ErsatzPasswords: Ending Password Cracking and Detecting Password Leakage

Mohammed H. Almeshekah
Computer Science Department
King Saud University
Riyadh, Saudi Arabia
meshekah@ksu.edu.sa

Christopher N. Gutierrez
Computer Science Department
Purdue University
West Lafayette, IN 47907
gutier20@purdue.edu

Mikhail J. Atallah
Computer Science Department
Purdue University
West Lafayette, IN 47907
matallah@purdue.edu

Eugene H. Spafford
Computer Science Department
Purdue University
West Lafayette, IN 47907
spaf@purdue.edu

ABSTRACT

In this work we present a simple, yet effective and practical, scheme to improve the security of stored password hashes, rendering their cracking detectable and insuperable at the same time. We utilize a machine-dependent function, such as a physically unclonable function (PUF) or a hardware security module (HSM) at the authentication server to prevent off-site password discovery, and a deception mechanism to alert us if such an action is attempted. Our scheme can be easily integrated with legacy systems without the need of any additional servers, changing the structure of the hashed password file or any client modifications. When using the scheme the structure of the hashed passwords file, *etc/shadow* or *etc/master.passwd*, will appear no different than in the traditional scheme.[1] However, when an attacker exfiltrates the hashed passwords file and tries to crack it, the only passwords he will get are the ersatzpasswords — the "fake passwords". When an attempt to login using these ersatzpasswords is detected an alarm will be triggered in the system. Even with an adversary who knows about the scheme, cracking cannot be launched without physical access to the authentication server. The scheme also includes a secure backup mechanism in the event of a failure of the hardware dependent function. We discuss our implementation and provide some discussion in comparison to the traditional authentication scheme.

[1]In a traditional Unix system. We believe the same could be made to be true in other systems.

ACSAC '15 Los Angeles, CA

Categories and Subject Descriptors

D.4.6 [**Operating Systems**]: Security and Protection—*Authentication*; K.6.5 [**Management of Computing and Information Systems**]: Security and Protection—*Authentication; Unauthorized access (e.g., hacking, phreaking)*

General Terms

Security

Keywords

Passwords, Password Cracking, Offline Dictionary Attack, Authentication, PUF, HSM, Deception

1. INTRODUCTION

Passwords are the most dominant form of online authentication and likely to remain so for a while despite their weaknesses. It thus behooves us to seek to protect them as much as possible. Within authentication servers, passwords are usually stored in a salted hashed format to prevent easy pre-image recovery. Nevertheless, an adversary who steals the list of hashed passwords can use brute-force to find a password p with a hash value $H(p)$ that equals the value stored for a given user. Later, the adversary can use p to impersonate the user at the authentication server.

There are a number of threats that come with the use of passwords. These threats fall into three main categories; technical, procedural and human related – these will be discussed in more detail in the following section. There have been a number of high-profile thefts of user passwords files in recent years. For example, Evernote reported the leakage of the hashed passwords for more than 50 million users [11]. Other attacks against Yahoo!, RockYou, LinkedIn and eHarmony has been reported [10] [27]. Furthermore, password cracking is often a precursor to more significant attacks as illustrated in [17].

The contribution of our work can be summarized in two main points: (i) we eliminate the possibility of any offline password cracking without physical access to the target's

machine, (ii) when using this scheme the passwords' hashes file will appear no different than a traditional file and if an attacker uses traditional cracking tools to recover users' passwords he will "discover" fake passwords that will trigger an alarm when used. We refer to these fake passwords as "ersatzpasswords". There are somewhat similar schemes that have been proposed in the literature such as "Honeywords" [12] and "Failwords" [18]. However, our mechanism has the following unique advantages (i) eliminating the requirement of any additional server/components (other than the commonly used HSM/TPM), (ii) never presenting the real user credentials to the attackers and, (iii) making password cracking impossible without physical access to the targeted machine. The scheme runs internally in the server without requiring any changes to the user interfaces, clients and/or experiences. A more detailed discussion of related literature is presented in the next section.

One additional contribution our scheme provides is that it imposes risks to any adversary who obtains a file of leaked usernames and passwords, causing mistrust within the market for such files, and rendering their use risky for many parties. This is because the unique property of our scheme of having the username and password file look identical to the file generated by the traditional authentication scheme. This property benefits not only the early adopters of the scheme, but also the overall security of other (non-adopting) systems. This is one of the distinguishing features of using ersatzpassword in comparison to Honeywords [12], PolyPasswordHasher [5], SAuth [14] and others.

2. BACKGROUND AND RELATED WORK

2.1 Passwords

There have been many high profile incidents involving the leakage of hashed passwords files [9]. Users are still using poor passwords, even with the existence of password policies that try to guide users towards choosing more secure passwords. This can be seen in the recent analysis of more than 70 million users' passwords [3]. Bonneau et al. presented an extensive comparative analysis of many authentication schemes replacing passwords [4]. However, passwords will remain in use for many users because of their convenience, ease of use, and ease of deployment.

