Cost-Asymmetric Memory Hard Password Hashing

Wenjie Bai[a], Jeremiah Blocki[a], Mohammad Hassan Ameri[a]

[a]*Purdue University, 610 Purdue Mall, West Lafayette, 47907, Indiana, United States*

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 reduce the percentage of user passwords that are cracked by a rational attacker by up to 10%.

Keywords: Memory Hard Functions, Password Authentication, Password Offline Attack, 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 and $salt_u$[1] the attacker can check if $pw_u = pw'$ by computing

Email addresses: bai104@purdue.edu (Wenjie Bai), jblocki@purdue.edu (Jeremiah Blocki), mameriek@purdue.edu (Mohammad Hassan Ameri)

[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 salt values and collision resistance of the cryptographic hash function H.

$h' = H(pw', salt_u)$ 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 [1]. Thus, it is imperative 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 [2] and BCRYPT [3]. 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. [4] argued that functions like BCRYPT or PBKDF2 cannot provide adequate protection against an offline attacker without introducing an unacceptable authentication delay e.g., 2 minutes.

Memory-Hard Functions (MHFs) [5] have been introduced to address the short-comings of hash-iteration based key-stretching algorithms like BCRYPT and PBKDF2. Candidate MHFs include SCRYPT [5], Argon2 (which was declared as the winner of Password Hashing Competition [6] in 2015) and DRSample [7]. 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 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 desirable as it allows an authentication server to increase password guessing costs rapidly without necessarily introducing an unacceptable authentication delay.

Peppering [8] 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.

> *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 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 (if all pepper values were tried out in parallel, the hash cost paid by the both authentication server and the attacker would be the equally expensive, which defeats the purpose of using pepper and incurs excessive computational workload for the server). 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 [9] introduced the notion of a halting puzzle where the "pepper" value is replaced with a random running time parameter. Boyen proposed to have the *user* select a running time parameter t_u in addition to their password pw_u. The system would then pick a random salt value $salt_u$ and generate the hash value $h_u = \mathsf{MHF}(pw_u, salt_u; t_u)$. The system would store the tuple $(u, salt_u, h_u)$, but running time time parameter t_u would be discarded[2]. 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 $L_{i+1} = H(L_i, L_j)$ where $j < i$ and H is the underlying cryptographic hash function. Thus, whenever the user attempts to login with a password pw'_u the system can simply start computing $\mathsf{MHF}(pw'_u, salt_u; \infty)$ 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 the system will be able to halt after t_u iterations. By contrast, if the user enters the wrong password $pw'_u \neq pw_u$ the system will effectively enter an "infinite loop" until the user manually terminates the process. Thus, in the remote-authentication setting it is necessary to impose an upper-bound t_{max} on the running time parameter t_u to avoid denial-of-service attacks when users mistakenly (or maliciously) enter the wrong password. Now the only way to definitely reject an incorrect password pw'_u is to finish computing $\mathsf{MHF}(pw'_u, salt_u; t_{max})$. The authentication delay is at most t_{max} and it seems like the attacker's area-time cost for each inccorect password guess will scale

[2]Boyen [9] was originally motivated by applications such as hard-drive encryption. In this application, the final value h_u would be used to derive a symmetric encryption key k_u and would not be stored directly on the authentication server.

quadratically i.e., t_{max}^2. 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 memory hard function $\mathsf{MHF}(pw', salt_u; t_{max})$ for each password guess. For example, suppose that 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 the first $\frac{t_{max}}{3}$ labels, i.e., compute $\mathsf{MHF}(pw', salt_u; t_{max}/3)$ comparing the stolen hash h_u with each of the first $t_{max}/3$ labels. The area-time cost to evaluate $\mathsf{MHF}(pw', salt_u; t_{max}/3)$ would decrease by a factor of 9 in comparison to the area-time cost to evaluate $\mathsf{MHF}(pw', salt_u, t_{max})$ i.e., $t_{max}^3/3^2$ vs t_{max}^2. By contrast, the attacker's success rate only diminishes by a factor of $1/3$ i.e., $\Pr[t_u \in [1, t_{max}/3]] = 1/3$. 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 extension 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 $t_m^2 = t_i^2(m/i)$ for all $1 \leq i < m$. In other words, we define $\beta_i := \frac{t_i}{t_1}$ and set $\beta_i = \sqrt{i}$ for cost even points. 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 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 stores the tuple $(u, salt_u, h_u)$ as the record for user u.

We adapt the Stackelberg game theoretic framework of Blocki and Datta [10] 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

Offline password attacks have been a major security concern for decades [11]. The risks are amplified by the well documented human tendency to pick low-entropy passwords [12] which are easier for an attacker to crack. Efforts to persuade (or force) user's to pick stronger passwords have shown mixed results [13]. For example, prior work has found that password strength meters often fail to persuade users to pick stronger passwords [14, 15]. Other websites impose stringent guidelines on the passwords that users can select (e.g., the password must contain number(s), uppercase letters, lowercase letters and/or special symbols). However, empirical studies have shown that these policies often incur undesirable usability costs [16, 17, 18, 19], and in some cases actually lead to users selecting weaker passwords [20, 21]. Thus, there is still an urgent need to protect low-entropy passwords against offline brute-force attackers.

Memory-Hard Functions (MHF) are a key cryptographic primitive for protecting lower entropy secrets against offline attacks. 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 [5], Balloon hashing [22], and Argon2 [23] (the winner of the Password Hashing Competition [6]). 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 [24], but they have the issue of possible side-channel attacks. iMHFs, on the other hand, can resist side-channel attacks but the aAT (amortized Area Time) complexity is at most $O(N^2 \log \log N / \log N)$ [25] — a combinatorial graph property called depth-robustness is both necessary [25] and sufficient [26] for constructing iMHFs with large aAT complexity. Ameri et al. [27] 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 $\Omega(N^2)$.

In this work we use MHFs as a black-box component in construction of our authentication mechanism to defend against offline attackers who have access to sophisticated password cracking models. We integrate the idea of password peppering with MHF so that our mechanism achieves a further skewed cost asymmetry, which makes massive offline attacks economically infeasible.

Our work is most closely related to Boyen's work on halting puzzles [9]. We modify Boyen's solution by introducing cost-even breakpoints and by introducing a hard cap t_{max} on the running time parameter. Our analysis framework also differs from [9] in several significant ways leading us to reach a very different conclusion about what the distribution over the running time parameter $t_u \leq t_{max}$ should look like. In particular, we consider a rational attacker who may quit cracking early as opposed to a persistent attacker [9] who will always continue attacking until the password is cracked — see Section 6.1 for additional discussion.

2. Background and Notation

Password Distributions and Datasets. 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 memory hard function $\mathsf{MHF}(x; t)$ 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 tends 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 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 [10].

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 [28], Markov Models [29, 30] or Probabilistic Context-Free Grammars (PCFGs) [31, 32, 33]. Finally, we assume that the attacker is untargetted meaning 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 [34] and [35] reported that Yahoo! e-mail passwords sell for $\approx$ \$1.

4.2. Cracking Process

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 $\Pr[(pw_i, t_j)] = \Pr[pw_i] q_j$. Thus, to recover the user's password the attacker will need to recover the pair (pw_u, t_u) where pw_u (resp. t_u) is the user's password (resp. breakpoint). We can specify the attacker's strategy as an ordered list of pairs $[pw_1, j_1], [pw_2, j_2], \ldots$ with $j_i \leq m$. Here, the instruction $[pw_i, j_i]$ means that the attacker will begin computing $\mathsf{MHF}(pw_i, salt_u; t_{j_i})$ to generate a stream of j_i labels. Of course when we evaluate $\mathsf{MHF}(pw_i, salt_u; t_{j_i})$ we will also evaluate $\mathsf{MHF}(pw_i, salt_u; t_r)$ for each $r < j_i$. If $pw_u = pw_i$ and $t_u \leq t_{j_i}$ then the attacker will notice that the label $\mathsf{MHF}(pw_i, salt_u; t_u)$ matches the stored hash value. At this point the attacker can immediately quit and output the user's password $pw_u = pw_i$. The visual representation of this cracking process is illustrated in Figure 1.

We will make the assumption that each password appears at most once in our ordered list $[pw_1, j_1], [pw_2, j_2], \ldots$. This assumption is intuitively justified by the observation that a rational attacker would be unlikely to benefit by computing $\mathsf{MHF}(pw_i, salt_u; t_{j_i})$ clearing memory and

then later computing $\mathsf{MHF}(pw_i, salt_u; t)$ for some $t > t_{j_i}$ — such a strategy requires the attacker to incur additional cost to recompute $\mathsf{MHF}(pw_i, salt_u; t_{j_i})$ and recover the discarded labels before s/he can finish computing $\mathsf{MHF}(pw_i, salt_u; t)$. Assuming that each password appears at most once in our ordered list we can *prove* that a rational attacker will order password guesses in descending order of likelihood i.e., $\Pr[pw_i] \geq \Pr[pw_{i+1}]$.

It will be convenient to re-write the attacker's checking sequence in atomic form i.e., the instruction $[pw_i, j_i]$ would be replaced by the atomic sequence $(pw_1, 1), \ldots (pw_i, j_i)$. We will use π to denote an ordered checking sequence written in atomic form. Written in atomic form the ordered checking sequence satisfies the following *natural ordering* over atomic instructions,

$$\begin{cases} (pw_{i_1}, t_{j_1}) < (pw_{i_2}, t_{j_2}), \text{ if } \Pr[pw_{i_1}] > \Pr[pw_{i_2}], \\ (pw_i, t_{j_1}) < (pw_i, t_{j_2}), \text{ if } j_1 < j_2. \end{cases} \tag{2}$$

We use $\Pi(n, m)$ to denote the atomic sequence corresponding to $[pw_1, t_m], \ldots, [pw_n, t_m]$ i.e., the checking sequence where we completely check all of the top n passwords. Formally,

$$\Pi(n, m) := (pw_1, t_1), \ldots, (pw_1, t_m), \ldots, (pw_n, t_1), \ldots, (pw_n, t_m)\,. \tag{3}$$

We also let

$$\varpi_i(j_1, j_2) := (pw_i, t_{j_1}), \ldots, (pw_i, t_{j_2}) \tag{4}$$

denote a sequence of consecutive atomic instructions for a single password pw_i i.e., an *instruction bundle*. We can then write the attacker's strategy in the form

$$\pi = \bigcirc_{i=1}^{\mathsf{Len}(\pi)} \varpi_i(1, t_{j_i}) := \varpi_1(1, t_{j_1}) \circ \varpi_2(1, t_{j_2}) \circ \cdots \circ \varpi_{\mathsf{Len}}(1, t_{j_{\mathsf{Len}}}), \tag{5}$$

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. Notice that π is fully specified by the largest label index t_{j_i} for each password pw_i and that π is a sub-sequence of $\Pi(n_p, m)$ where n_p is the total number of passwords in the distribution.

