt value $salt_u$ is recorded while the running time parameter t_u is discarded.

Password Verification. When a user u attempts to login to his/her account by submitting (u, pw'_u), the authentication server would first retrieve record $(u, salt_u, h_u)$, calculate $h_1 = \mathsf{MHF}(pw'_u, salt_u; t_1)$ and compare h_1 with h_u. It they are equal, login request is granted. Otherwise, the server would continue to calculate $h_2 = \mathsf{MHF}(pw'_u, salt_u; t_2)$, compare h_2 with h_u, so on and so forth. If any of h_i matches h_u, then user u successfully logs in his/her account. However, if for all possible running time parameters $t \in T$ we have $h_u \neq \mathsf{MHF}(pw'_u, salt_u; t)$ then the login request is rejected.

Defender Action and Workload Constraint. The defender's (leader's) action in our Stackelberg game is to select the probability distribution $q_1, \ldots, q_m$ over the running time breakpoints. The goal is to pick the distribution $q_1, \ldots, q_m$ to minimize the percentage of passwords cracked by a rational adversary subject

to constraints on the expected server workload. Whenever user u logs in with the correct password pw_u the authentication server will incur cost $c_M t_u^2$. Since $t_u = t_i$ with probability q_i the expected cost of verifying a correct password is $\sum_{i=1}^{m} q_i c_M t_i^2$. Thus, given a maximum workload parameter C_{max} we require that the distribution $q_1, \ldots, q_m$ are selected subject to the constraints that $q_i \geq 0$, $q_1 + \ldots q_m = 1$ and

$$\sum_{i=1}^{m} q_i c_M t_i^2 \leq C_{max}. \tag{1}$$

4 Attacker's Model

In this section, we first state the assumptions we use in our economic analysis. Then we show how a rational attacker who steals the password hashes from the server would run a dictionary offline attack. Finally, we present the Stackelberg game in modeling the interaction between the defender and the attacker within the framework of [12].

4.1 Assumptions of Economics Analysis

We assume that the attacker is rational, knowledgeable and untarteged. By rationality, we mean that the attacker will attempt to maximize its expected utility i.e., the value of the cracked password(s) minus the attacker's guessing costs. By knowledgeable we mean that by Kerckhoffs's principle the attacker knows the exact distribution $\mathcal{P}$ from which the user's password was sampled. In practice, an attacker would not have perfect knowledge of the distribution $\mathcal{P}$, but could still rely on sophisticated password cracking models e.g., using Neural Networks [33], Markov Models [22,40] or Probabilistic Context-Free Grammars (PCFGs) [28,42,43]. Finally, we assume that the attacker is untargetted meaning we assume that each account has the same value v for the attacker and the attacker does not have background information about the passwords that individual user's may have selected. One can derive a range of estimates for v based on black market studies e.g., Symantec reported that passwords generally sell for \$4—\$30 [25] and [38] reported that Yahoo! e-mail passwords sell for $\approx$ \$1.

4.2 Cracking Process

We now specify how an offline attacker would use the stolen hash to run a dictionary attack. The password distribution and the breakpoint distribution induce a joint distribution over pairs $(pw, t) \in \mathbb{P} \times \{t_1, \ldots, t_m\}$ where we have $\Pr[(pw_i, t_j)] = \Pr[pw_i]q_j$.

The adversary's strategy is to formulate a checking sequence $\pi = \{(pw_i, t_j)\}$ with the purpose of finding the target (pw_u, t_u). An instruction (pw_i, t_j) in π means the adversary selects pw_i as current guess and compute the jth label for pw_i i.e., evaluate $\mathsf{MHF}(pw_i, salt_u; t_j)$. The cracking process terminates when

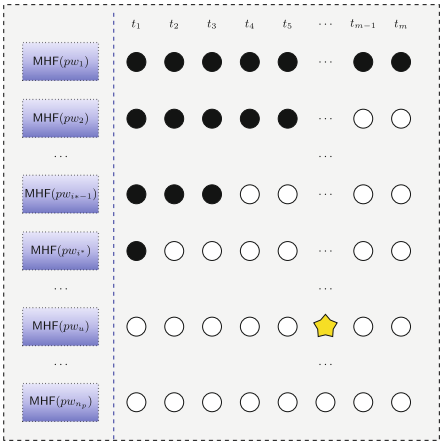


Fig. 1. Password Cracking Process. Black nodes denote current checking sequence π. White nodes denote unchecked instructions $\Pi(n_p, m) - \pi$. Star denotes unknown target (pw_u, t_u).

the adversary found the hidden target (pw_u, t_u) or timeout. Thus, the order of instructions in a checking sequence π can impact the attackers expected cost.

A checking sequence is subject to *legit restrictions*:

1. *Small label first.* If (pw_i, t_{j_1}) appears before (pw_i, t_{j_2}) in π, then it should be the case $t_{j_2} > t_{j_1}$.
2. *Label backward continuity.* If $(pw_i, t_j) \in \pi$ then $(pw_i, t_1), \ldots, (pw_i, t_{j-1}) \in \pi$.
3. *No inversions.* Inversions in the form of $(pw_{i_1}, t_{j_1}), (pw_{i_2}, t_{j_2}), (pw_{i_1}, t'_{j_1})$ where $t'_{j_1} > t_{j_1}$ are not allowed.

The first two restrictions state that the attacker cannot advance to a larger label without computing all previous labels. The third is an assumption that we made. Intuitively, the assumption is justified because an attacker who computes labels for pw_{i_2} while storing labels for pw_{i_1} will incur extra memory cost which is undesirable for a rational attacker. The cracking process is illustrated in Fig. 1.

4.3 Attacker's Utility

After specifying the restrictions for a legit checking sequence, we can formulate the the attacker's utility. Suppose the kth instruction in checking sequence π is $\pi_k = (pw_i, t_j)$, then the probability that the attacker succeeds on step k is $\Pr[\pi_k] = \Pr[pw_i] \cdot q_j$. Let $\lambda(\pi, B) \doteq \sum_{k=1}^{B} \Pr[\pi_k]$ denote the attacker's probability of success after the first $B \leq |\pi|$ instructions and let $\lambda(\pi) \doteq \lambda(\pi, |\pi|)$ denote the attacker's overall probability of success. Recall that the overall cost to compute $\mathsf{MHF}(\cdot; t_j)$ is $c_M t_j^2$. After computing $\mathsf{MHF}(pw_i; t_{j-1})$ the *additional* cost of executing instruction π_k to compute $\mathsf{MHF}(pw_i; t_j)$ is denoted $c(\pi_k) \doteq c_M(t_j^2 - t_{j-1}^2)$. For notational convenience, we define $t_0 \doteq 0$.