2.2 Password-Related Threats

The convenient and versatile use of passwords comes with its own challenges. We define password-related threats as the attacks designed to retrieve valid password(s) of current legitimate users of the systems. These host-based[2] threats can be grouped into three main categories.

Technical Threats

There are two categories of technical threats associated with the use of passwords; server-side and client-side. Any piece of malware or key logger that can be installed at the user's machine to obtain the user's password is a threat to any password-based authentication system. At the server side, adversaries can obtain the file of stored passwords and then impersonate the user using the stolen passwords. Strong host security is needed to protect the client and server systems, but there are multiple opportunities for an attacker to capture a copy of the stored password information.

A computer system needs to save an "authenticator" for every user during user enrollment that is used to verify the identity claim during the login phase. Current computer systems store a salted cryptographic hash (H) of the password along with the username. In a system with n users, we have the pairs $(u_1, H(p_1))$, $(u_2, H(p_2))$, ..., $(u_n, H(p_n))$, where u_i is the username of user i and p_i is the password of user i.[3] An attacker who steals this list can launch an offline attack to recover the hashed passwords using some dictionary and replicating the hashing algorithm used. Many tools already exist to automate an attack, such as John the Ripper.[4] There have been many attempts to address this challenge, usually falling into one of three major approaches; (i) significantly increasing the resources needed to match a password, (ii) strengthening user passwords to reduce the likelihood of their recovery as they will be unlikely to be found in a dictionary, and (iii) instrumenting passwords files with decoy passwords triggering an alarm when used indicating that the password file has been attacked.

The development of password hashing algorithms from crypt to bcrypt, scrypt, and others is mainly driven by the goal of increasing the resources needed to crack the users' passwords. The introduction of *private* salts [13] was also intended to increase the work required for cracking the password files. In addition, increasing the number of rounds these algorithms apply to a password is a parallel approach to increasing the work factor. All these attempts are restricted by a maximum threshold that should not be passed; otherwise, normal user login attempts will be noticeably delayed. Additionally, marginally increasing the workload for an adversary to crack a single dictionary password is of little utility when a user chooses a common dictionary word. As these choices are all-too-common,[5] the adversary can still recover some passwords and compromise computer systems.

Cappos and Torres proposed "PolyPasswordHasher" [5] as a scheme to protect passwords from offline dictionary attacks. Their scheme additionally protects passwords with a secret share obtained using the Shamir Secret Sharing scheme [22]. The secret is saved in memory and used to verify passwords. One of the limitations of their scheme is that it requires additional fields in the password file specifying which share to use. Also, if an attacker obtains a single access to the system memory she can steal the secret.

Deception has been used to address the threats associated with cracking password files. One approach is to inject fake accounts into the password file. Another approach is to place decoy password files in the system luring the attackers to access them believing they are the real files. Schemes such as *Honeywords* [12] are intended to confuse the attacker by presenting him with many passwords associated with a single username, where all of them are fake except one.

[2]We acknowledge but are ignoring network snooping and other such remote mechanisms as our work here is directed only at securing host-based password databases. Standard network-based defenses, including encryption, should be employed along with our mechanism.

[3]Salts, as an additional item in many systems, are described later.

[4]http://www.openwall.com/john/

[5]http://splashdata.com/press/worst-passwords-of-2014.htm

Procedural Threats

Password-recovery procedures associated with password-based authentication systems are sometime exploited to override current user passwords [21].

User-Centric Threats

Threats such as phishing, shoulder-surfing, password re-use, and other techniques can be used to undermine the security of password-based authentication systems. Our approach does not address these issues, but can be used in conjunction with other approaches to minimize these risks. For instance, a filter applied at password enrollment can prevent password reuse.

2.3 Injecting Deceit

Deception has been used in computing since at least the 1980s [25, 23]. The prefix "honey" has been used to refer to a wide range of techniques that incorporate the act of deceit. The fundamental idea behind the use of the word "honey" is for those techniques to work they need to entice attackers to interact with them, i.e., fall for the bait: "honey." The term honeytokens was proposed by Spitzner [24] to refer to honeypots but at a smaller granularity. Yuill et al. used the term *honeyfiles* to refer to files that have enticing names distributed in the system that act as a beaconing mechanism when accessed [30]. HoneyGen was also used to refer to tools that are used to generate honeytokens [1].

The use of deceit has been used to address some of the limitations associated with the use of passwords. Yue and Wang proposed a scheme named BogusBiter that shows when users fall for phishing by submitting fake, i.e. deceitful credentials, [29]. Rivest and Juels also proposed augmenting the password database in Unix with negative information such that cracked password files can be detected [12]. In their scheme, $N-1$ fake passwords, i.e. honeywords, are added to each user's entry in the password file. When an adversary cracks the file they are faced with N difference choices of passwords and the real one is one of them.