4.3. Attacker's Utility

After specifying the attacker's strategy in the form of a checking sequence, we can formulate 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:

$$U_{adv}(v, \vec{q}, \pi) = v \cdot \lambda(\pi) - \sum_{k=1}^{|\pi|} c(\pi_k)\,(1 - \lambda(\pi, k-1))\,. \tag{6}$$

The first term in equation (6) 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

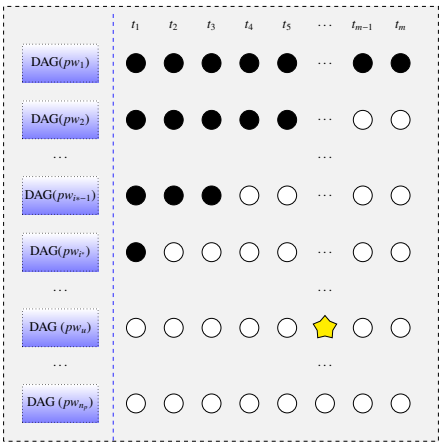


Figure 1: Password Cracking Process

Black nodes denote current checking sequence π and the attacker would check black nodes in the first row from left to right then move to the second row, etc. White nodes denote unchecked instructions $\Pi(n_p, m) - \pi$. Star denotes unknown target (pw_u, t_u).

succeeds with probability $\lambda(\pi)$ i.e., in expectation the reward is $v \cdot \lambda(\pi)$. The second term in equation (6) gives us the attacker's expected guessing 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., $1 - \lambda(\pi, k-1)$.

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 consider the algorithmic challenge of finding π^*.

We present a local search algorithm which is guaranteed to find the attacker's optimal strategy π^* under certain restrictions e.g., the defender utilizes cost-even breakpoints with a uniform distribution. In other settings (e.g., time-even breakpoints) the local search algorithm is not always guaranteed to find the optimal solution, but we are still able to design an efficient optimality test which can be used to verify if the optimal solution was found. We remark that the breakpoints setting, which is the defender's strategy in the first stage of the Stackelberg game, as well as password value and password distribution is known to the attacker at the time of developing the (locally) optimal strategy in the second stage of the game.

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 $O\left(\sum_{\mathsf{Len}=1}^{n_p} m^{\mathsf{Len}}\right) \subseteq O(m^{n_p})$ 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 Bfield). This is why we need to design polynomial time algorithms.

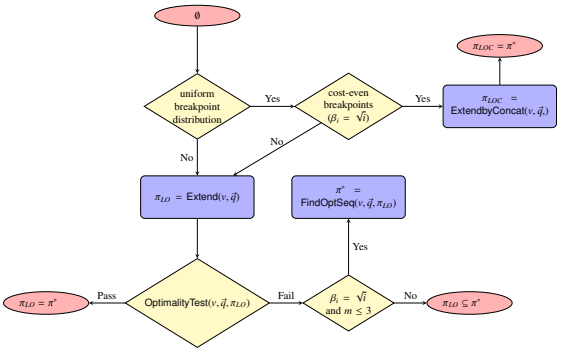


Figure 2: Algorithm Flowchart

In the following subsections, we first specify a superset[3] 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 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 specific parameter settings. As a overview we briefly summarize our results (also demonstrated in the flowchart, see Figure 2) in this section as follows:

- When we use cost-even breakpoints sampled from uniform distribution, namely, $\beta_i = \frac{t_i}{t_1} = \sqrt{i}$ and $q_i = \frac{1}{m}$, we have a local search algorithm $\mathsf{ExtendbyConcat}(v, \vec{q}, \emptyset)$ which iteratively

[3]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 of the order, we say $A \subseteq B$

considers instruction bundle that can be concatenated, $\mathsf{ExtendbyConcat}(v, \vec{q}, \emptyset)$ runs in time $O(n_p m)$ and provably gives the optimal checking sequence;

- When breakpoints are cost-even ($\beta_i = \sqrt{i}$) but the distribution is non-uniform, we design an algorithm $\mathsf{Extend}(v, \vec{q})$ which returns a locally optimal checking sequence π_{LO} in time $O(n_p m)$. By locally optimal we mean that advancing any number of labels, i.e., evaluate the MHF with subsequent running time parameters, 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 $\mathsf{OptimalityTest}(v, \vec{q}, \pi_{LO})$ to check if π_{LO} is also a global optimum. If $\mathsf{OptimalityTest}(v, \vec{q}, \pi_{LO})$ 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 $\mathsf{FindOptSeq}(v, \vec{q}, \pi_{LO})$, which runs in time $O(n_p^2)$, to reach the global optimum.

- When $\beta_i \neq \sqrt{i}$, regardless of the breakpoint distribution we can still run $\mathsf{Extend}(v, \vec{q})$ to obtain locally optimal π_{LO}, and feed π_{LO} to $\mathsf{OptimalityTest}(v, \vec{q}, \pi_{LO})$. If $\mathsf{OptimalityTest}(v, \vec{q}, \pi_{LO})$ returns PASS, again we have $\pi_{LO} = \pi^*$; 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, confirming that $\pi_{LO} = \pi^*$ or finding 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 $\Delta(\pi_1, \pi_1 \circ e)$ and $\Delta(\pi_1, \pi_1 + e)$ in short as $\Delta^\circ(e \,|\, \pi_1)$ and $\Delta^+(e \,|\, \pi_1)$, respectively, where e is some ordered set of instructions, referred to as *extension*. Recall that $\circ$ is concatenation operation, here we formally introduce the insertion operation $+$.

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

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

For notational convenience we occasionally discard the superscript and 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 must satisfy all of our restrictions.

When e is a singleton, from equation (6) 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 $\Pr[pw_i]q_j \sum_{e'>e, e' \in \pi} c(e')$ captures the influence of e on the expected cost of 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 equation (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.

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

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

where

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

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. Note that by Lemma 1, given any checking sequence $\pi = \bigcirc_{i'=1}^{i-1} \varpi_i(1, \tau_{i'})$ the marginal utility upon concatenation of instruction bundle $\varpi_i(0, j)$ satisfies

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

If $F(v, \vec{q}, i) < 0$, then $\varpi_i(0, j)$ would certainly provide negative marginal utility, hence cannot be included in π^*. This intuition is formalized by Theorem 1.

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

Proof. Suppose e_k is the ith instruction bundle in e and takes the form of (pw_i, t_j), from equation 10 we have

$$\Delta^{\circ}(e_1 \,|\, \pi_1) = \Pr[pw_i]q_j v - (1 - \lambda(\pi_1))\, c_M(t_j^2 - t_{j-1}^2),$$

and

$$\Delta^{\circ}(e_1 \,|\, \pi_2) = \Pr[pw_i]q_j v - (1 - \lambda(\pi_2))\, c_M(t_j^2 - t_{j-1}^2).$$

Thus,

$$\Delta^\circ(e_1 \,|\, \pi_1) \le \Delta^\circ(e_1 \,|\, \pi_2).$$

Similarly, we can repeatedly append instruction bundle e_i to $\pi_1 \circ \bigcirc_{i'=1}^{i-1} e_{i'}$ (resp. $\pi_2 \circ \bigcirc_{i'=1}^{i-1} e_{i'}$) to obtain

$$\Delta^\circ\left(e_i \,\middle|\, \pi_1 \circ \bigcirc_{i'=1}^{i-1} e_{i'}\right) \le \Delta^\circ\left(e_i \,\middle|\, \pi_2 \circ \bigcirc_{i'=1}^{i-1} e_{i'}\right), \forall e_i \subseteq e.$$

Summing up the above inequalities over i, we have

$$\Delta^\circ(e \,|\, \pi_1) = \sum_i \Delta^\circ\left(e_i \,\middle|\, \pi_1 \circ \bigcirc_{i'=1}^{i-1} e_{i'}\right) \le \sum_i \Delta^\circ\left(e_i \,\middle|\, \pi_2 \circ \bigcirc_{i'=1}^{i-1} e_{i'}\right) = \Delta^\circ(e \,|\, \pi_2).$$

□

Theorem 1.

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

Proof. Given $\pi^* = \bigcirc_{i'=1}^{i^*} \varpi_{i'}(1, \tau_{i'})$, suppose there exists $\varpi_i(1, \tau_i) \subseteq \pi^*$, for some $i > \mathsf{Len}_{max}, \tau_i > 0$, we have

$$\begin{aligned}
&\Delta^\circ\left(\varpi_i(1,\tau_i) \,\middle|\, \bigcirc_{i'=1}^{i-1} \varpi_i(1,\tau_{i'})\right) \\
&\le \max_{1\le j\le m} \left\{\Delta^\circ\left(\varpi_i(1,j) \,\middle|\, \bigcirc_{i'=1}^{i-1} \varpi_i(1,\tau_{i'})\right)\right\} \le \max_{1\le j\le m} \{\Delta^\circ(\varpi_i(1,j) \,|\, \Pi(i-1,m))\} && \text{by Lemma 1} \\
&= F(v,\vec{q},i) < 0. && \text{by definition of } \mathsf{Len}_{max}
\end{aligned}$$

Then we can safely remove instructions $\varpi_i(1, \tau_i)$ for all $i > \mathsf{Len}_{max}$ from π^* to obtain another checking sequence that yields a better utility. Contradiction. □

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 in further pruning 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, recall that n_p is the number of distinct password. After the $i-1$th round we obtain $\pi_{LOC} \subseteq \Pi(i-1, m)$ 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 the marginal utility upon concatenation. More specifically, in round i we compute the stopping label index $\tau_i = \arg\max_{0\le j\le m}\{\Delta^\circ(\varpi_i(0,j) \,|\, \pi_{LOC})\}$ 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.