The attacker's utility is described by the equation below:

$$U_{adv}(v, \vec{q}, \pi) = v \cdot \lambda(\pi) - \sum_{k=1}^{|\pi|} c(\pi_k)\,(1 - \lambda(\pi, k-1))\,. \tag{2}$$

The first term in Eq. (2) gives us the attacker's expected reward. In particular, the attacker will receive value v if s/he crack's the password and, given a checking sequence π, the attacker succeeds with probability $\lambda(\pi)$ i.e., in expectation the reward is $v \cdot \lambda(\pi)$. The second term in Eq. (2) gives us the attacker's expected guesing costs, which is the summation of product of 2 terms where the probability that the attacker incurs cost $c(\pi_k)$ to evalute the instruction π_k is given by the probability that the attacker does not succeed after the first $k-1$ steps i.e., $1 - \lambda(\pi, k-1)$.

Besides legit restrictions that make a checking sequence valid a rational attacker would restrict its attention to checking sequences π that satisfy the following *opt restrictions*:

1. *Popular password first.* If (pw_{i_1}, t_j) appears before (pw_{i_2}, t_j), then $\Pr[pw_{i_1}] \geq \Pr[pw_{i_2}]$.
2. *Password backward continuity.* If $(pw_i, t_j) \in \pi$ for some j, then $(pw_{i-1}, t_{j'}) \in \pi$ for some j'.
3. *Stop at equivalence class boundary.* If (pw_i, t_j) is the last instruction in π where $pw_i \in es_k$, then $pw_{i+1} \in es_{k+1}$.

It can be easily proved that an attacker who violates opt restrictions will suffer utility loss. Legit restrictions, together with the first 2 opt restrictions, determine a complete ordering, which we call *natural ordering*, over all instructions $\{(pw_i, t_j)\}$, namely,

$$\begin{cases} (pw_{i_1}, t_{j_1}) < (pw_{i_2}, t_{j_2}), \text{ if } \Pr[pw_{i_1}] > \Pr[pw_{i_2}], \\ (pw_i, t_{j_1}) < (pw_i, t_{j_2}), \text{ if } j_1 < j_2. \end{cases} \tag{3}$$

We use $\Pi(n, m)$ to denote the sequence of all instructions for top n passwords with respect to natural ordering,

$$\Pi(n, m) := (pw_1, t_1), \ldots, (pw_1, t_m), \ldots, (pw_n, t_1), \ldots, (pw_n, t_m). \tag{4}$$

We say a sequence containing consecutive instructions for a single password is a *instruction bundle*, which is denoted by

$$\varpi_i(j_1, j_2) := (pw_i, t_{j_1}), \ldots, (pw_i, t_{j_2}). \tag{5}$$

Specifically, $\varpi_i(j_1, j_2) = \emptyset$ when $j_1 = j_2 = 0$. Then the attacker's strategy π is a sub-sequence of $\Pi(n_p, m)$ (recall that n_p is the number of distinct passwords) in the form of

$$\pi = \oplus_{i'=1}^{\mathsf{Len}(\pi)} \varpi_{i'}(1, \tau_{i'}) := \varpi_1(1, \tau_1) \circ \varpi_2(1, \tau_2) \circ \cdots \circ \varpi_{\mathsf{Len}}(1, \tau_{\mathsf{Len}}), \tag{6}$$

where $\circ$ denotes the concatenation of two disjoint instruction sequence and $\mathsf{Len}(\pi)$ is the largest index of password for which the attacker would check at

least one label, which depends on the associated checking sequence, when the context is clear it is just written as Len. Because of opt restriction 3, Len can only take values in $\{0, |es_1|, |es_1| + |es_2|, \ldots, \sum_{k=1}^{n_e} |es_k|\}$. Notice that π is fully specified by the largest label index τ_i for pw_i.

4.4 Stackelberg Game

We use Stackelberg game to model the interaction between the attacker and defender. The defender (leader) fixes a distribution $\vec{q}$ over the breakpoints $\{t_1, \ldots, t_m\}$. The attacker (follower) responds by selecting checking sequence $\pi^* = \arg\max U_{adv}(v, \vec{q}, \pi)$ to maximize its utility.

Define server's utility to be $U_{ser}(v, \vec{q}) = -\lambda(\pi^*)$, where π^* is the attacker's best response to defender's strategy $\vec{q}$ given password value v. At equilibrium no player has the incentive to deviate form her/his strategy, thus equilibrium profile $(\vec{q}^*, \pi^*)$ satisfies,

$$\begin{cases} U_{adv}(v, \vec{q}, \pi^*) \geq U_{adv}(v, \vec{q}, \pi), \forall \pi, \\ U_{ser}(v, \vec{q}^*) \geq U_{ser}(v, \vec{q}), \forall \vec{q}. \end{cases} \tag{7}$$

The defender's goal is try to find a distribution $\vec{q}$ which minimizes $\lambda(\pi^*)$ subject to the constraint that the rational attacker responds with its utility optimizing strategy π^* given the breakpoint distribution $\vec{q}$ and value parameter v. Thus, before the defender can attempt to optimize $\vec{q}$ we need to be able to compute the attacker's response π^*.

5 Computing the Attacker's Optimal Strategy

As we noted in the previous section a rational attacker will use its utility optimizing strategy $\pi^* = \arg\max U_{adv}(v, \vec{q}, \pi)$. In this section, we show how to compute the attacker's optimal strategy π^* for both time-even breakpoints and cost-even breakpoints.

Before we introduce our algorithm used to find the optimal checking sequence, let us see why the native brute force algorithm is computationally infeasible. If the attacker chose to check top Len passwords; for each password pw_i the attacker has m possible choices for each password i.e., select $\tau_i \in \{1, \ldots, m\}$ and evaluate $\mathsf{MHF}(pwd_i; t_{\tau_i})$. Thus the native brute force algorithm runs in time $\mathcal{O}\left(\sum_{\mathsf{Len}=1}^{n_p} m^{\mathsf{Len}}\right) \subseteq \mathcal{O}(m^{n_p})$ with a very large exponent ($n_p \approx 2.14 \times 10^7$ for our largest dataset Linkedin, and $n_p \approx 3.74 \times 10^5$ for our smallest dataset Bfiled). This is why we need to design polynomial time algorithms.

In the following subsections, we first specify a superset[2] of π^*, setting a boundary within which we will gradually extend the checking sequence from an empty one. Then we introduce our local search algorithm which finds the optimal

[2] We use the concept and notation of subset and superset for ordered sequences the way they were defined for regular set. If all elements of sequence A are also elements of sequence B regardless the order, we say $A \subseteq B$.