Bojinov et al. proposed Kamouflage, a scheme that is intended to protect the list of passwords used by a user and saved locally by a password manager [2]. Their scheme hides the real list with a a set of "fake" lists. Kontaxis et al. proposed an authentication scheme (SAuth) that requires each user's login attempt to be vouched by another service provider where an attacker cannot impersonate a user by simply obtaining the password for one web site [14]. They use deception in their scheme as a way to address the common behavior of passwords reuse across multiple service providers. We use deceitful passwords in our scheme, referred to as *ersatzpasswords*. We discuss this in further details in the next section.

3. TECHNICAL SPECIFICATION

3.1 Background

A number of cryptographic functions have been used in computer systems to protect passwords, including crypt, bcrypt, and scrypt. As discussed earlier, part of the motivation to develop additional algorithms is to make the cracking process of stolen password hashes files a resource-intensive process. Our scheme works with any of these underlying functions; we will denote the function used as ($\mathbf{H}$). In later discussion of implementation we will use bcrypt to give a concrete example, but without any loss of generality.

Throughout this section we will assume the following format of the stored password file. For each user i in the system we have the following triplet, at a minimum, (u_i, s_i, α_i) saved in the password file: the username (u_i), a multibyte (multi character) public salt (s_i), and the hash of the user's password p_i as $\alpha_i = \mathbf{H}(p_i \| s_i)$.

In addition, we will use a hardware-specific function denoted as ($\mathbb{HDF}$). This can be implemented as a physically unclonable function (PUF) [26], a hardware security module (HSM) [19] with a unique key, or any other mechanism of equivalent general functionality. Such $\mathbb{HDF}$ can rely on the intrinsic uniqueness of the hardware or tamper-proof protected secret. We note that there has been some previous attempts to secure passwords at the server using an HSM such as S-CRIB's password scrambler.[6]

Our goal is to enhance the security of the storage of passwords in three ways: (i) require the process of computing the hash of the password to require access to a hardware dependent function, thereby thwarting offline cracking of stolen password files, (ii) when an adversary attempts to crack the password file he will be presented with a fake password that can trigger an alarm at the server when used, and (iii) maintain the same appearance and format of the password file while implementing the new scheme. The final property is essential to the success of the deceptive process of injecting "fake" passwords. Unlike the Honeyword scheme, which mixes real passwords with fake ones, our scheme eliminates the ability of an adversary to obtain the real password (without physical access to the targeted machine during the cracking process) and seamlessly presents a fake password during an offline cracking process.

3.2 Threats

In this paper we are considering two main threats against our mechanism. We summarize these threats in this section and a security analysis against each threat later, in section 4.4, showing how our scheme protects against them. In each case, the adversary goal is to obtain the username/password pair of a legitimate *privileged* user account and successfully login.

3.2.1 *Offline Adversary*

An adversary obtains a file of usernames and hashed passwords, e.g., an `etc/shadow` file, of a particular system. The adversary knows about the existence of our scheme, but has no knowledge whether it has been used in the targeted system or not. This threat can be realized by an adversary who compromises a computer system and leaks out the passwords file or obtains such file from the black-market.

3.2.2 *Online Adversary*

In this case, an adversary has access to the system, where she can observe authentication attempts, has access to usernames/passwords files, and can query the hardware-dependent function $\mathbb{HDF}$; the only thing an adversary does not have access to is plaintext passwords and the secret HSM key. This threat can be realized by working with a colluding insider or installing malware in the targeted system.

[6] http://www.s-crib.com/

3.3 One-time Initialization

3.3.1 System-Wide Initialization

The initialization steps in our scheme are performed in two stages; *system-wide* initialization and *user-specific* initialization. The former makes all the users' saved, hashed, passwords machine-dependent – applying the hardware-dependent function as follows. The hardware-dependent function, $\mathbb{HDF}$ is applied to each stored password hash α_i and is then fed to the same hashing function, $\mathbf{H}$, with the original salt, s_i. After that, the output is stored in the password file replacing the old stored value. This system-wide initialization will have each user password stored in the file as the following

$$\beta_i = \mathbf{H}(\mathbb{HDF}(\alpha_i) \| s_i)$$

If an adversary obtains this file and tries to crack any user passwords, the probability that he will get any apparent match is negligible, even if a user password is from a standard dictionary. The cracking software will be searching[7] for a password equal to ${p_i}' = \mathbb{HDF}(\alpha_i)$, where $\alpha_i = \mathbf{H}({p_i}' \| s_i)$, that when hashed will give β_i. However, even if successful, this does not give the true password, α_i that will be needed for access.