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

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}

Theorem 2.

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

Proof. Suppose $\pi^* = \bigcirc_{i=1}^{i^*} \varpi_i(1, \tau_i)$ and $\varpi_i(\tau_i + 1, j) \in \pi_{LOC}$ is the first instruction bundle that π^* and π_{LOC} disagree.

Split π^* into two parts π_a^* and π_b^* where π_a^* is the sub-sequence from the beginning of π^* to instruction (pw_i, t_{τ_i}) inclusive and π_b^* is remaining checking sequence. Formally,

$$\pi_a^* := \bigcirc_{i'=1}^{i} \varpi_{i'}(1, \tau_{i'}),$$

and

$$\pi_b^* := \bigcirc_{i'=i+1}^{i^*} \varpi_{i'}(1, \tau_{i'}).$$

Since $\varpi_i(\tau_i + 1, j) \in \pi_{LOC}$, by criterion of constructing π_{LOC} in Algorithm 1 we have

$$\Delta^\circ\left(\varpi_i(\tau_i + 1, j) \,\middle|\, \pi_a^*\right) \ge 0.$$

We can contrive another checking sequence $\sigma = \pi_a^* \circ \varpi_i(\tau_i + 1, j) \circ \pi_b^*$. By Lemma 1 we have

$$\Delta^\circ\left(\pi_b^* \,\middle|\, \pi_a^* \circ \varpi_i(\tau_i + 1, j)\right) > \Delta^\circ\left(\pi_b^* \,\middle|\, \pi_a^*\right)$$

Adding both sides of above two inequalities, then both sides of the resulting inequality are added by $\Delta(\emptyset, \pi_a^*)$, we have

$$\begin{aligned} U_{adv}(v, \vec{q}, \sigma) &= \Delta(\emptyset, \pi_a^*) + \Delta^\circ\left(\varpi_i(\tau_i + 1, j) \,\middle|\, \pi_a^*\right) + \Delta^\circ\left(\pi_b^* \,\middle|\, \pi_a^* \circ \varpi_i(\tau_i + 1, j)\right) \\ &> \Delta(\emptyset, \pi_a^*) + \Delta^\circ\left(\pi_b^* \,\middle|\, \pi_a^*\right) \\ &= U_{adv}(v, \vec{q}, \pi^*), \end{aligned}$$

contracting optimality of π^*. □

From Theorem 1 and Theorem 2, we can derive the following corollaries.

Corollary 1.

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

and

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

where

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

Corollary 2.

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

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. Intuitively, When $q_i = \frac{1}{m}$ and $\beta_i = \sqrt{i}$, $\mathsf{ExtendbyConcat}(v, \vec{q}, \emptyset)$ always sets $\tau_i \in \{0, m\}$, leaving π_{LOC}—the subset of π^* and $\Pi(\mathsf{Len}_{max}, m)$—the superset of π^* to be the same. Thus, π_{LOC} is optimal.

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., $\pi_{LOC} = \pi^*$.*

Proof. Given these parameters it is easy to verify that

$$\begin{aligned} &\Delta^\circ\left(\varpi_i(1,1) \,\middle|\, \bigcirc_{i'=1}^{i-1} \varpi_i(1, \tau_{i'})\right) \\ &< \Delta^\circ\left(\varpi_i(2,2) \,\middle|\, \bigcirc_{i'=1}^{i-1} \varpi_i(1, \tau_{i'}) \circ \varpi_i(1,1)\right) \\ &< \cdots \\ &< \Delta^\circ\left(\varpi_i(m,m) \,\middle|\, \bigcirc_{i'=1}^{i-1} \varpi_i(1, \tau_{i'}) \circ \varpi_i(1, m-1)\right). \end{aligned}$$

Therefore,

$$\max_{0 \leq j \leq m} \left\{\Delta^\circ\left(\varpi_i(0,j) \,\middle|\, \bigcirc_{i=1}^{i-1} \varpi_i(1, \tau_i)\right)\right\} = \max\left\{0, \Delta^\circ\left(\varpi_i(1,m) \,\middle|\, \bigcirc_{i=1}^{i-1} \varpi_i(1, \tau_i)\right)\right\}$$

Algorithm $\mathsf{ExtendbyConcat}(v, \vec{q}, \emptyset)$ will set $\tau_i = m$ for $i \leq \mathsf{Len}(\pi_{LOC})$ and $\tau_i = 0$ for $i > \mathsf{Len}(\pi_{LOC})$. In other words,

$$\begin{cases} \Delta^\circ\left(\varpi_i(1,m) \,|\, \Pi(i-1,m)\right) \geq 0, \ \text{if } i \leq \mathsf{Len}(\pi_{LOC}), \\ \Delta^\circ\left(\varpi_i(1,m) \,|\, \Pi(i-1,m)\right) < 0, \ \text{if } i > \mathsf{Len}(\pi_{LOC}). \end{cases}$$

Those are also the criterion of defining Len_{max} under current parameter settings, hence $\mathsf{Len}(\pi_{LOC}) = \mathsf{Len}_{max}$. Moreover, the superset and subset of π^* are identical, i.e., $\Pi(\mathsf{Len}_{max}, m) = \pi_{LOC}$. Since there are no unchecked instruction bundle $\varpi_i(j_1, j_2)$ for $i \leq \mathsf{Len}_{max}$, we have $\pi_{LOC} = \pi^* = \Pi(\mathsf{Len}_{max}, m)$. □

Notice that if the breakpoints are cost-even and follow uniform distribution, the optimal checking sequence takes a simple form of $\pi^* = \Pi(\mathsf{Len}_{max}, m)$, i.e., checking all labels for each password until the password indexed Len_{max} is reached. By adopting these parameter setting, we ensure that our mechanism yields a lower fraction of cracked passwords compared to deterministic cost hashing. In the deterministic cost hashing, the cost of checking each password remains

constant (password hashes are computed using a predetermined label if a memory-hard function were employed). The attacker's optimal strategy for cracking would be checking a sequence of passwords in descending order of likelihood up to a cut point, at which point the attacker ceases the attempt (i.e., $\pi^d = pw_1, pw_2, ..., pw_d$). The following theorem formally highlights the advantage of our mechanism over traditional approach.

Theorem 4. *Let* AM_1 *(authentication mechanism) be cost-asymmetric memory hard password hashing with* $\beta_i = \sqrt{i}$ *and* $q_i = \frac{1}{m}$ *and* AM_2 *represent the traditional deterministic cost hashing where each password is hashed with cost* C_{max} *(which also equals to the expected hash cost of* AM_1 *per password for the purpose of fair comparison), Assuming optimal attacker strategy,* AM_1 *outperforms* AM_2 *in terms of attacker's success rate. Specifically, let* $\pi^* = \Pi(\mathsf{Len}_{max}, m)$ *and* $\pi^d = pw_1, pw_2, ..., pw_d$ *be the optimal strategy against* AM_1 *and* AM_2*, respectively, then we have* $\lambda(\pi^*) \leq \lambda(\pi^d)$ *where* $\lambda(\pi^d)$ *is defined analogously as* $\sum_{i=1}^{d} Pr[pw_i]$.

Proof. When using cost-even breakpoints with uniform distribution, the execution trace of the attacker's cracking process is similar to that of deterministic hashing, i.e., sequentially eliminating the possibility of $pw_u = pw_i$ (or verifying it with any luck) for $i = 1, 2, \ldots$. It can be verified (see Appendix .1) that

$$C^u_{adv}(\varpi_i(1,m)|\Pi(i-1,m)) \geq C^d_{adv}(pw_i|\bigcirc_{i'=1}^{i-1} pw_{i'}),\ \forall i,$$

where $C^u_{adv}(\varpi_i(1,m)|\Pi(i-1,m))$ is the marginal cost of checking $\varpi_i(1,m)$, given $\Pi(i-1,m)$ has already been checked. $C^d_{adv}(pw_i|\bigcirc_{i'=1}^{i-1} pw_{i'})$ is the marginal cost of checking pw_i under deterministic cost hashing given passwords $pw_1, \ldots, pw_{i-1}$ have been checked. In order to achieve the same success rate, uniform cost-even breakpoints would incur more cost than deterministic cost hashing. See it in another way, when v/C_{max} is fixed for both cases uniform cost-even breakpoints results in a lower adversary success rate. □

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 $O(n_p m)$, 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 $O(1)$. One pass of `repeat` loop of Algorithm 3 takes time $O(|\Pi(\mathsf{Len}_{max}, m)| - |\pi_{LOC}|) \subseteq O(n_p m)$, 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 $O(n_p m)$.

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

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

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

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

Input: $v, \vec{q}$
Output: π_{LO}

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

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

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

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

Proof. In proof of Theorem 2 we already know $\circ$ operation preserves the following invariant

$$\pi \circ e \subseteq \pi^*, \text{ if } \pi \subseteq \pi^* \text{ and } \Delta^\circ(e \,|\, \pi) \geq 0.$$

Lemma 2 states

$$\pi + e \subseteq \pi^*, \text{ if } \pi \subseteq \pi^* \text{ and } \Delta^+(e \,|\, \pi) \geq 0.$$

π_{LO} is obtained by alternatively applying $\circ$ and + operation, hence is a subset of π^* □

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

In the previous subsections, we designed a polynomial algorithm $\mathsf{Extend}(v, \vec{q})$ 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.