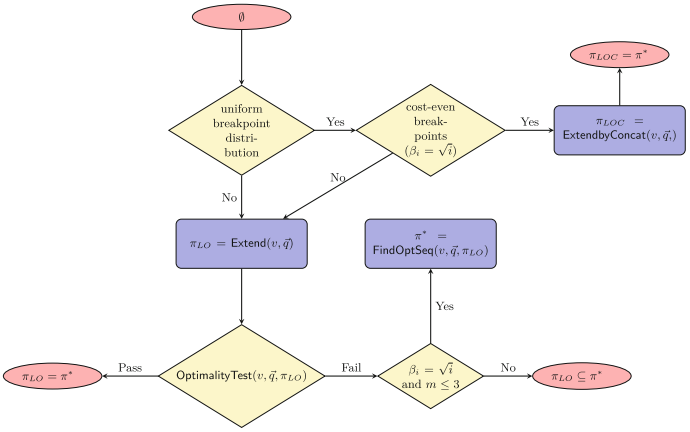


Fig. 2. Algorithm Flowchart

checking sequence most of the time. Our key intuition in designing algorithms is that an unchecked instruction bundle should be included into the optimal checking sequence if it provides non-negative marginal utility. Generally there are two local search directions, either concatenate instructions at the end of current checking sequence or insert instructions in the middle of current checking sequence. After the local search algorithm terminates we reach a local optimum π_{LO}. Finally we design algorithms to verify if the local optimum is also global optimum or promote the local optimum to global optimum under specifc parameter settings. As a overview we briefly summarize our results (also demonstrated in the flowchart, see Fig. 2) in this section as follows:

- When we use cost-even breakpoints sampled from uniform distribution, namely, $\beta_i = \sqrt{i}$ and $q_i = \frac{1}{m}$, we have a local search algorithm ExtendbyConcat $(v, \vec{q}, \emptyset)$ which iteratively considers instruction bundle that can be concatenated, ExtendbyConcat$(v, \vec{q}, \emptyset)$ runs in time $\mathcal{O}(n_p m)$ and gives optimal checking sequence;
- When breakpoints are cost-even ($\beta = \sqrt{i}$) but the distribution is non-uniform, we design an algorithm Extend$(v, \vec{q})$ which returns a locally optimal checking sequence π_{LO} in time $\mathcal{O}(n_p m)$. By locally optimal we mean that advancing any number of labels for any single password on the basis of π_{LO} will decrease attacker's utility.
  After obtaining π_{LO}, we can run a polynomial algorithm OptimalityTest$(v, \vec{q}, \pi_{LO})$ to check if π_{LO} is also a global optimum. If OptimalityTest$(v, \vec{q}, \pi_{LO})$ returns PASS, we know for sure that $\pi_{LO} = \pi^*$; otherwise, no conclusion can be drawn. If $m \leq 3$ we will use an efficient brute force algorithm FindOptSeq $(v, \vec{q}, \pi_{LO})$, which runs in time $\mathcal{O}(n_p^2)$, to the reach global optimum.
- When $\beta \neq \sqrt{i}$, regardless of the breakpoint distribution we can still run Extend$(v, \vec{q})$ to obtain locally optimal π_{LO}, and feed π_{LO} to OptimalityTest $(v, \vec{q}, \pi_{LO})$. If OptimalityTest$(v, \vec{q}, \pi_{LO})$ returns PASS, again we have $\pi_{LO} =$

π^*; if $\mathsf{OptimalityTest}(v, \vec{q}, \pi_{LO})$ returns FAIL, we cannot deduce any information about the global optimality of π_{LO}; in this case, confirm that $\pi_{LO} = \pi^*$ or promote π_{LO} to π^* will take exponential time.

5.1 Marginal Utility

Since we are going to use marginal utility as metrics of state transition in local search, we first specify how to compute marginal utility.

Definition 1. *Fixing v and $\vec{q}$, define $\Delta(\pi_1, \pi_2)$ to be marginal utility from strategy π_1 to π_2, namely,*

$$\Delta(\pi_1, \pi_2) := U_{adv}(v, \vec{q}, \pi_2) - U_{adv}(v, \vec{q}, \pi_1). \tag{8}$$

For most of the time π_2 is the result of modifying π_1 which is called *base*, in order to avoid redundantly repeating base we often write $\Delta^{\circ}(e \,|\, \pi_1)$ and $\Delta^{+}(e \,|\, \pi_1)$ to denote $\Delta(\pi_1, \pi_1 \circ e)$ and $\Delta(\pi_1, \pi_1 + e)$, respectively, where e is some ordered set of instructions, referred to as *extension*. Recall that $\circ$ is concatenation operation, here we formally introduce insertion operation $+$.

Definition 2. *Given a checking sequence $\pi = \oplus_{i=1}^{\mathsf{Len}} \varpi_i(1, \tau_i)$ and an instruction bundle $\varpi_{i'}(j_1, j_2)$, define operation $\pi + \varpi_{i'}(j_1, j_2)$ to be the checking sequence*

$$\pi + \varpi_{i'}(j_1, j_2) := \oplus_{i=1}^{i'} \varpi_i(1, \tau_i) \circ \varpi_{i'}(j_1, j_2) \circ \oplus_{i=i'+1}^{\mathsf{Len}} \varpi_i(1, \tau_i).$$

We discard superscript and comprehensively write $\Delta(e \,|\, \pi)$ to denote the marginal utility by including e into π, either through concatenation or insersion. Operations are valid only if the extension is *compatible* with the base. By compatible we mean the resulting checking sequence also satisfy both legit restrictions and opt restrictions.

When e is a singleton, from Eq. (2) we can derive the marginal utility by inserting instruction $e = (pw_i, t_j) \notin \pi$ to base π,

$$\Delta^{+}(e \,|\, \pi) = \Pr[pw_i] q_j \left(v + \sum_{e' > e, e' \in \pi} c(e') \right) - \left(1 - \sum_{e' < e, e' \in \pi} \Pr[e'] \right) c_M(t_j^2 - t_{j-1}^2). \tag{9}$$

where $\Pr[pw_i] q_j \sum_{e' > e, e' \in \pi} c(e')$ is the influence of e on future instructions since it eliminates some uncertainty about the user's password pw_u thus reduces the *expected* cost for future trials.