An adversary with knowledge of the scheme cannot distinguish between a password file that was computed using our scheme or using the traditional scheme. Even under a stronger assumption, where the adversary knows that the file has been computed using our new scheme, the attacker gains no advantage as he cannot crack users' passwords without access to the hardware used to compute the $\mathbb{HDF}$. In the case where the attacker is an insider, any significant use of the $\mathbb{HDF}$ can be noticed by monitoring API usage.

3.3.2 User-Specific Initialization

To incorporate the additional deceptive alarm component into our scheme — returning an "ersatzpassword" when the adversary attempts cracking the password file — we need to involve each user in a seamless fashion during any normal user authentication. This process requires the user to enter her password, which is a natural step during any authentication, (because the password is not actually stored or recoverable) and can be done during the first login process after the system-wide initialization. We cannot do this step during the system-wide initialization because the passwords are not available in plaintext.

When the user attempts the first login after initializing our system, the password is checked using the original hash function to see if it matches. If so, the scheme will recompute the stored password value β_i as follows. The hardware-dependent function will be applied to the actual password p_i and then an ersatzpassword (p^*) will be chosen – we will discuss the use, choice and characteristics of the ersatzpassword later in section 4.1. A new user-specific salt is then selected, to be used when computing the function $\mathbf{H}$, to satisfy the following property; $[\ {s_i}' = \mathbb{HDF}(p_i || u_i) \ \oplus \ p^*\ ]$. The scheme will take the first 128-bits of the result, assuming we are using a function $\mathbf{H}$ such as *bcrypt* that uses 128-bit salts, as the new salt overwriting the existing salt s_i. It worth noting that the username is part of the $\mathbb{HDF}$ input. The username u_i serves as a "salt" for the $\mathbb{HDF}$ to produce unique outputs even if two users select the same password. Alternatively, a nonce can be generated for each user rather than using the username. However, storing the nonce within the password file would reveal the use of the ersatzpassword scheme and is thus unsuitable. In general, using a username for a salt is bad because an attacker can easily precompute password hashes with common usernames such as root, admin, etc. However, in our proposed scheme, an attacker cannot precompute values for HDF because we assume that the attacker does not have access to the HDF key. We discuss this further in the security analysis section.

We note that the ersatzpassword length can be, at maximum, as long as the salt. In the current implementation of the *bcrypt* function, widely adopted to implement the hash function $\mathbf{H}$, the salt is 128-bits long. This gives us an ersatzpassword of up to 16 characters. This does not impact the plausibility feature of the ersatzpassword, which will be discussed below. In the largest user password study analyzing more than 70 million real user passwords, Bonneau reports that users tend to use passwords of 6-8 characters [3]. If the ersatzpassword is shorter than the salt, the above computation will result in having the salt include some of the output of the $\mathbb{HDF}$ function. This does not affect the security of the system as such output does not leak any useful information about the real password even to someone who has knowledge of the scheme and the length of the ersatzpassword $p*$.

To compute the stored value β our scheme calculates:

$$\beta_i = \mathbf{H}[\ (\mathbb{HDF}(p_i \parallel u_i) \oplus {s_i}') \parallel {s_i}'\ ]$$

If the output of the $\mathbb{HDF}$ is longer than the salt, we address this as follows. We divide this output into chunks of length equal to the salt length. After that, we XOR these chunks together and then XOR the result with the salt ${s_i}'$. Finally, this becomes the input to the hash function $\mathbf{H}$ along with the concatenated salt.

The stored value in the password file will be in the same format used in traditional schemes. When an adversary tries to crack the users' passwords file, he will try to find a password ${p_i}'$ that when hashed using $\mathbf{H}$ and salt s_i will give β_i. In our scheme, we compute beta in a format equivalent to the traditional password storage where the password is p^*, i.e. $\beta = \mathbf{H}(p^* \parallel {s_i}')$. As a result, an attacker who is launching a dictionary attack against a stolen password file will likely find a result identifying p^* as the user password, which is the *ersatzpassword* injected in the system. When the adversary uses this password to login, an internal alarm will be triggered alerting the administrator that someone exfiltrated and attempted to crack the user passwords file.

3.4 Login

There are three main cases of login in our scheme: successful login, when the user enters the correct user/password pair; malicious login, when an adversary uses an ersatzpassword; and error login, when the username/password pair does not match anything. In this section we discuss how to evaluate the login request in these three cases.

When the user i wants to login she presents the username u_i and password $\bar{p}$ to the authentication server. The system identifies the username record and obtains the stored value β_i and the salt s_i associated with it. The scheme computes

$${\beta_i}' = \mathbf{H}[\ (\mathbb{HDF}(\bar{p} \parallel u_i) \oplus s_i) \parallel s_i\ ]$$

and checks whether ${\beta_i}'$ equals β_i; if so, the user is successfully authenticated.