Since π_{LO} is locally optimal adding any instruction bundle for a single password $e = \varpi_i(j_1, j_2)$ into π_{LO} will decrease the overall utility, namely, $\Delta(e \,|\, \pi) < 0$, recall that $\Delta(e \,|\, \pi)$ denotes the marginal utility by including e into π, either through concatenation or insertion. However, there is no guarantee $\Delta(S \,|\, \pi_{LO}) < 0$ where S is an ordered set of instruction bundles $\{e_1, e_2, \ldots, e_b\}$ since marginal utility is not *additive* with respect to instruction bundles. To see this, from equation (6) we can derive

$$\Delta(S \,|\, \pi_{LO}) = \sum_{e \in S} \Delta(e \,|\, \pi_{LO}) + \sum_{e_2 \in S} \overbrace{c(e_2) \sum_{\substack{e_1 \in S \\ e_1 < e_2}} \Pr[e_1]}^{\text{cost reduction for } e_2}, \tag{13}$$

where $\Pr[e]$ and $c(e)$ are probability summation and round cost summation of instructions in e, respectively. Equation (13) shows that the marginal cost by including a ordered set S to π_{LO} stems from 2 parts. The first is the summation of individual contribution and the second is cost reduction when checking $e_2 \in S$ because previously included instruction bundles $e_1 < e_2, e_1 \in S$ have already eliminate some uncertainty. Even though every instruction bundle solely contributes negative marginal utility i.e, $\Delta(e \,|\, \pi_{LO}) < 0$, $e \in S$, the sign of $\Delta(S \,|\, \pi_{LO})$ is not determined because of the cost reduction term. If a set S exists such that $\Delta(S \,|\, \pi) \geq 0$, we will refer it to as a *good set*.

By definition of good set and Theorem 5, we have

$$\pi^* = \pi_{LO} \cup S^*, \text{ s.t. } S^* = \arg\max_{S} \Delta(S \,|\, \pi_{LO}). \tag{14}$$

Recall that $\cup$ denotes inclusion (whether through concatenation or insertion) while maintaining natural ordering. [4]

Verify if S is a good set of π_{LO} is easy but find one is hard. We design a polynomial algorithm to check if the local optimum π_{LO} is in fact a global optimum, i.e., $\pi_{LO} = \pi^*$. Our algorithm utilizes the following observation.

[4] Technically, $\arg\max_S$ returns a set of solutions S^*. However, if this set contains multiple elements we can break ties according to the size of $|S^*|$ and followed by an arbitrary lexicographic ordering over solutions with same size.

Observation 1. *if $S = \{e_1, \ldots, e_b\}$ is a good set for π_{LO}, then its last element e_b must provide non-negative utility (otherwise, it can be safely removed from S without hurting marginal utility), namely*

$$\Delta(e_b \,|\, \pi_{LO} \cup S \setminus e_b) \geq 0,$$

where $S \setminus e_b$ is the ordered set excluding e_b.

It is not clear which elements are inside $S \setminus e_b$ but we know $S \setminus e_b \subseteq \mathsf{Before}(e_b \,|\, \pi_{LO})$ where $\mathsf{Before}(e_b \,|\, \pi_{LO})$ is the ordered set of all unchecked instructions that appear before e_b in natural ordering, given π_{LO} already being checked, namely,

$$\mathsf{Before}(e_b \,|\, \pi_{LO}) := \{e : e < e_b \text{ and } e \notin \pi_{LO}\}.$$

We use following Lemma to negate the existence of a good set ending with e_b.

Lemma 3. *For a unchecked instruction bundle e_b, define*

$$\mathsf{test}(e_b) := \Delta(e_b \,|\, \pi_{LO}) + \sum_{e \in \mathsf{Before}(e_b \,|\, \pi_{LO})} \Pr[e] c(e_b),$$

if $\mathsf{test}(e_b) < 0$ then a good set S for π_{LO} ending with e_b does not exist.

Proof.

$$\begin{aligned}
\Delta(e_b \,|\, \pi_{LO} \cup S \setminus e_b) &= \Delta(e_b \,|\, \pi_{LO}) + \sum_{e \in S \setminus e_b} \Pr[e] c(e_b), \\
&\leq \Delta(e_b \,|\, \pi_{LO}) + \sum_{e \in \mathsf{Before}(e_b \,|\, \pi_{LO})} \Pr[e] c(e_b) \\
&= \mathsf{test}(e_b).
\end{aligned}$$

If $\mathsf{test}(e_b) < 0$, then $\Delta(e_b \,|\, \pi_{LO} \cup S \setminus e_b) < 0$. By contrapositive of Observation 1, S cannot be a good set. □

We can interpret the term $\sum_{e \in \mathsf{Before}(e_b \,|\, \pi_{LO})} \Pr[e] c(e_b)$ as the maximum possible cost reduction when checking e_b, then $\mathsf{test}(e_b)$ is the maximum marginal utility e_b can provide as the last instruction bundle in S. If $\mathsf{test}(e_b) < 0$, a set ending with e_b cannot be a good set; if this is the case for all instructions $e_b \in \Pi(\mathsf{Len}_{max}, m) \setminus \pi_{LO}$ that might be added to π_{LO}, then a good set ending with any unchecked instruction bundle does not exist. Thus, in Equation (14) we have $S^* = \emptyset$ and $\pi_{LO} = \pi^*$. We use $\mathsf{OptimalityTest}(v, \vec{q}, \pi_{LO})$ to examine if $\pi_{LO} = \pi^*$ — see Algorithm 4.

Algorithm 4: $\mathsf{OptimalityTest}(v, \vec{q}, \pi_{LO})$

Input: $v, \vec{q}, \pi_{LO}$
Output: π^*

```
foreach e_b ∈ Π(Len_max, m) \ π_LO do
    if test(e_b) ≥ 0 then
        return FAIL;
    end
end
return PASS
```

Theorem 6. *If* $\mathsf{OptimalityTest}(v, \vec{q}, \pi_{LO})$ *returns PASS, then* $\pi_{LO} = \pi^*$.

Proof. Since $\mathsf{OptimalityTest}(v, \vec{q}, \pi_{LO})$ returns PASS, then we have

$$\Delta\left(e_b \,|\, \pi_{LO} \cup S \setminus e_b\right) \leq \mathsf{test}(e_b) < 0, \ \forall S.$$

For any S it would only increase overall utility by repeatedly removing the last element in S until it is empty. Therefore, a good set is a empty set. Equivalently, $\pi_{LO} = \pi^*$. □

By storing of $\lambda(\pi_{LO}, i)$ and $\sum_{i'=1}^{i} \Pr[pw_{i'}]$ for all i, which are intermediate values in execution of $\mathsf{Extend}(v, \vec{q})$, in amortized sense we can evaluate $\mathsf{test}(e_b)$ in time $O(1)$, then algorithm $\mathsf{OptimalityTest}(v, \vec{q}, \pi_{LO})$ runs in time $O(n_p m)$. Even if π_{LO} fails $\mathsf{OptimalityTest}(v, \vec{q}, \pi_{LO})$, it does not imply $\pi_{LO} \neq \pi^*$, since $\mathsf{test}(e_b) < 0$ is a sufficient condition of no good set ending with e_b, not a necessary one. Fortunately, $\mathsf{OptimalityTest}(v, \vec{q}, \pi_{LO})$ returns PASS for most of the time in our experiments, confirming $\pi_{LO} = \pi^*$ for most v/C_{max} ratios; otherwise, we might discard $\mathsf{OptimalityTest}(v, \vec{q}, \pi_{LO})$ as well.

5.6. Finding π^* for cost-even breakpoints when $m \leq 3$

When our optimality test fails in rare cases, we design algorithms to promote locally optimal solution to globally optimal solution for cost-even breakpoints and $m \leq 3$.

If π_{LO} fails $\mathsf{OptimalityTest}(v, \vec{q}, \pi_{LO})$, we cannot deduce any conclusions about the optimality of π_{LO}, but when $\beta_i = \sqrt{i}$ and $m \leq 3$ we can design an efficient brute force algorithm to find π^*. We first define the concept of $\mathsf{peak}(\vec{q})$ and show that it is true that the stopping label indices $\tau_i \in \mathsf{peak}(\vec{q})$ for both π_{LO} (Lemma 4) and π^* (Lemma 5). Therefore, an instruction bundle e in a good set S can only the form of spanning two peaks, i.e., $e = \varpi_i(j_1, j_2)$ where $j_1, j_2 \in \mathsf{peak}(\vec{q})$, then we use this intuition to prune down the search space of good set S.

Definition 4. Given a vector of real numbers $\vec{q} = (q_1, \ldots, q_m)$ we call index j of q a *peak* of $\vec{q}$ if (1) $j = m$, or (2) $j = 1$ and $q_1 > q_2$, or (3) $q_{j-1} \leq q_j$ and $q_j > q_{j+1}$. We use $\mathsf{peak}(\vec{q})$ to denote the set of all peak indices in $\vec{q}$.

Lemma 4. *Fix an arbitrary breakpoint distribution* $\vec{q} = (q_1, \ldots, q_m)$. *Suppose that* $\beta_j = \sqrt{j}$ *for all* $j \leq m$ *and* $\pi_{LO} = \bigcirc_{i=1}^{\mathsf{Len}(\pi_{LO})} \varpi_i(1, \tau_i)$, *then for all* $i \leq \mathsf{Len}(\pi_{LO})$ *we have* $\tau_i \in \mathsf{peak}(\vec{q})$.

Proof. Suppose $\tau_i = j \neq m$, since $(pw_i, t_j) \in \pi_{LO}$, then

$$\Delta\left(\pi_{LO} - (pw_i, t_j), \pi_{LO}\right) \geq 0,$$

where $-$ is removal operation. Since j is not a peak, then we have $q_{j+1} \geq q_j$ which leads to

$$\Delta\left((pw_i, t_{j+1}), \pi_{LO}\right) > \Delta\left(\pi_{LO} - (pw_i, t_j), \pi_{LO}\right).$$

Therefore, $\Delta\left(\pi_{LO}, \pi_{LO} + (pw_i, t_{j+1})\right) > 0$. It is still profitable to advance a label for pw_i i.e., check (pw_i, t_{j+1}), so (pw_i, t_{j+1}) should have been included into π_{LO} in local search but in relity it is not. Contradiction. □

Lemma 5. *Fix an arbitrary breakpoint distribution* $\vec{q} = (q_1, \ldots, q_m)$. *Suppose that* $\beta_j = \sqrt{j}$ *for all* $j \leq m$ *and* $\pi^* = \bigcirc_{i=1}^{\mathsf{Len}(\pi^*)} \varpi_i(1, \tau_i^*)$, *then for all* $i \leq \mathsf{Len}(\pi^*)$ *we have* $\tau_i^* \in \mathsf{peak}(\vec{q})$.