When e is a singleton, marginal utility upon concatenation has no future influence, hence,

$$\Delta^{\circ}(e \,|\, \pi) = \Pr[pw_i] q_j v - (1 - \lambda(\pi)) c_M(t_j^2 - t_{j-1}^2). \tag{10}$$

When e consists of multiple consecutive instructions, the marginal utility can be computed by iteratively applying Eq. (9) and (10). Namely,

$$\Delta(e \,|\, \pi) = \sum_{i=1}^{|e|} \Delta(e_i \,|\, \pi \cup \{e_0, \ldots, e_{i-1}\}), \tag{11}$$

where $e_0 = \emptyset$, e_i is the ith instruction of e and $\cup$ denotes inclusion (whether through concatenation or insertion) while maintaining natural ordering.

5.2 A Superset of the Optimal Checking Sequence

Before we present our algorithms we first show how to prune down the search space for π^*. Particularly, fixing v and $\vec{q}$ we find an index Len_{max} such that $\pi^* \subseteq \Pi(\mathsf{Len}_{max}, m)$ i.e., π^* will not even partially check passwords with rank larger than Len_{max}. Thus there is no need to consider any instructions beyond $\Pi(\mathsf{Len}_{max}, m)$ in construction of the optimal checking sequence.

Lemma 1. $\Delta^\circ(\pi_3 \,|\, \pi_1) \leq \Delta^\circ(\pi_3 \,|\, \pi_2)$, *if* $\lambda(\pi_1) \leq \lambda(\pi_2)$.

Definition 3. *Fixing* v *and* $\vec{q}$ *we define*

$$\mathsf{Len}_{max} := \begin{cases} \max_i\{i : F(v, \vec{q}, i) \geq 0\}, & \textit{if such } i \textit{ exists}, \\ 0, & \textit{o.w.} \end{cases}$$

where

$$F(v, \vec{q}, i) := \begin{cases} \max_{1\leq j\leq m}\{\Delta(\emptyset, \varpi_i(0, j))\}, & \textit{if } i = 1, \\ \max_{1\leq j\leq m}\{\Delta^\circ(\varpi_i(0, j) \,|\, \Pi(i-1, m))\}, & \textit{o.w.} \end{cases}$$

Intuitively, Len_{max} is the largest possible password index for which at least one of instruction bundles $\varpi_{\mathsf{Len}_{max}}(1, j), 1 \leq j \leq m$ provide non-negative marginal utility no matter what previous instructions are. We remark even though there is no theoretical proof of monotonicity of $F(v, \vec{q}, i)$, we have verified that $F(v, \vec{q}, i)$ is decreasing in i for our empirical password distribution. Note that by Lemma 1 we have

$$\Delta^\circ\left(\varpi_i(0, j) \,\middle|\, \oplus_{i=1}^{i-1}\varpi_i(1, \tau_i)\right) \leq F(v, \vec{q}, i),$$

if $F(v, \vec{q}, i) < 0$, then $\varpi_i(0, j)$ would certainly provide negative marginal utility, thus cannot be included in π^*. It is described in the following theorem.

Theorem 1.

$$\pi^* \subseteq \Pi(\mathsf{Len}_{max}, m).$$

5.3 Extension by Concatenation

We have established a superset of π^* in last subsection, now we design a local search algorithm that gives us a checking sequence π_{LOC} which is a subset of π^*. Here, LOC stands for "locally optimal with respect to concatenation." The sequence π_{LOC} will be helpful to further prune down the search space for π^*. In fact, in the special case where the breakpoint distribution is uniform ($q_i = \frac{1}{m}$) and cost-even breakpoints ($\beta_i = \sqrt{i}$) are used, we can prove that equality holds i.e., $\pi_{LOC} = \pi^*$ *is* the optimal solution.

To find our sequence π_{LOC} we start with the empty sequence of instructions and repeatedly include instructions that provide non-negative marginal utility upon concatenation to the current solution. We design a local search algorithm $\mathsf{ExtendbyConcat}(v, \vec{q}, \emptyset)$ to find a checking sequence π_{LOC}. Our local search algorithm $\mathsf{ExtendbyConcat}(v, \vec{q}, \emptyset)$ terminates after at most n_p rounds.

After the $i-1$th round we have $\pi_{LOC} \subseteq \Pi(i-1, m)$ i.e., the current solution only includes checking instructions for the first $i-1$ passwords. In

the ith round we find an instruction bundle for password i which maximizes (marginal) utility upon concatenation. More specifically, in round i we compute $\tau_i = \arg\max_{0\le j\le m}\{\Delta^\circ(\varpi_i(0,j)\,|\,\pi_{LOC})\}$ and append this instruction bundle to obtain an updated checking sequence $\pi_{LOC} = \pi_{LOC} \circ \varpi_i(0,\tau_i)$. Details can be found in Algorithm 1.

Algorithm 1: ExtendbyConcat$(v, \vec{q}, \pi)$

Input: v, $\vec{q}$
Output: π_{LOC}

$\pi_{LOC} = \pi$;
$start = i^*(\pi_{LOC})$;
for $i = start : n_p$ **do**
  for $j = 0 : m$ **do**
    Compute $\Delta^\circ(\varpi_i(0,j)\,|\,\pi_{LOC})$;
  end
  $\tau_i = \arg\max_{0\le j\le m}\{\Delta^\circ(\varpi_i(0,j)\,|\,\pi_{LOC})\}$;
  if $\tau_i > 0$ **then**
    $\pi_{LOC} = \pi_{LOC} \circ \varpi_i(1,\tau_i)$;
    else break;
  end
end
return π_{LOC}

We can use Eq. (10) to compute the marginal utility in time $\mathcal{O}(1)$ by caching previously computed values of $\lambda(\pi)$. Thus, ExtendbyConcat$(v, \vec{q}, \emptyset)$ runs in time $\mathcal{O}(\mathsf{Len}_{max} m) \subseteq \mathcal{O}(n_p m)$, recall that n_p is the number of distinct password.

Theorem 2.

$$\pi_{LOC} \subseteq \pi^*.$$

From Theorem 1 and Theorem 2, it is easy to derive the following corollaries.

Corollary 1.

$$\mathsf{Len}(\pi_{LOC}) \le \mathsf{Len}(\pi^*) \le \mathsf{Len}_{max},$$

and

$$\mathsf{Len}(\pi_{LOC}), \mathsf{Len}(\pi^*), \mathsf{Len}_{max} \in \{x_0, x_1, \ldots, x_{n_e}\},$$

where

$$x_k = \begin{cases} 0, & \text{if } k = 0, \\ \sum_{k'=1}^{k} |es_{k'}|, & \text{if } k = 1, \ldots, n_e. \end{cases} \tag{12}$$

Corollary 2.

$$\lambda(\pi_{LOC}) \le P_{adv} = \lambda(\pi^*) \le \lambda\left(\Pi(\mathsf{Len}_{max}, m)\right).$$

Now we have a polynomial algorithm that returns a checking sequence π_{LOC} locally optimal with respect to concatenation. The following theorem states that $\pi_{LOC} = \pi^*$ if breakpoints are cost-even and follow uniform distribution.