[7] Either using a brute-force method or dictionary attack.

If the match fails, the system checks whether the password presented is the *ersatzpassword*. This is done by computing

$$\beta_i'' = \mathbf{H}[\, \bar{p} \parallel s_i \,]$$

and checking whether this equals β_i. If they are equal, this indicates that someone is trying to impersonate the user after cracking the password file, and an alarm is triggered.

If neither matching attempt succeeds, this can be treated as an erroneous login. The system's policy for erroneous login can then be applied, e.g., up to three unsuccessful attempts are allowed before introducing a temporary lock-out.

3.5 Password Administration

3.5.1 Password Change

The user's password change requests can be treated exactly as a new password. The only difference from traditional password schemes is that our approach mandates the generation of a new salt that satisfies the property discussed above, the XOR operation between the salt and the output of applying $\mathbb{HDF}$ on the password gives an ersatzpassword.

3.5.2 Backup

One of the major factors that hinders the use of hardware-dependent functions is that the system fails catastrophically in the rare case where the hardware associated with the $\mathbb{HDF}$ fails or is no longer available. Thus, we outline a backup procedure that can be used to recover the system in such a failure scenario. This procedure utilizes public-key encryption and is initialized by generating a suitably strong public/private key pair. The private key is never used in normal operation and can be stored in a secure location offline or even in a different physical location. The private key is only needed in the rare event of a failing $\mathbb{HDF}$ and password recovery is needed. The public key is used during the system-wide initialization process and during the process of password change.

When the system is initialized to adopt the new authentication scheme, all the current username u_i, password hash α_i and salt s_i triplets are encrypted using the public key and stored as a backup. In addition, whenever a user changes her password, the new value $\alpha_i{}'$ (the new hash value resulting from the new password using the traditional hash) is computed and the new triple overwrites or is appended to the backup log, along with the u_i and s_i values. As a result, the backup file would have the following list (u_i, s_i, α_i), for every user i in the system, encrypted under the public key.

If a recovery is needed after failure, the private key is fetched and used to recover the log file, which is then used to restore a traditional version of the password file. That file can be instantiated on new hardware, with a new $\mathbb{HDF}$, and users can be forced to reset their passwords — leading to transition to our new scheme as they do so.

It worth noting that decrypting the backup file using means of brute-force should not be practical. Even if the adversary, hypothetically, manages to recover the information in the backup file the resultant password security is at least as strong as currently deployed schemes. The cost in storage and computation to build the recovery log is minimal.

3.5.3 Previous Passwords Storage

It is common for many authentication servers to store previously used users' passwords to prevent users from recycling them [8]. This can put users at risk when such files are compromised. Although users are not using these passwords to login, they can be used to impersonate users at other websites. If systems need to store these passwords nevertheless, our scheme provides an additional advantage over traditional methods of securely storing these passwords.

As our scheme saves the user passwords in a machine-dependent format, using the function $\mathbb{HDF}$, we can have some assurance that this password cannot be cracked offline without physical access to the target machine. Later, when attempting to store the previous passwords used in the system, we can save the passwords using the $\mathbb{HDF}$ function.

3.5.4 Fail-Safe Procedure

We finally point out that in addition to the backup mechanism discussed above to recover the system in the rare case of $\mathbb{HDF}$ function failure, our scheme comes with an intrinsic fail-safe procedure. In this case, we can use the traditional authentication method to check the passwords, comparing $\mathbf{H}(p_i \parallel s_i)$ with the stored value β_i, where the effective user password becomes the ersatzpasswords.

It is important that this is used with care within strict policies as it can be exploited as a backdoor to the system. A session needs to clearly record whether the user logged in using his normal password or the ersatzpassword. This information might also be communicated to the real user. This approach is strongly deprecated.

4. ERSATZPASSWORDS – THE USE OF DECEPTION

The scheme presented in this paper provides the assurance that stored user passwords cannot be cracked without physical access to the hardware-dependent function ($\mathbb{HDF}$), assuming that function is chosen with care.[8] With the increased complexity of computer systems and targeted attacks, computer systems are still vulnerable to security compromise and the list of stored passwords can be stolen. In addition, the latest Verizon Data Breach Investigation Report (DBIR) shows that about 50% of attacks thwarting authentication mechanisms take months or longer to be discovered. Even worse, 88% of these attacks are discovered by external parties. Integrating deceptive passwords in the design of our scheme addresses these two issues.

When attackers obtain the stolen credentials and apply the cracking process, we can design our system to negatively respond to this activity as in [6]. This alerts an attacker who obtains this file to seek different vulnerabilities to exploit. Instead, our default scheme presents an attacker with plausible deceptive passwords leading him to believe that he successfully cracked some passwords. When a login is attempted using the deceptive passwords, defenders will be immediately alerted to two facts: (i) the login credentials database was leaked; and (ii) an attacker is currently trying to impersonate the system's users to gain access. Subsequent defensive and investigative measures can then be taken but without successful access to the system by the attacker.