Observation 2. *when* $m = 2$, $\mathsf{peak}(\vec{q})$ *is in* $\{\{2\}, \{1, 2\}\}$*; when* $m = 3$, $\mathsf{peak}(\vec{q})$ *is in* $\{\{3\}, \{1, 3\}, \{2, 3\}\}$*;*

$|\mathsf{peak}(\vec{q})| = 1$ corresponds to uniform breakpoint distribution for which $\mathsf{ExtendbyConcat}(v, \vec{q}, \emptyset)$ already gives the optimal checking sequence π^*.

Lemmas 4 and 5 imply that if a good set S *with respect to insertion* exists for π_{LO}, every instruction bundle in S must start at a peak position and end with another peak position. Specially, when $\mathsf{peak}(\vec{q}) = \{peak_1, m\}$, the only tentative insertion operation to promote locally optimal π_{LO} to global optimal π^* is to check a password to completion, i.e., change the largest label index τ_i from $peak_1$ to m.

The following theorem states that we can efficiently construct an ordered set $S_c(i, \pi)$ which provides largest marginal utility upon insertion (on the basis of π) than any other ordered set of the same size. Thus, we can efficiently search $S_c(i, \pi), \forall i$ instead of all subsets of $\mathsf{unchecked}(\pi)$ in order to find a good set.

Theorem 7. *If* $\mathsf{peak}(\vec{q}) = \{peak_1, m\}$ *and* $\beta_i = \sqrt{i}$ *for* $i \leq m$*, given* $\pi = \bigcirc_{i=1}^{\mathsf{Len}} \varpi_i(1, \tau_i)$ *with* $\tau_i \in \mathsf{peak}(\vec{q})$ *and* $\mathsf{unchecked}(\pi) = \{\varpi_i(peak_1, m) : i \leq \mathsf{Len}(\pi)$ *and* $\tau_i = peak_1\}$*—the set of unchecked instruction bundles spanning two peaks, we define*

$$S_c(i, \pi) := \begin{cases} \emptyset, \text{ if } i = 0, \\ \{e_1, \ldots, e_i\}, \text{ if } i > 0, \end{cases}$$

where e_i, $\forall i > 0$ *is recursively defined as*

$$e_i := \underset{e \in \mathsf{unchecked}(\pi + S_c(i-1, \pi))}{\arg\max} \Delta^+ \left(e \,|\, \pi + S_c(i-1, \pi)\right).$$

Then we have

$$\Delta^+ \left(S_c(i, \pi) \,|\, \pi\right) \geq \Delta^+ \left(S \,|\, \pi\right), \ \forall i, \ \forall S \subseteq \mathsf{unchecked}(\pi) \text{ s.t. } |S_c(i, \pi)| = |S|.$$

We first prove the following Lemma, which is utilized in proof of Theorem 7.

Lemma 6. *When* $\beta_i = \sqrt{i}$*, suppose* $e_1 = \varpi_{i1}(j_1, j_2)$*, and* $e_2 = \varpi_{i_2}(j_1, j_2)$*, if* $\Delta^+ (e_1 \,|\, \pi) \geq \Delta^+ (e_2 \,|\, \pi)$*, then* $\Pr(e_1) \geq \Pr(e_2)$*.*

Proof. Proof by contradiction. Let $c_M(t_{j_2}^2 - t_{j_1}^2) = c$. Suppose $\Pr(e_1) < \Pr(e_2)$, then $e_1 > e_2$, we have

$$\begin{aligned}
&\Delta^+ (e_1 \,|\, \pi) - \Delta^+ (e_2 \,|\, \pi) \\
&= \Pr(e_1) \left(v + \sum_{e > e_1, e \in \pi} c \right) - \left(1 - \sum_{e < e_1, e \in \pi} \Pr(e) \right) c - \Pr(e_2) \left(v + \sum_{e > e_2, e \in \pi} c \right) + \left(1 - \sum_{e < e_2, e \in \pi} \Pr(e) \right) c \\
&= (\Pr(e_1) - \Pr(e_2)) \left(v + \sum_{e > e_1, e \in \pi} c \right) - \Pr(e_2) \sum_{e_2 < e \leq e_1, e \in \pi} c + c \sum_{e_2 \leq e < e_1, e \in \pi} \Pr(e) \\
&= (\Pr(e_1) - \Pr(e_2)) \left(v + \sum_{e > e_1, e \in \pi} c \right) + c \sum_{e_2 < e < e_1} \Pr(e) - \Pr(e_2) \\
&\leq (\Pr(e_1) - \Pr(e_2)) \left(v + \sum_{e > e_1, e \in \pi} c \right) < 0,
\end{aligned}$$

which contradicts the precondition $\Delta^+(e_1 \,|\, \pi) \geq \Delta^+(e_2 \,|\, \pi)$ □

Now we prove Theorem 7.

Proof. In this proof $S_c(i, \pi)$ is written in $S_c(i)$ for simplicity. Let $S = \{e'_1, e'_2, \ldots, e'_i\}$. Lemma 4 and 5 guarantee that instruction bundles in S and $S_c(i)$ have the same size, i.e, $|e_j| = |e'_k|, \forall j, k$. Let $c(e) = c, \forall e \in S_c(i),\ \forall e \in S$. By Theorem 7 and Lemma 6 we have $\Pr(e_j) \geq \Pr(e'_j),\ \forall j$.

Theorem 7 equivalently claims for fixed i and π,

$$\Delta^+(S_c(i) \,|\, \pi) \geq \Delta^+\left(S_c(i-n) \cup \{e'_{i-n+1}, \ldots, e'_i\} \,\middle|\, \pi\right), \forall n, \tag{15}$$

We will use mathematical induction to prove the above inequalities.

Base case $n = 1$:

$$\begin{aligned}
&\Delta^+(S_c(i) \,|\, \pi) \\
&= \Delta^+(S_c(i-1) \,|\, \pi) + \Delta^+(e_i \,|\, \pi + S_c(i-1)) \\
&\geq \Delta^+(S_c(i-1) \,|\, \pi) + \Delta^+(e'_i \,|\, \pi + S_c(i-1)) \\
&= \Delta^+(S_c(i-1) \cup \{e'_i\} \,|\, \pi).
\end{aligned}$$

The inequality holds because of the definition of e_i i.e., $e_i = \arg\max_e \Delta^+(e \,|\, \pi + S_c(i-1))$.

Inductive hypothesis: equation (15) holds true for $n = k$.

Inductive step: in the following we prove that equation (15) holds true for $n = k + 1$.

$$\begin{aligned}
&\Delta^+(S_c(i) \,|\, \pi) \\
&\geq \Delta^+\left(S_c(i-k) \cup \{e'_{i-k+1}, \ldots, e'_i\} \,\middle|\, \pi\right) \\
&= \Delta^+(S_c(i-k-1) \,|\, \pi) + \Delta^+(e_k \,|\, \pi + S_c(i-k-1)) + \Delta^+\left(\{e'_{i-k+1}, \ldots, e'_i\} \,\middle|\, \pi + S_c(i-k-1) + e_k\right) \\
&\geq \Delta^+(S_c(i-k-1) \,|\, \pi) + \Delta^+\left(e'_k \,\middle|\, \pi + S_c(i-k-1)\right) + \Delta^+\left(\{e'_{i-k+1}, \ldots, e'_i\} \,\middle|\, \pi + S_c(i-k-1) + e_k\right) \\
&\geq \Delta^+(S_c(i-k-1) \,|\, \pi) + \Delta^+\left(e'_k \,\middle|\, \pi + S_c(i-k-1)\right) + \Delta^+\left(\{e'_{i-k+1}, \ldots, e'_i\} \,\middle|\, \pi + S_c(i-k-1) + e'_k\right) \\
&= \Delta^+\left(S_c(i-k-1) \cup \{e'_{i-k}, \ldots, e'_i\} \,\middle|\, \pi\right).
\end{aligned}$$

The first inequality is inductive hypothesis; the second inequality holds because of the definition of e_k; the third inequality is the result of Lemma 1 given $\Pr[e_k] \geq \Pr[e'_k]$.

□

We can loop over all candidates $S_c(i, \pi_{LO}), \forall i$ and find the *good set w.r.t insertion* $S^*_c(\pi_{LO})$, which provides the largest marginal utility for π_{LO}, namely,

$$S^*_c(\pi_{LO}) = \underset{S_c(i,\pi_{LO})}{\arg\max}\, \Delta^+(S_c(i, \pi_{LO}) \,|\, \pi_{LO}). \tag{16}$$

A good set S^* might contain instruction bundles that can be concatenated to π_{LO}, to handle this case we also need to loop over *len*—possible length of π^*. The efficient brute force algorithm $\mathsf{FindOptSeq}(v, \vec{q}, \pi)$ is present in Algorithm 5.

Suppose $|\mathsf{unchecked}(\pi)| = n$, finding e_1 takes time $O(n)$; finding e_2 takes time $O(n-1)$, etc. Thus, the inner loop in Algorithm 5 takes time $O(n^2) \subseteq O(n_p^2)$, and the total running time is $O(n_p^2\,(\mathsf{Len}_{max} - \mathsf{Len}(\pi))$.

Algorithm 5: FindOptSeq$(v, \vec{q}, \pi)$

Input: $v, \vec{q}, \pi$
Output: π^*

$U^* = \Delta(\emptyset, \pi)$;
for $len = \mathsf{Len}(\pi) : \mathsf{Len}_{max}$ **do**
  if $len > \mathsf{Len}(\pi)$ **then**
    $\pi = \pi \circ \varpi_{len}(1, peak_1)$;
  end
  foreach i **do**
    construct $S_c(i, \pi)$;
    compute $\Delta^+(S_c(i, \pi \mid \pi)$;
  end
  $S_c^*(\pi) = \arg\max_i \Delta^+(S_c(i, \pi \mid \pi)$;
  if $\Delta(\emptyset, \pi + S_c^*(\pi)) \geq U^*$ **then**
    $\pi^* = \pi + S_c^*(\pi)$;
    $U^* = \Delta(\emptyset, \pi + S_c^*(\pi))$;
  end
end
return π^*

Theorem 8. *When* $|\mathsf{peak}(\vec{q})| = 2$ *and* $\beta_i = \sqrt{i}$, FindOptSeq$(v, \vec{q}, \pi_{LO})$ *returns the optimal checking sequence.*

Proof. Lemma 4 and 5 restrict a good set to be a subset of $\mathsf{unchecked}(\pi)$. Theorem 7 guarantees that $S_c(i, \pi)$ is "better" than any other set S of the same size, namely,

$$\Delta(\emptyset, \pi + S_c(i, \pi)) \geq \Delta(\emptyset, \pi + S), \forall S \subseteq \mathsf{unchecked}(\pi).$$

By definition of $S_c^*(\pi)$, we have $\Delta(\emptyset, \pi + S_c^*(\pi)) \geq \Delta(\emptyset, \pi + S_c(i, \pi)), \forall i$. The outer loop of FindOptSeq$(v, \vec{q}, \pi)$ traverses all possible $\mathsf{Len}(\pi^*)$ and returns the $\pi + S_c^*(\pi)$ with largest utility. By equation (14) the the returned checking sequence is optimal. □

We could potentially run FindOptSeq$(v, \vec{q}, \emptyset)$ to find the optimal checking sequence π^*. As a shortcut, we run FindOptSeq$(v, \vec{q}, \pi_{LO})$ instead to reduce the running time.