Theorem 3. *When* $q_i = \frac{1}{m}$ *and* $\beta_i = \sqrt{i}$, $\mathsf{ExtendbyConcat}(v, \vec{q}, \emptyset)$ *returns the optimal checking sequence, i.e.,* $\pi_{LOC} = \pi^*$.

Even though the attacker behaviors optimally—following strategy π^*. We can guarantee that our mechanism results in lower (or equal if no passwords are cracked) percentage of cracked passwords than deterministic cost hashing, which is captured by Theorem 4.

Theorem 4. *When* $\beta_i = \sqrt{i}$ *and* $q_i = \frac{1}{m}$ *then,* $\lambda(\pi^*) \leq P^d_{adv}$, *where* P^d_{adv} *is the percentage of cracked passwords in traditional deterministic cost hashing.*

We have shown that our mechanism configured with cost-even breakpoints sampled from uniform distribution will only decrease the percentage of cracked passwords. In the next subsections we consider how the attacker would react to general configuration of the mechanism.

5.4 Local Search in Two Directions

In the previous section we introduced an algorithm $\mathsf{ExtendbyConcat}(v, \vec{q}, \emptyset)$ to produce a locally optimal solution π_{LOC} with respect to concatenation. We showed the instruction sequence π_{LOC} is a subset of the instructions in π^* and argued that in specific cases the algorithm is guaranteed to find the optimal solution. However, in more general cases the local optimum may not be globally optimum. One possible reason for this is that there may be a missing instruction from π^* that we would like to insert into the middle of the checking sequence π_{LOC}, while our local search algorithm $\mathsf{ExtendbyConcat}(v, \vec{q}, \emptyset)$ only considers instructions that can be appended to π_{LOC}.

In this subsection we extend the local search algorithm to additionally consider insertions. Note that we can still use local search to test if inserting instruction bundle $\varpi_i(j_1, j_2)$ improves the overall utility, i.e., $\Delta^+(\varpi_i(j_1, j_2) \,|\, \pi) \geq 0$. We design an algorithm $\mathsf{ExtendbyInsert}(v, \vec{q}, \pi)$ which performs such an update. Combining $\mathsf{ExtendbyConcat}(v, \vec{q}, \pi)$ and $\mathsf{ExtendbyInsert}(v, \vec{q}, \pi)$, we design an Algorithm $\mathsf{Extend}(v, \vec{q})$ to construct a checking sequence π_{LO} (LO=Locally Optimal) which is locally optimal with respect to both operations: concatenation and insertions. Specifically, after each call of $\mathsf{ExtendbyInsert}(v, \vec{q}, \pi)$ we immediately run $\mathsf{ExtendbyConcat}(v, \vec{q}, \pi)$ to ensure that the solution is still locally optimal with respect to concatenation. See Algorithm 3 for details. The algorithms still maintain the invariant that π_{LO} is a subset of π^*—see Theorem 5.

Given π_{LOC} computed in time $\mathcal{O}(n_p m)$, the number of unchecked instructions is upper bounded by $|\Pi(\mathsf{Len}_{max}, m)| - |\pi_{LOC}|$. By caching the probability summation of previous and future instructions at each insertion position, verify if an instruction bundle is profitable and update the checking sequence take time $\mathcal{O}(1)$. One pass of `repeat` loop of Algorithm 3 takes time $\mathcal{O}(|\Pi(\mathsf{Len}_{max}, m)| - |\pi_{LOC}|) \subseteq \mathcal{O}(n_p m)$, the number of `repeat` loop execution is finite (in experiment it terminates after at most 3 passes). Therefore, $\mathsf{Extend}(v, \vec{q})$ runs in time $\mathcal{O}(n_p m)$.

Algorithm 2: ExtendbyInsert$(v, \vec{q}, \pi)$

Input: v, $\vec{q}$, π
Output: π_{LOI}

$\pi_{LOI} = \pi$;
while *e exists such that* $\Delta^+(e \mid \pi_{LOI}) \geq 0$ **do**
  $\pi_{LOI} = \pi_{LOI} + e$
end
return π_{LOI}

Algorithm 3: Extend$(v, \vec{q})$

Input: v, $\vec{q}$
Output: π_{LO}

$\pi_{LO} =$ ExtendbyConcat$(v, \vec{q}, \emptyset)$;
repeat
  $\pi_{LO} =$ ExtendbyInsert$(v, \vec{q}, \pi_{LO})$;
  $\pi_{LO} =$ ExtendbyConcat$(v, \vec{q}, \pi_{LO})$;
until *no single profitable instruction bundle exist*;
return π_{LO}

Lemma 2. *If* $\pi \subseteq \pi^*$ *and* $\Delta^+(e \mid \pi) \geq 0$ *then* $\pi + e \subseteq \pi^*$.

Lemma 2 guarantees that + operation preserves the invariance that our construction is subset of π^*. Naturally follows Theorem 5, which states the output of Extend$(v, \vec{q})$ is a subset of π^*.

Theorem 5. *Let* $\pi_{LO} =$ Extend$(v, \vec{q})$, *then* $\pi_{LO} \subseteq \pi^*$.

Since we are using local search to construct π_{LO}, together with Theorem 5 we know π_{LO} is a local optimum. When Algorithm 3 terminates, advancing any number of labels for any single password cannot improve the overall utility, but there is no guarantee of utility reduction upon inclusion of multiple instruction bundles that associated with different passwords. In the next subsection we will discuss how to verify if the local optimum π_{LO} is indeed the global optimum and design an efficient brute force algorithm that improves local optimum to global optimum under specific parameter settings.

5.5 Optimality Test and Globally Optimal Checking Sequence

In the previous subsections, we designed a polynomial algorithm Extend$(v, \vec{q})$ to construct locally optimal checking sequence π_{LO} with respect to insertions and concatenation. We also proved that the sequence π_{LO} is a subset of the optimal sequence π^*. In practice we find that it is often the case that $\pi_{LO} = \pi^*$ and we give an efficient heuristic algorithm which (often) allows us to confirm the global optimality of π_{LO}. In particular, our procedure will never falsely indicate

that $\pi_{LO} = \pi^*$ though it may occasionally fail to confirm that this is the case. When our optimality test fails, we design algorithms to promote locally optimal solution to globally optimal solution for cost-even breakpoints and $m \leq 3$, see full version of this paper [10] for details.