The process of generating an ersatzpassword for each user account can be formalized as follows. Let $Gen(p_i)$ be the function that takes the user's password and outputs the selected ersatzpassword. This function should have two im-

[8]A PUF is, by definition, unclonable. A TPM with a unique internal encryption key should also serve the same purpose.

portant properties: plausibility and non-deducibility. The former ensures that an ersatzpassword generated by $Gen()$ is plausible to an adversary as a real user password. The latter provides the guarantee that even when an adversary knows the scheme, he cannot deduce any information from the ersatzpassword about the user's real password. We also want this function to be nondeterministic, giving us a different ersatzpassword every time we use it. We define these properties more formally below. After that, we present several constructions of how to realize this function and discuss the advantages and disadvantages of each construction.

4.1 ErsatzPasswords Properties

Incorporating deception in this scheme actively feeds an adversary cracking a stolen password file with some ersatzpasswords chosen to trigger internal alarms when used. These passwords should have the following properties to ensure their effectiveness.

4.1.1 Plausibility

When an adversary is cracking a password file, ersatzpasswords will present themselves as a successful outcome, i.e. when hashed along with the salt they will match the stored hashed user password in the traditional way. For the scheme to be effective, these need to be plausible user passwords. Thus, the generation algorithm should produce plausible ersatzpasswords (so their generation is random subject to plausibility rather than in absolute terms).

We are concerned with two types of plausibility; individual ersatzpassword plausibility and collective ersatzpasswords plausibility. The former ensures that the adversary will not suspect a single user's ersatzpassword as not being likely. The latter ensures that when an adversary gets the output of a password cracking tool, she will be as likely to believe the outcome as if our scheme has not been used.

We can define a plausible generator function $Gen()$ by using the following game:

- The adversary views many runs of the function $p^* = Gen(p)$ along with their associated usernames, where p^* is the ersatzpassword. The adversary can choose the values of u and/or p.
- The adversary sends a username to $Gen()$.
- $Gen()$ selects a password p, either from the real user, existing user or public password files, and computes the ersatzpassword $p^* = Gen(p)$. Then $Gen()$ sends both p and p^* back without indicating which is which.
- The adversary outputs (1) if she thinks $p*$ is the ersatzpassword and (0) otherwise. The adversary wins if she distinguishes the ersatzpassword from the real password with probability Pr.

We say that $Gen()$ is a plausible function if the probability for adversarial success is one-half — an adversary cannot do better than random guessing which of the two passwords is the ersatzpassword. That is, $Pr = 1/2 + \epsilon$ where ϵ is increasingly negligible as the number of trials increases.

For collective plausibility, we add another requirement in addition to having to achieve individual plausibility for every password. This requirement is that the ratio of "broken" passwords to "not broken" needs to be plausible. We discuss this further, below in 4.1.5.

4.1.2 Typo-Resilience

When the user is typing her real password, she may make a mistake by mistyping some characters. The ersatzpassword associated with her account should have enough edit distance from the actual password to avoid triggering an erroneous alarm. As the real user password is present when selecting which ersatzpassword to use, the server can compute an edit distance to ensure an appropriate difference.

4.1.3 Non-Deducibility

It is essential for the ersatzpassword to not reveal any useful information about the real user password. Even though we do not actively give adversaries the ersatzpasswords, we store them with the same level of protection used to store current real users' passwords. We define the function $Gen()$ to provide non-deducibility using the following game:

- The adversary views many runs of the function $p^* = Gen(p)$ where she can choose the values of u and/or p (p^* is the ersatzpassword).
- The adversary chooses two passwords p_1 and p_2, and sends then to function $Gen()$.
- $Gen()$ flips a coin and computes $p^* = Gen(p_1)$ if it gets heads or $p^* = Gen(p_2)$ otherwise. p^* is then presented to the adversary.
- The adversary outputs (1) if she thinks $p^* = Gen(u, p_1)$ and (0) otherwise with probability Pr.

We say that $Gen()$ is a non-deducible function if the probability for adversary success is half. — the adversary cannot do better than randomly guessing which of the two passwords was used to generate p^*. That is, $Pr = 1/2 + \epsilon$ where ϵ is increasingly negligible as the number of trials increases.

4.1.4 Policy Adherence

It is essential that ersatzpasswords adhere to any system-wide policy of how users' password should be chosen. For example, some restrictions can be imposed on the length, format and composition of user passwords. An adversary who sees any password violating the system's policy can deduce that this cannot be a real password — the system would not have accepted it. Some websites mandate that user passwords cannot be dictionary words. In these cases, using a password list to generate ersatzpasswords can be challenging as it is nontrivial to generate a long list satisfying each server's policy. Additionally, any change to the policy would require recomputing the list. However, the use of grammar-based approaches, similar to the one illustrated below, can be much simpler as grammar can become part of the input of the generator function $Gen()$.