Corollary 3. *When* $m \leq 3$ *and* $\beta_i = \sqrt{i}$, *There are polynomial algorithms that always find the optimal checking sequence.*

If $|\mathsf{peak}(\vec{q})| = 1$, ExtendbyConcat$(v, \vec{q}, \emptyset)$ returns the optimal checking sequence; if $|\mathsf{peak}(\vec{q})| = 2$, FindOptSeq$(v, \vec{q}, \pi_{LO})$ returns the optimal checking sequence.

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

$$
\begin{aligned}
\min_{\vec{q}} \quad & \lambda(\pi^*) \\
\text{s.t.} \quad & 0 \le q_i \le 1, \ \forall 1 \le i \le m, \\
& \sum_{i=1}^{m} q_i = 1, \\
& \sum_{i=1}^{m} q_i c_M t_i^2 \le C_{max}, \\
& \pi^* = \arg\max U_{adv}(v, \vec{q}, \pi))
\end{aligned}
\tag{17}
$$

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 a heuristic black box optimization solver [36] to optimize $\vec{q}$. We refer to the black box solver as FindOptDis(). This heuristic algorithm is parameterized 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 $\vec{q}^*$.

In the following, we present two examples illustrating the potential issues that may arise when setting breakpoints $\{t_i\}$ to be time-even, emphasizing the advantages of adopting our suggested cost-even breakpoints.

6.1. Example 1: Zipf's Law with Time-Even Breakpoints

Boyen [9] proposed that when the user selects the running time parameter the resulting distribution will follow Zipf's law i.e., $\Pr[t_u = i] = c \cdot i^{-\beta}$ for some Zipf Law parameters $c, \beta > 0$. Boyen's analysis [9] indicated that setting $\beta = 1$ maximizes the attacker's expected workload ratio i.e., the work performed by a persistent attacker who does not know the secret running time parameter divided by the work performed by a persistent attacker who does know t_u. However, there are several important differences between Boyen's model and the rational attacker model we use for our analysis. The most significant difference is that Boyen's considers a persistent attacker who will always continue guessing until the password is cracked[5]. By contrast, we consider a rational attacker who might choose to partially check or even completely ignore a password if the expected cost outweights the expected reward. Similarly, our optimization goal for the defender is to directly minimize the probability that a rational attacker cracks the user's password. Thus, it is natural to ask whether or not Zipf's Law is still an appropriate distribution in our context. We argue that the answer is no.

In this section we provide theoretical analysis demonstrating that Zipf's Law with time-even breakpoints can significantly increase the fraction of cracked passwords cracked by a rational attacker. Fix Zipf's law constants $\beta = 1$ and $c = \sum_{i=1}^{t_{max}} i^{-\beta}$ and let $q_i \doteq i^{-\beta} c$ be the probability of selecting $t = i$ as the running time parameter. Observe that $\sum_{i=1}^{t_{max}} q_i = 1$ so that we have

[5]There are several other differences. Boyen's analysis [9] implicitly assumes a uniform distribution over passwords while our model allows for non-uniform password distributions. Our model includes a parameter for the value of a cracked password since we are considering rational attackers. Our model also captures the quadratic cost scaling in the area-time complexity of an ideal memory-hard function while Boyen's analysis assumes that costs scale linearly in the running time parameter.

a valid probability distribution and that the expected workload for the authentication server is $C_{max} \doteq \sum_{i=1}^{t_{max}} \frac{1}{i \cdot c} c_M i^2 = \frac{1}{c} \sum_{i=1}^{t_{max}} i c_M \approx \frac{c_M t_{max}^2}{2 \ln t_{max}}$.

Our analysis is centered around the following observation: If the breakpoints are time-even, for any $t' \leq t_{max}$ we have $\Pr[t \leq t'] = \frac{\sum_{i=1}^{t'} \frac{1}{i}}{\sum_{i=1}^{t_{max}} \frac{1}{i}} \approx \frac{\ln t'}{\ln t_{max}}$. If, for example, we set $t' = t_{max}/\ln t_{max}$ then as $t_{max} \to \infty$ we have $\Pr[t \geq t'] \to 0$ i.e., if t_{max} is sufficiently large then it is almost certain that $t \leq t_{max}/\ln t_{max}$. This suggests that the a rational attacker could benefit by adopting a strategy where s/he checks every password up to breakpoint $t' \approx t_{max}/\ln t_{max}$. Observe that the attacker's expected work to check any password up to breakpoint t is *upper bounded* by $t'^2 c_M = \frac{c_M t_{max}^2}{\ln^2 t_{max}} \approx \frac{2C_{max}}{\ln t_{max}}$. Thus, as $t_{max} \to \infty$ we have $\frac{t'^2 c_M}{C_{max}} \to 0$ i.e., the attacker's work per password guess is arbitrarily smaller than C_{max}.

To provide a concrete example suppose that the distribution over passwords is uniform over the range $[1, 10^4]$ and the value of a cracked password is $v = 100 \cdot C_{max}$. If the defender simply uses a deterministic MHF with cost C_{max} then the value is small enough that a rational attacker will crack 0% of passwords since expected guessing costs will greatly exceed the expected reward i.e., if the attacker continues guessing until the password is cracked the expected costs will be $C_{max} \sum_{i=1}^{10^4} i \cdot 10^{-4} > 5v$. By contrast, if uniform time-even breakpoints were used, we claim that a rational attacker will crack nearly 100% of passwords as the parameter t_{max} grows large. To see this we compute the utility of a (possibly sub-optimal) attacker who checks every password up to breakpoint $t' = t_{max}/\ln t_{max}$ is *at least* $v \Pr[t \leq t'] - 10^4 t'^2 c_M$. We have $\Pr[t \leq t'] \approx 1$ and $10^4 t'^2 c_M \approx \frac{2 \cdot 10^4 C_{max}}{\ln t_{max}} = \frac{200v}{\ln t_{max}}$. In particular, the utility of our attacker approaches v as $t_{max} \to \infty$ grows large. This is significant because v upper bounds the maximum possibly utility of any attack i.e., this would be the utility of an attacker who always cracks the password and somehow incurs no guessing cost. We have not shown that the attacker's strategy above (check every password up to breakpoint $t' \approx t_{max}/\ln t_{max}$) is optimal. However, the expected utility of optimal strategy can only be larger i.e., the expected utility of the optimal attack also approaches v as t_{max} grow large. It follows that the optimal rational attacker will crack the user's password with probability approaching 1!

6.2. Example 2: Uniform Distribution with Time-Even Breakpoints

We also provide another (admittedly contrived) example to show that time-even breakpoints could still be harmful even if the distribution over breakpoints is uniform. In particular, we assume a uniform distribution over two passwords and assume that the value of a password is $v = 1.45 C_{max}$.

Deterministic MHF. If the authentication server adopts a deterministic MHF with cost C_{max} then the rational attacker's optimal strategy to maximize expected utility is simply to give up immediately. If instead the attacker decided to keep guessing until he cracks the password the attackers expected utility would be negative i.e., the expected guessing costs would be $\frac{1}{2} C_{max} - \frac{1}{2} 2C_{max} = 1.5 C_{max} > v$ and utility would be $v - 1.5 C_{max} = -0.05 C_{max} < 0$. Thus, the optimal strategy for our utility maximizing attacker is simply to give up immediately.

Time-Even Breakpoints. If we use time-even breakpoints with uniform distribution then the hash cost of evaluating MHF to the first label is $c_M t_1^2 = 0.4 C_{max}$ and the hash cost of evaluating MHF to the second label is $c_M t_2^2 = c_M (2t_1)^2 = 1.6 C_{max}$ with the amortized cost being $\frac{0.4+1.6}{2} C_{max} = C_{max}$. Checking the first label of both passwords gives the attacker utility $v/2 - c_M t_1^2 - (1 -$

$\frac{1}{4})c_M t_1^2 = 0.025 C_{max} > 0$. Thus, a rational attacker will always want to check the first label of both passwords. As a result, the probability of the user's password being cracked is $\geq \frac{1}{4}$.

More generally, if time-even breakpoints with uniform distribution is deployed, we have $\frac{\Pr[pw_u]\Pr[t1]}{\Pr[pw_u]} = \frac{1}{m}$ while $\frac{c_M t_1^2}{C_{max}} = \frac{1}{(m+1)(2m+1)} \approx \frac{1}{m^2}$. In other words, the probability that the the first label is correct drops linearly in m while the cost of making that guess drops quadratically in m. This makes checking (at least) the first label of a password guess an increasingly more attractive target for a rational attacker.