6 Defender's Optimal Strategy

When making decisions about breakpoint distribution, the defender will take attacker's best response into consideration. Specifically, the defender would choose $\vec{q}^* = \arg\min \lambda(\pi^*)$ where $\pi^* = \arg\max U_{adv}(v, \vec{q}, \pi))$. Formally, the optimization problem (OPT) is

$$\begin{aligned} \min_{\vec{q}} \quad & \lambda(\pi^*) \\ \text{s.t.} \quad & 0 \leq q_i \leq 1, \ \forall 1 \leq i \leq m, \\ & \sum_{i=1}^{m} q_i = 1, \\ & \sum_{i=1}^{m} q_i c_M t_i^2 \leq C_{max}, \\ & \pi^* = \arg\max U_{adv}(v, \vec{q}, \pi)) \end{aligned} \tag{13}$$

The optimization goal is to minimize attacker's success rate. The first two constrains guarantee q_i are valid probabilities. The third constraint forces that the expected cost does not exceed maximum workload C_{max}. The last constraint states that the attacker responds optimally given password value v and the defender's strategy $\vec{q}$. Since there is no closed form expression of $\lambda(\pi^*)$ we use heuristic black box optimization solvers to optimize $\vec{q}$. We refer to the black box solver as FindOptDis(). This heuristic algorithm is parametrized by the attacker's value v and by the password distribution $\mathcal{P}$ and outputs a distribution $\vec{q}$. As a caveat our heuristic algorithm is not absolutely guaranteed to find the optimal breakpoint distribution $\vec{q}^*$. Detailed discussion about FindOptDis() can be found in the full version of this paper [10].

7 Experiments

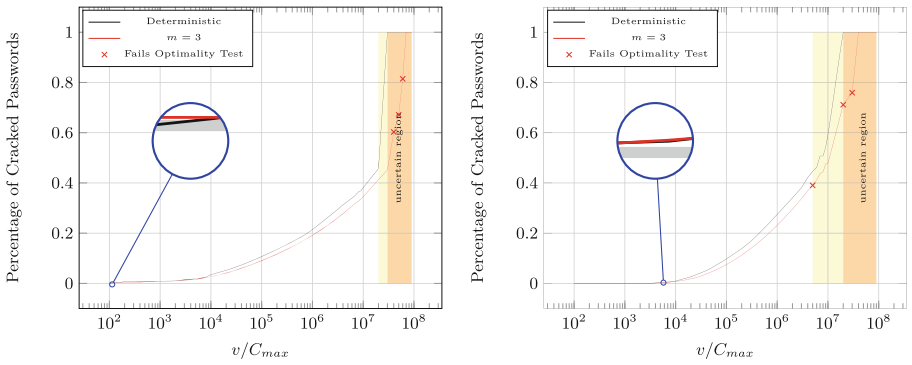
7.1 Experiment Setup

In this section, we evaluate the performance of our mechanism using empirical password datasets. Due to length limitations we only report results for the two largest datasets: Linkedin ($1.74 * 10^8$ accounts with $5.74 * 10^7$ distinct passwords) and Neopets ($6.83 * 10^7$ accounts with $2.8 * 10^7$ distinct accounts). In the full version [10] we include results for 6 additional password datasets (Bfield, Brazzers, Clicksense, CSDN, RockYou and Webhost)[3].

[3] The password datasets we analyze and experiment with are publicly available and widely used in literature research. We did not crack any new passwords. Thus, our usage of the datasets would not cause further harm to users.

For each dataset we derive the corresponding empirical distribution $\mathcal{D}_e$ (namely, $\Pr_{pw\sim\mathcal{D}_e}[pw] = f_i/n_a$ where f_i is the frequency of pw) and analyze the attacker's success rate under this password distribution. The drawback is that the tail of empirical distribution $\mathcal{D}_e$ can significantly diverge from real distribution $\mathcal{P}$. We follow the approach of [9] and use Good-Turing Frequency estimation to upbound the CDF divergence E between $\mathcal{D}_e$ and $\mathcal{P}$. In particular, we use yellow (resp. red) to denote the unconfident region where the empirical distribution might diverge significantly from the real distribution $E > 0.01$ (resp. $E > 0.1$).

We plot the attacker's success rate $\lambda(\pi^*)$ as the ratio v/C_{max} varies under different conditions. In Fig. 3 we consider time-even breakpoints with uniform distribution over breakpoints. Similarly, Fig. 4 considers cost-even breakpoints under the uniform distribution as the number of breakpoints m varies. In Fig. 5, we fix $m = 3$ continue to use cost-even breakpoints, and run our algorithm FindOptDis() (implemented with BITEOPT [41]), to optimize the breakpoint distribution.

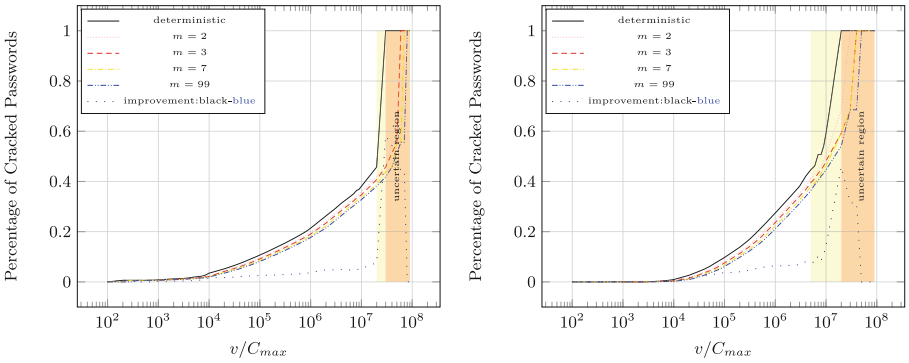


(a) Linkedin (b) Neopets

Fig. 3. Time-Even Breakpoints, Uniform Breakpoint Distribution

7.2 Experiment Analysis and Discussion

Time-Even Breakpoints with Uniform Distribution. Fig. 3 plots the attacker's success rate (vs. v/C_{max}) when we use time-even breakpoints with the uniform distribution i.e., Boyen's Halting puzzles [18]. In most parameter ranges the usage of Boyen's Halting puzzles reduces the % of cracked passwords in comparison to using deterministic (cost-equivalent) memory hard functions. However, one significant observation is that for some parameters v/C_{max} (highlighted with amplified circles on the plots) using Boyen's halting puzzles can actually increase the percentage of cracked passwords. Take LinkedIn as example, when $v/C_{max} = 100$ using time-even breakpoints increases the % of cracked passwords from 0% (determistic MHF) to 0.2%. Similar phenomenon can be observed in