4.1.5 Crackable

Part of the plausibility aspect of our scheme is deciding whether all ersatzpasswords should be crackable or not. Generally, they should not. Many current systems add more stringent requirements of password choice to highly privileged users. When they become easily crackable, this might increase an adversary's suspicions. It would also look suspicious if all user passwords were crackable. Thus, it would be wise to use some randomly-generated ersatzpasswords within a system to enhance the scheme's plausibility.

Most studies found that 20-50% of passwords in a password files can be cracked using state of the art cracking scheme [3]. This has been done with dictionaries in the range of $2^{20} - 2^{30}$. These studies have also found that increasing the size of the dictionary has diminishing returns. When generating ersatz passwords we should seek to have a crackability ratio of 20-50% to ensure the plausibility of our design. Fully random passwords can be used as the uncrackable ersatz passwords.

4.2 ErsatzPasswords Generation

4.2.1 Total Password Replacement

When *Gen*() receives the user password it can generate the ersatzpassword using the following procedure. For every character in the user password, replace it with a randomly chosen character from the same category (alphabetical with alphabetical, a digit with a digit, and a special character with a special character). After this replacement process, a cyclic shift is applied to the password by a random number of positions to generate the ersatzpassword.

We note that this process reveals two properties of the real password to an adversary when he views the ersatzpassword: the password's length and its character composition. If this happens, adversaries can use probabilistic context-free replacement to significantly narrow the space of possible user passwords using knowledge of the ersatzpassword [28]. This may be overcome by randomly truncating or injecting some random characters to generate the ersatzpassword.

4.2.2 List-Based

One of the most straightforward ways of generating the ersatzpassword using *Gen*() is to randomly choose a word from an internal dictionary of candidates. This realization of *Gen*() has two major limitations: the generation of ersatzpassword is not influenced by user-specific information and the existence of such a list in the system can affect the stealthiness of the deceptive component (the existence of the list is a sign that such a scheme is currently being used by the system). The former limitation is not as significant because the attacker never sees the "real" users' passwords. The advantage of using such method is the ability to have a high degree of plausibility of the ersatzpasswords. We can compile a list of some of the previously leaked passwords used by real users and use them as our ersatz passwords.

4.2.3 Grammar-Based Methods

Bojinov et al. propose a new method of generating plausible user passwords in [2] extending the work of Ross et al. [20] and Weir [28]. Their method is similar to our *total password replacement* method, however they tokenize the password representing distinct syntactic elements. For example, the password *"wtyy234ou*"* has the following token sequence $W_1 = \{wtyy\} \mid D_2 = \{234\} \mid W_3 = \{ou\} \mid S_4 = \{*\} \mid$. When generating the ersatzpassword, each token will be replaced with another token, of the same length, from a dictionary.

The main drawback of this method is that it leaks the type, number, and length of tokens of the original password. We address this concern by enhancing their implementation of *Gen*() as follows. After tokenizing the password, we perform the following:

- We may randomly append or delete k tokens. For example, we add token S_5 to the above password.
- After that, we may randomly shuffle the order of these tokens. In the above example, the shuffle might give us the following order $W_3 \mid S_5 \mid D_2 \mid S_4 \mid W_1$.
- Finally, we may randomly choose a word from a dictionary that matches each token class. The chosen token can have a length that is different from the original token. In our example, $W_3 =$ "*abc*", $S_5 =$ "!", $D_2 =$ "10", $S_4 =$ " + " and $W_1 =$ "*test*".

Using the grammar-based method with our modification on our example might produce the ersatzpassword *"abc!10+test"*, which bears little resemblance to the input.

4.2.4 Using User Input

Our discussion so far assumes that the scheme can work without any interaction with the system users. However, we note that ersatzpassword can be constructed with implicit or explicit user input. Many authentication servers save previously used user passwords in the system preventing users from recycling their old password when their current password expires. This implicit user input, previously chosen user passwords, can be used as the ersatzpassword for this user account. With explicit user input, the system can prompt the user to enter another password during registration and use this as the ersatzpassword password.

The main advantage of using implicit user input is ensuring a high degree of plausibly of the ersatzpassword as this has been previously used as a real password. However, this method suffers from two major disadvantages. First, if an adversary cracks the password file and recovers the ersatzpassword, this might put the user in danger as users are known to reuse passwords across multiple sites [7]. Second, this has the potential of signaling a false alarm when the user forgets and uses his previous password to login.