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). We defer results for 6 additional password datasets (Bfield, Brazzers, Clicksense, CSDN, RockYou and Webhost) to Appendix .3. [6]

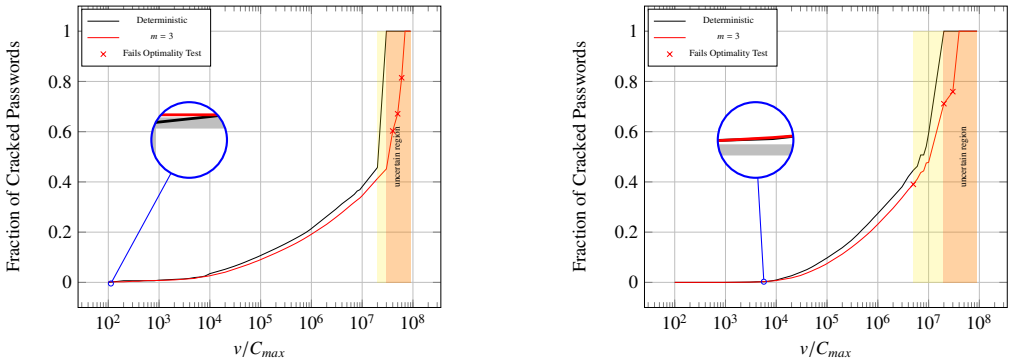
For each dataset we derive the corresponding empirical distribution $\mathcal{D}_e$ (namely, $\Pr_{pw \sim \mathcal{D}_e}[pw] = f_i/n_a$ where f_i is the frequency of pw) and analyze the attacker's success rate under this password distribution. The drawback is that the tail of empirical distribution $\mathcal{D}_e$ can significantly diverge from real distribution $\mathcal{P}$. We follow the approach of [37] and use Good-Turing Frequency estimation to uppe bound 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 Figure 3 we consider time-even breakpoints with uniform distribution over breakpoints. Similarly, Figure 4 considers cost-even breakpoints under the uniform distribution as the number of breakpoints m varies. In Figure 5, we fix $m = 3$ and continue to use cost-even breakpoints, then run our algorithm FindOptDis() (implemented with BITEOPT [36]), to optimize the breakpoint distribution.

7.2. Experiment Analysis and Discussion

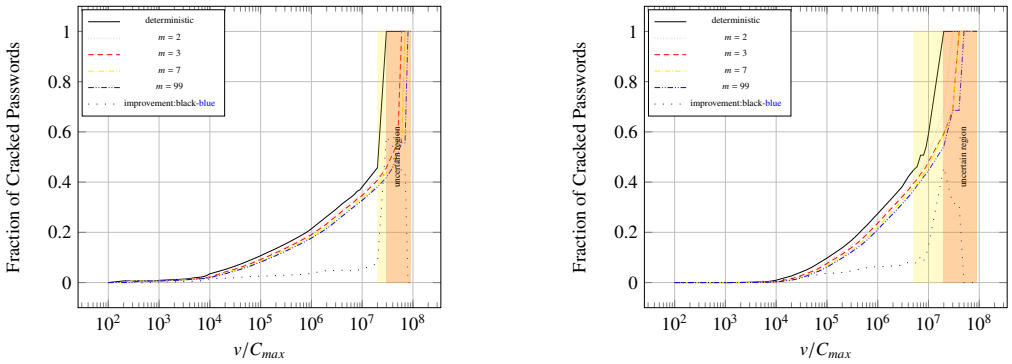
Time-Even Breakpoints with Uniform Distribution. Figure 3 plots the attacker's success rate (vs. v/C_{max}) when we use time-even breakpoints with the uniform distribution. In most parameter ranges the usage of time-even breakpoints with the uniform distribution reduces the % of cracked passwords in 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) time-even breakpoints with the uniform distribution can actually increase the fraction of cracked passwords. Take LinkedIn as example, when $v/C_{max} = 100$ no passwords would be cracked with deterministic cost hashing while 0.2% would be cracked using time-even breakpoints. Similar phenomenon can be observed in other datasets. Intuitively, these findings are explained by the observation that it is relatively cheap for the attacker to check the first few time-even breakpoints.

[6] 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.



(a) Linkedin (b) Neopets

Figure 3: Time-Even Breakpoints, Uniform Breakpoint Distribution

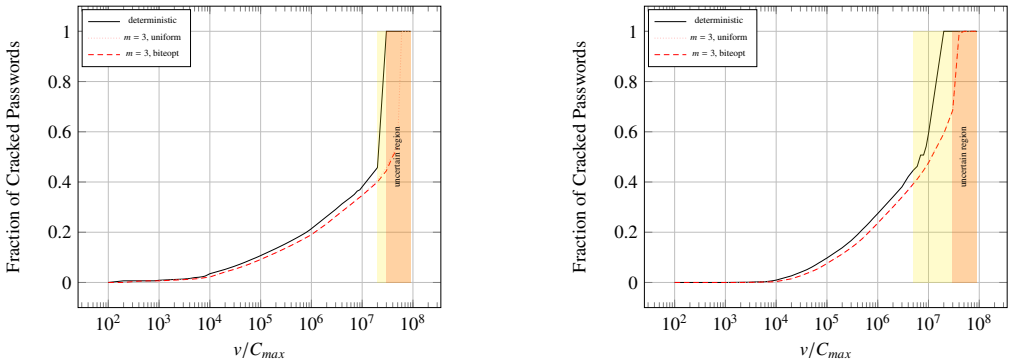


(a) Linkedin (b) Neopets

Figure 4: Cost-Even Breakpoints, Uniform Breakpoint Distribution

Cost-Even Breakpoints and Uniform Distribution. Figure 4 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 Figure 4 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 diminishes as m increases — see [38] for additional discussion. When $m = 99$ we find instances where the attacker's success rate is decreased by an additive factor of 10% for large value of v/C_{max} while the advantage is less significant for small values of v/C_{max}.

Optimized Distribution and Cost-Even Breakpoints. Continuing to use cost-even breakpoints we attempted to optimize the breakpoint distribution using BITEOPT [36] — see Figure 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



(a) Linkedin (b) Neopets

Figure 5: Cost-Even Breakpoints, Optimized Breakpoint Distribution

practice there is a very real risk that we would optimize $\vec{q}$ with respect to 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%.

References

[1] J. Bonneau, C. Herley, P. C. van Oorschot, F. Stajano, The quest to replace passwords: A framework for comparative evaluation of web authentication schemes, in: 2012 IEEE Symposium on Security and Privacy, IEEE Computer Society Press, San Francisco, CA, USA, 2012, pp. 553–567. doi:10.1109/SP.2012.44.

[2] B. Kaliski, Pkcs# 5: Password-based cryptography specification version 2.0 (2000).

[3] N. Provos, D. Mazieres, Bcrypt algorithm, USENIX, 1999.

[4] J. Blocki, B. Harsha, S. Zhou, On the economics of offline password cracking, in: 2018 IEEE Symposium on Security and Privacy, IEEE Computer Society Press, San Francisco, CA, USA, 2018, pp. 853–871. doi:10.1109/SP.2018.00009.

[5] C. Percival, Stronger key derivation via sequential memory-hard functions, in: BSDCan 2009, 2009.

[6] Password hashing competition, https://password-hashing.net/.

[7] J. Alwen, J. Blocki, B. Harsha, Practical graphs for optimal side-channel resistant memory-hard functions, in: B. M. Thuraisingham, D. Evans, T. Malkin, D. Xu (Eds.), ACM CCS 2017: 24th Conference on Computer and Communications Security, ACM Press, Dallas, TX, USA, 2017, pp. 1001–1017. doi:10.1145/3133956.3134031.

[8] U. Manber, A simple scheme to make passwords based on one-way functions much harder to crack, Computers & Security 15 (2) (1996) 171–176.
[9] X. Boyen, Halting password puzzles: Hard-to-break encryption from human-memorable keys, in: N. Provos (Ed.), USENIX Security 2007: 16th USENIX Security Symposium, USENIX Association, Boston, MA, USA, 2007.
[10] J. Blocki, A. Datta, CASH: A cost asymmetric secure hash algorithm for optimal password protection, in: IEEE 29th Computer Security Foundations Symposium, 2016, pp. 371–386.
[11] R. Morris, K. Thompson, Password security: A case history, Communications of the ACM 22 (11) (1979) 594–597.
[12] J. Bonneau, The science of guessing: Analyzing an anonymized corpus of 70 million passwords, in: 2012 IEEE Symposium on Security and Privacy, IEEE Computer Society Press, San Francisco, CA, USA, 2012, pp. 538–552. doi:10.1109/SP.2012.49.
[13] J. Campbell, W. Ma, D. Kleeman, Impact of restrictive composition policy on user password choices, Behaviour & Information Technology 30 (3) (2011) 379–388.
[14] B. Ur, P. G. Kelley, S. Komanduri, J. Lee, M. Maass, M. Mazurek, T. Passaro, R. Shay, T. Vidas, L. Bauer, N. Christin, L. F. Cranor, How does your password measure up? the effect of strength meters on password creation, in: Proceedings of USENIX Security Symposium, 2012.
[15] X. Carnavalet, M. Mannan, From very weak to very strong: Analyzing password-strength meters, in: ISOC Network and Distributed System Security Symposium – NDSS 2014, The Internet Society, San Diego, CA, USA, 2014.
[16] P. G. Inglesant, M. A. Sasse, The true cost of unusable password policies: Password use in the wild, in: Proceedings of the SIGCHI Conference on Human Factors in Computing Systems, CHI '10, ACM, New York, NY, USA, 2010, pp. 383–392. doi:10.1145/1753326.1753384.
URL http://doi.acm.org/10.1145/1753326.1753384
[17] M. Steves, D. Chisnell, A. Sasse, K. Krol, M. Theofanos, H. Wald, Report: Authentication diary study, Tech. Rep. NISTIR 7983, National Institute of Standards and Technology (NIST) (2014).
[18] D. Florêncio, C. Herley, P. C. Van Oorschot, An administrator's guide to Internet password research, in: Proceedings of the 28th USENIX Conference on Large Installation System Administration, LISA'14, 2014, pp. 35–52.
[19] A. Adams, M. A. Sasse, Users are not the enemy, Communications of the ACM 42 (12) (1999) 40–46.
[20] J. Blocki, S. Komanduri, A. Procaccia, O. Sheffet, Optimizing password composition policies, in: Proceedings of the fourteenth ACM conference on Electronic commerce, ACM, 2013, pp. 105–122.
[21] S. Komanduri, R. Shay, P. G. Kelley, M. L. Mazurek, L. Bauer, N. Christin, L. F. Cranor, S. Egelman, Of passwords and people: measuring the effect of password-composition policies, in: CHI, 2011, pp. 2595–2604.
URL http://dl.acm.org/citation.cfm?id=1979321
[22] D. Boneh, H. Corrigan-Gibbs, S. E. Schechter, Balloon hashing: A memory-hard function providing provable protection against sequential attacks, in: J. H. Cheon, T. Takagi (Eds.), Advances in Cryptology – ASIACRYPT 2016, Part I, Vol. 10031 of Lecture Notes in Computer Science, Springer, Heidelberg, Germany, Hanoi, Vietnam, 2016, pp. 220–248. doi:10.1007/978-3-662-53887-6_8.
[23] A. Biryukov, D. Dinu, D. Khovratovich, Argon2: new generation of memory-hard functions for password hashing and other applications, in: Security and Privacy (EuroS&P), 2016 IEEE European Symposium on, IEEE, 2016, pp. 292–302.
[24] J. Alwen, B. Chen, K. Pietrzak, L. Reyzin, S. Tessaro, Scrypt is maximally memory-hard, in: J. Coron, J. B. Nielsen (Eds.), Advances in Cryptology – EUROCRYPT 2017, Part III, Vol. 10212 of Lecture Notes in Computer Science, Springer, Heidelberg, Germany, Paris, France, 2017, pp. 33–62. doi:10.1007/978-3-319-56617-7_2.
[25] J. Alwen, J. Blocki, Efficiently computing data-independent memory-hard functions, in: M. Robshaw, J. Katz (Eds.), Advances in Cryptology – CRYPTO 2016, Part II, Vol. 9815 of Lecture Notes in Computer Science, Springer, Heidelberg, Germany, Santa Barbara, CA, USA, 2016, pp. 241–271. doi:10.1007/978-3-662-53008-5_9.
[26] J. Alwen, J. Blocki, K. Pietrzak, Depth-robust graphs and their cumulative memory complexity, in: J. Coron, J. B. Nielsen (Eds.), Advances in Cryptology – EUROCRYPT 2017, Part III, Vol. 10212 of Lecture Notes in Computer Science, Springer, Heidelberg, Germany, Paris, France, 2017, pp. 3–32. doi:10.1007/978-3-319-56617-7_1.
[27] M. H. Ameri, J. Blocki, S. Zhou, Computationally data-independent memory hard functions, ArXiv abs/1911.06790 (2020).
[28] W. Melicher, B. Ur, S. M. Segreti, S. Komanduri, L. Bauer, N. Christin, L. F. Cranor, Fast, lean, and accurate: Modeling password guessability using neural networks, in: T. Holz, S. Savage (Eds.), USENIX Security 2016: 25th USENIX Security Symposium, USENIX Association, Austin, TX, USA, 2016, pp. 175–191.
[29] C. Castelluccia, M. Dürmuth, D. Perito, Adaptive password-strength meters from Markov models, in: ISOC Network and Distributed System Security Symposium – NDSS 2012, The Internet Society, San Diego, CA, USA, 2012.
[30] B. Ur, S. M. Segreti, L. Bauer, N. Christin, L. F. Cranor, S. Komanduri, D. Kurilova, M. L. Mazurek, W. Melicher, R. Shay, Measuring real-world accuracies and biases in modeling password guessability, in: J. Jung, T. Holz (Eds.),