(a) Linkedin (b) Neopets

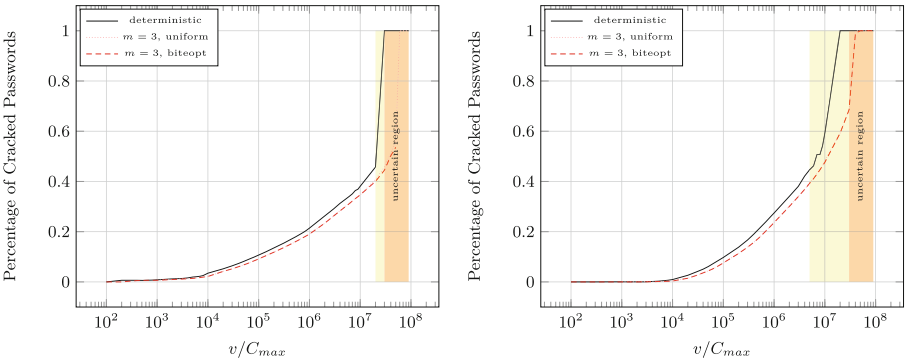
Fig. 4. Cost-Even Breakpoints, Uniform Breakpoint Distribution

other datasets. Intuitively, these findings are explained by the observation that it is relatively cheap for the attacker to check the first few cost-even breakpoints.

We also provide a (admitedly contrived) example to show that time-even breakpoints could be very harmful. Suppose a dataset has 2 passwords, each occurs with probability $\frac{1}{2}$, and password value $v = 1.45$, and cost parameter $C_{max} = 1$. With deterministic MHFs a simple calculation shows that the rational attacker's utility optimal strategy is to give up immediately without checking any passwords. On the other hand, if we use Halting Puzzles (time-even breakpoints with a uniform distribution) then a rational attacker will recover the user's password with probability at least $\frac{1}{4}$ e.g., a rational attacker will always want to check the first label of both passwords.

Cost-Even Breakpoints and Uniform Distribution. Figure 5 plots the success rate of the rational adversary when we use cost-even breakpoints with the uniform distribution. Our results are consistent with Theorem 4 where we proved that cost-even breakpoints with the uniform distribution can *never* increase the attacker's success rate. In Fig. 5 we also explore the impact of increasing the number of breakpoints m. We find that increasing m decreases the attacker's success rate although the impact dimishes as m increases—see the [10] for additional discussion. When $m = 99$ we find instances where the attacker's success rate is decreased by an additive factor of 10%.

Optimized Distribution and Cost-Even Breakpoints. Continuing to use cost-even breakpoints we attempted to optimize the breakpoint distribution using BITEOPT[41]—see Fig. 5. In all instances we only obtained marginal reductions in the attacker's success rate when compared to the uniform distribution over breakpoints. Furthermore, optimizing the breakpoint distribution $\vec{q}$ requires the defender to know the password distribution and the attacker's value v a priori. In practice there is a very real risk that we would optimize $\vec{q}$ with respect to



(a) Linkedin (b) Neopets

Fig. 5. Cost-Even Breakpoints, Optimized Breakpoint Distribution

the wrong distribution or value v. Thus we recommend to use cost-even break points with uniform distribution as this solution can be implemented *without* any knowledge of v or the password distribution.

8 Conclusion

In this paper, we introduce cost-asymmetric memory hard password authentication, a prior independent authentication mechanism, to defend against offline attacks. As traditional hash function are replaced by memory hard functions, we propose to use random breakpoints in evaluation of an MHF in order to have the benefit of both cost asymmetry and cost quadratic scaling. The interaction between the defender and the attacker is modeled by a Stackelberg game, within the game theory framework we formulate the optimal strategies for both defender and attacker. We theoretically proved that cost-asymmetric memory hard password authentication with cost-even breakpoints sampled from uniform distribution will reduce attacker's cracking success rate. In addition we set up experiments to validate the effectiveness of our proposed mechanism for arbitrary parameter settings, experiment results show that the reduction of attacker's success rate is up to 10%.

Acknowledgments. The research was supported in part by the National Science Foundation under awards CNS #2047272 and by IARPA under the HECTOR program. Mohammad Hassan Ameri was also supported in part by a Summer Research Grant from Purdue University.

References

1. Hashcat: advanced password recovery. https://hashcat.net/hashcat/
2. Password hashing competition. https://password-hashing.net/

3. Adams, A., Sasse, M.A.: Users are not the enemy. Commun. ACM **42**(12), 40–46 (1999)
4. Alwen, J., Blocki, J.: Efficiently computing data-independent memory-hard functions. In: Robshaw, M., Katz, J. (eds.) CRYPTO 2016. LNCS, vol. 9815, pp. 241–271. Springer, Heidelberg (2016). https://doi.org/10.1007/978-3-662-53008-5_9
5. Alwen, J., Blocki, J., Harsha, B.: Practical graphs for optimal side-channel resistant memory-hard functions. In: Thuraisingham, B.M., Evans, D., Malkin, T., Xu, D. (eds.) ACM CCS 2017, pp. 1001–1017. ACM Press, Dallas, TX, USA, 31 Oct–2 Nov 2017. https://doi.org/10.1145/3133956.3134031
6. Alwen, J., Blocki, J., Pietrzak, K.: Depth-robust graphs and their cumulative memory complexity. In: Coron, J.-S., Nielsen, J.B. (eds.) EUROCRYPT 2017. LNCS, vol. 10212, pp. 3–32. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-56617-7_1
7. Alwen, J., Chen, B., Pietrzak, K., Reyzin, L., Tessaro, S.: `Scrypt` is maximally memory-hard. In: Coron, J.-S., Nielsen, J.B. (eds.) EUROCRYPT 2017. LNCS, vol. 10212, pp. 33–62. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-56617-7_2
8. Ameri, M.H., Blocki, J., Zhou, S.: Computationally data-independent memory hard functions. In: Vidick, T. (ed.) ITCS 2020. vol. 151, pp. 36:1–36:28. LIPIcs, Seattle, WA, USA, 12–14 Jan 2020. https://doi.org/10.4230/LIPIcs.ITCS.2020.36
9. Bai, W., Blocki, J.: DAHash: distribution aware tuning of password hashing costs. In: Borisov, N., Diaz, C. (eds.) FC 2021. LNCS, vol. 12675, pp. 382–405. Springer, Heidelberg (2021). https://doi.org/10.1007/978-3-662-64331-0_20
10. Bai, W., Blocki, J., Ameri, M.H.: Cost-asymmetric memory hard password hashing (2022). https://arxiv.org/abs/2206.12970
11. 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)
12. Blocki, J., Datta, A.: CASH: a cost asymmetric secure hash algorithm for optimal password protection. In: IEEE 29th Computer Security Foundations Symposium, pp. 371–386 (2016)
13. Blocki, J., Harsha, B., Zhou, S.: On the economics of offline password cracking. In: 2018 IEEE Symposium on Security and Privacy. pp. 853–871. IEEE Computer Society Press, San Francisco, CA, USA, 21–23 May 2018. https://doi.org/10.1109/SP.2018.00009
14. Blocki, J., Komanduri, S., Procaccia, A., Sheffet, O.: Optimizing password composition policies. In: Proceedings of the Fourteenth ACM Conference on Electronic Commerce, pp. 105–122. ACM (2013)
15. Boneh, D., Corrigan-Gibbs, H., Schechter, S.: Balloon hashing: a memory-hard function providing provable protection against sequential attacks. In: Cheon, J.H., Takagi, T. (eds.) ASIACRYPT 2016. LNCS, vol. 10031, pp. 220–248. Springer, Heidelberg (2016). https://doi.org/10.1007/978-3-662-53887-6_8
16. Bonneau, J.: The science of guessing: analyzing an anonymized corpus of 70 million passwords. In: 2012 IEEE Symposium on Security and Privacy, pp. 538–552. IEEE Computer Society Press, San Francisco, CA, USA, 21–23 May 2012. https://doi.org/10.1109/SP.2012.49
17. Bonneau, J., Herley, C., van Oorschot, P.C., Stajano, F.: The quest to replace passwords: a framework for comparative evaluation of web authentication schemes. In: 2012 IEEE Symposium on Security and Privacy, pp. 553–567. IEEE Computer Society Press, San Francisco, CA, USA, 21–23 May 2012. https://doi.org/10.1109/SP.2012.44

18. Boyen, X.: Halting password puzzles: hard-to-break encryption from human-memorable keys. In: Provos, N. (ed.) USENIX Security 2007, pp. 6–10, Boston, MA, USA. Aug, USENIX Association (2007)
19. Campbell, J., Ma, W., Kleeman, D.: Impact of restrictive composition policy on user password choices. Behav. Inf. Technol. **30**(3), 379–388 (2011)
20. Carnavalet, X., Mannan, M.: From very weak to very strong: analyzing password-strength meters. In: NDSS 2014. The Internet Society, San Diego, CA, USA, 23–26 Feb 2014
21. Castelluccia, C., Chaabane, A., Dürmuth, M., Perito, D.: When privacy meets security: leveraging personal information for password cracking. arXiv preprint arXiv:1304.6584 (2013)
22. Castelluccia, C., Dürmuth, M., Perito, D.: Adaptive password-strength meters from Markov models. In: NDSS 2012. The Internet Society, San Diego, CA, USA, 5–8 Feb 2012
23. Designer, S.: John the ripper password cracker (2006)
24. Florêncio, D., Herley, C., Van Oorschot, P.C.: An administrator's guide to Internet password research. In: Proceedings of the 28th USENIX Conference on Large Installation System Administration, pp. 35–52. LISA 2014 (2014)
25. Fossi, M., et al.: Symantec report on the underground economy (2008). Accessed 1 Aug 2013
26. Inglesant, P.G., Sasse, M.A.: The true cost of unusable password policies: password use in the wild. In: Proceedings of the SIGCHI Conference on Human Factors in Computing Systems, pp. 383–392. CHI 2010, ACM, New York, NY, USA (2010). https://doi.org/10.1145/1753326.1753384
27. Kaliski, B.: Pkcs# 5: password-based cryptography specification version 2.0 (2000)
28. Kelley, P.G., et al.: 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 Computer Society Press, San Francisco, CA, USA, 21–23 May 2012. https://doi.org/10.1109/SP.2012.38
29. Komanduri, S., et al.: Of passwords and people: measuring the effect of password-composition policies. In: CHI, pp. 2595–2604 (2011). http://dl.acm.org/citation.cfm?id=1979321
30. Liu, E., Nakanishi, A., Golla, M., Cash, D., Ur, B.: Reasoning analytically about password-cracking software. In: 2019 IEEE Symposium on Security and Privacy (SP), pp. 380–397. IEEE (2019)
31. 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 Computer Society Press, Berkeley, CA, USA, 18–21 May 2014. https://doi.org/10.1109/SP.2014.50
32. Manber, U.: A simple scheme to make passwords based on one-way functions much harder to crack. Comput. Secur. **15**(2), 171–176 (1996)
33. Melicher, W., et al.: Fast, lean, and accurate: modeling password guessability using neural networks. In: Holz, T., Savage, S. (eds.) USENIX Security 2016, pp. 175–191. USENIX Association, Austin, TX, USA, 10–12 Aug 2016
34. Morris, R., Thompson, K.: Password security: a case history. Commun. ACM **22**(11), 594–597 (1979)
35. Percival, C.: Stronger key derivation via sequential memory-hard functions. In: BSDCan 2009 (2009)
36. Provos, N., Mazieres, D.: Bcrypt algorithm. USENIX (1999)

37. Steves, M., Chisnell, D., Sasse, A., Krol, K., Theofanos, M., Wald, H.: Report: authentication diary study. Technical report NISTIR 7983, National Institute of Standards and Technology (NIST) (2014)
38. Stockley, M.: What your hacked account is worth on the dark web (2016). https://nakedsecurity.sophos.com/2016/08/09/what-your-hacked-account-is-worth-on-the-dark-web/
39. Ur, B., et al.: How does your password measure up? the effect of strength meters on password creation. In: Proceedings of USENIX Security Symposium (2012)
40. Ur, B., et al.: Measuring real-world accuracies and biases in modeling password guessability. In: Jung, J., Holz, T. (eds.) USENIX Security 2015, pp. 463–481. USENIX Association, Washington, DC, USA, 12–14 Aug 2015
41. Vaneev, A.: BITEOPT - derivative-free optimization method (2021). https://github.com/avaneev/biteopt. c++ source code, with description and examples
42. Veras, R., Collins, C., Thorpe, J.: On semantic patterns of passwords and their security impact. In: NDSS 2014. The Internet Society, San Diego, CA, USA, 23–26 Feb 2014
43. Weir, M., Aggarwal, S., de Medeiros, B., Glodek, B.: Password cracking using probabilistic context-free grammars. In: 2009 IEEE Symposium on Security and Privacy, pp. 391–405. IEEE Computer Society Press, Oakland, CA, USA, 17–20 May 2009. https://doi.org/10.1109/SP.2009.8