USENIX Security 2015: 24th USENIX Security Symposium, USENIX Association, Washington, DC, USA, 2015, pp. 463–481.
[31] M. Weir, S. Aggarwal, B. de Medeiros, B. Glodek, Password cracking using probabilistic context-free grammars, in: 2009 IEEE Symposium on Security and Privacy, IEEE Computer Society Press, Oakland, CA, USA, 2009, pp. 391–405. doi:10.1109/SP.2009.8.
[32] P. G. Kelley, S. Komanduri, M. L. Mazurek, R. Shay, T. Vidas, L. Bauer, N. Christin, L. F. Cranor, J. Lopez, Guess again (and again and again): Measuring password strength by simulating password-cracking algorithms, in: 2012 IEEE Symposium on Security and Privacy, IEEE Computer Society Press, San Francisco, CA, USA, 2012, pp. 523–537. doi:10.1109/SP.2012.38.
[33] R. Veras, C. Collins, J. Thorpe, On semantic patterns of passwords and their security impact, in: ISOC Network and Distributed System Security Symposium – NDSS 2014, The Internet Society, San Diego, CA, USA, 2014.
[34] M. Fossi, E. Johnson, D. Turner, T. Mack, J. Blackbird, D. McKinney, M. K. Low, T. Adams, M. P. Laucht, J. Gough, Symantec report on the underground economyRetrieved 1/8/2013. (November 2008).
[35] M. Stockley, What your hacked account is worth on the dark web (Aug 2016).
URL https://nakedsecurity.sophos.com/2016/08/09/what-your-hacked-account-is-worth-on-the-dark-web/
[36] A. Vaneev, BITEOPT - Derivative-free optimization method, Available at https://github.com/avaneev/biteopt, c++ source code, with description and examples (2021).
[37] W. Bai, J. Blocki, Dahash: Distribution aware tuning of password hashing costs, in: Financial Cryptography and Data Security, Springer International Publishing, 2021.
[38] W. Bai, J. Blocki, M. H. Ameri, Cost-asymmetric memory hard password hashing (2022).
URL https://arxiv.org/abs/2206.12970
[39] N. Hansen, The cma evolution strategy: A comparing review (2006).

Appendix .1. Marginal Cost

When breakpoints are cost-even and follow uniform distribution, let $\sum_{i'=1}^{i-1} \Pr[pw_i] = \lambda$, $c_M(t_i^2 - t_{i-1}^2) = c$, then the marginal cost of checking all labels associated with pw_i is

$$\begin{aligned} C^u_{adv}(\varpi_i(1,m)|\Pi(i-1,m)) &= (1-\lambda)c + \left(1-\lambda-\Pr[pw_i]\frac{1}{m}\right)c + \ldots, + \left(1-\lambda-\Pr[pw_i]\frac{m-1}{m}\right)c \\ &= (1-\lambda)mc - \frac{\Pr[pw_i]c(m-1)}{2}. \end{aligned} \tag{.1}$$

Due to server workload constraint we have,

$$\sum_{i=1}^{m} \beta_i^2 c/m = \frac{m+1}{2}c = C_{max},$$

which leads to $c = \frac{2C_{max}}{m+1}$. Substitute it into equation (.1), we have

$$C^u_{adv}(\varpi_i(1,m)|\Pi(i-1,m)) = \frac{2m}{m+1}(1-\lambda)C_{max} - \frac{(m-1)\Pr[pw_i]C_{max}}{m+1}. \tag{.2}$$

On the other hand, for deterministic hashing the marginal cost of checking password pw_i (given all previous passwords $pw_{i'}$, $i' < i$ have been checked) is

$$C^d_{adv}(pw_i|\bigcirc_{i'=1}^{i-1} pw_{i'}) = (1-\lambda)C_{max}. \tag{.3}$$

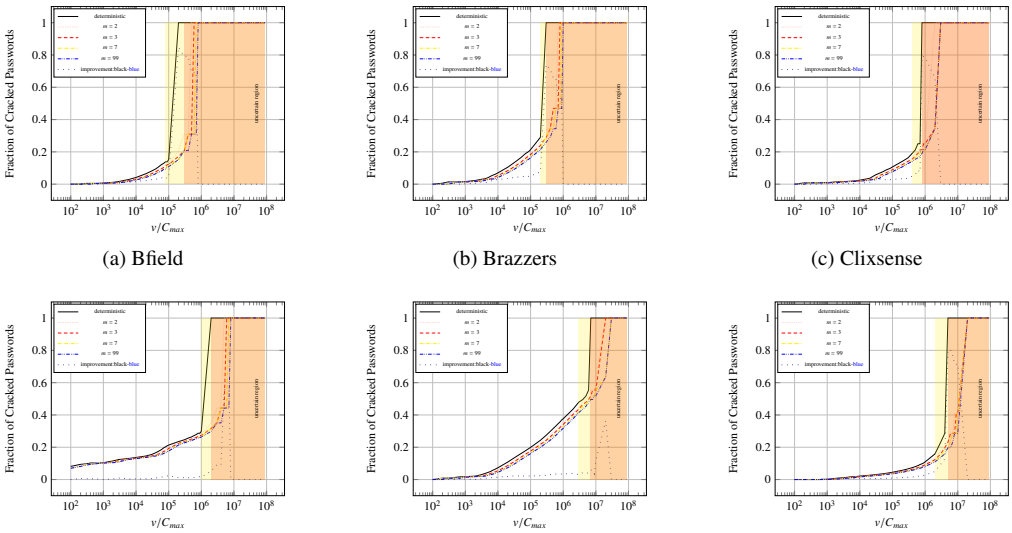
Take the difference of equation (.2) and equation (.3),

$$C^u_{adv}(\varpi_i(1,m)|\Pi(i-1,m)) - C^d_{adv}(pw_i|\bigcirc_{i'=1}^{i-1} pw_{i'}) = \frac{m-1}{m+1}C_{max}\left((1-\lambda) - \Pr[pw_i]\right) \geq 0.$$

Appendix .2. Derivative-Free Optimization

There are many derivative-free optimization solvers available in the literature, generally they fall into two categories, deterministic algorithm (such as Nelder-Mead) and evolutionary algorithm (such as BITEOPT [36] and CMA-EA [39]). OptPepperDis() takes password value v as input and outputs optimal pepper distribution $\vec{q}^*$ and attacker's success rate P^*_{adv} when playing with best response given defender's strategy $\vec{q}^*$. During one iteration of OptPepperDis(), some candidate pepper distributions $\{\vec{q}_{c_i}\}$ are proposed, together they are referred as *population*. Then the algorithm BestRes$(v, \vec{q}_{c_i})$ is called as a subroutine for each member of population, and the returned P_{adv} is recorded as "fitness". At the end of each iteration, the population is updated according to fitness of its' members, the update could be either through deterministic transformation (Nelder-Mead) or randomized evolution (BITEOPT, CMA-EA). When the iteration number reaches a pre-defined value *ite*, the best fit member $\vec{q}^*$ and its fitness P^*_{adv} are returned.

Appendix .3. Results for Other datasets



(a) Bfield (b) Brazzers (c) Clixsense

(d) CSDN (e) Rockyou (f) Webhost

Figure .6: Cost-Even Breakpoints, Uniform Pepper Distribution