Explicit user input requires some changes to the user interface. More importantly, users are likely to pay less attention, choosing very guessable passwords and/or confusing the ersatz passwords with their real ones leading to the problem of false alarms. In addition, users may provide an additional, ersatzpassword that is closely related to their real password, e.g. by appending a number or a character to their real password to create the ersatzpassword.

A combination of several of these methods may be the best approach, with user input restricted to particularly well-informed users so as to prevent accidents. We also note that our scheme requires no relationship between the ersatz and the real passwords. Thus, in many cases it might be better to use a generation algorithm that do not use the original passwords as an input in its design.

4.3 Detecting Database Leakage

Our scheme is designed with two main goals; ending password cracking and detecting password file leakage. With the integration and use of HDF to store password hashes, we eliminate the possibility that users' password can be cracked without physical access to the machine where they have been stored. However, protecting one part, i.e. the password file, informs adversaries to focus their resources on other, probably more vulnerable, parts of the computer systems. The design of our scheme gives defenders an edge (although it may be slight) over attackers to better detect compromises and proactively respond to them.

An adversary who cracks the password file and tries to use

the cracked erstazpassword will alert system administrators that: (i) (at least part of) the password file has been taken, (ii) someone attempted to crack the file; and (iii) someone is trying to use a cracked password to gain access to the system. Security administrators can easily configure their systems to forward adversaries to a fake account where they can be further monitored and observed. Thus, our scheme comes with an intrinsic incentive that security administrators will not only enhance the security of password storage, but also detect when such files are leaked and cracked. Additionally, our scheme can help defenders to safely direct any usage of ersatzpasswords to a honeypot or fake account to further observe behavior and learn about their attackers.

There is an externality effect that comes with the user of our scheme. If a small percentage of servers use ersatzpassword and adversaries know about it, this will increase the suspension and doubt of any leaked and cracked password files. The adversaries will perceive an added risk – of being deceived and/or monitored – to the use of those cracked passwords as they cannot distinguish whether they are the real passwords or the ersatzpasswords.

4.4 Security Analysis

In this section we discuss the provided security of our scheme and give a detailed discussion of the resistance against the threats presented above.

Our method stores values in the same format as an unmodified system: username, password-dependent hash and a salt. The salt itself is computed as follows:

$$salt = \mathbb{HDF}(passwd \parallel username) \oplus ersatzpassword$$

and the password hash is stored as:

$$\mathbf{H}[\ (\mathbb{HDF}(passwd \parallel username) \oplus salt) \parallel salt\ ] = \mathbf{H}(ersatzpassword \parallel salt\ )$$

To obtain the real user's password, an adversary with no knowledge that our scheme has been used will "crack" the hash, retrieving the ersatzpassword. However, an adversary with knowledge that the scheme has been used will be able to retrieve $\mathbb{HDF}(passwd \parallel username)$ by XOR'ing the salt with the ersatzpassword. To succeed, the adversary will then need to check every password guess by querying the $\mathbb{HDF}$. An *offline adversary* is unable to access the $\mathbb{HDF}$ and is thus blocked. *Online adversaries* may query the $\mathbb{HDF}$ for every possible "guess" but this spike in usage can be detected. Moreover, the $\mathbb{HDF}$ API can be hardened to require superuser privileges. This protects the system against partially-known plaintext attacks to break our implementation of the $\mathbb{HDF}$ function, because the adversary knows the username. However, if an adversary has those privileges, she can simply change the targeted user or use her privileges to act on her target and the password is a minor concern! We also note that if the adversary breaks the $\mathbb{HDF}$ function, the security of the password is reduced to the currently used hashing-based password storage schemes.

We exert some control over the choice of salt in our design. Salts are traditionally used for two main goals: (i) reduce the effectiveness of precomputed hash tables; and (ii) hide the fact that two users have the same password [16]. The first property of using salts is still maintained for cracking both the real password and the ersatzpasswords. For the ersatzpasswords, one needs to use the salt to crack them in the traditional ways. For the real passwords, the adversary does not have offline access to the $\mathbb{HDF}$, so she cannot precompute the dictionary of all possible passwords. The second property of using salts is also maintained in our scheme. No two $\mathbb{HDF}$ outputs will be the same because we use the username in the input and the username is locally unique.

As discussed earlier, we need to achieve plausibility of both individual ersatzpasswords and the whole password file. The former can be achieved by a good choice of a $Gen()$ function. The latter can be achieved by selecting a plausible ratio of passwords that cannot be cracked.

In our discussion, we we assume that the attacker does not have an active "root" access on the authentication server. If she has root access, she might use sophisticated measure to detect whether passwords have been stored in non-standardized way. One way to achieve that to check the binary of the authentication modules. However, this will only inform an adversary that these modules have been changed without giving any details of how they have been changed – unless they can be exfiltrated and disassembled.

5. IMPLEMENTATION AND PERFORMANCE EVALUATION

5.1 Implementation Details

We implemented a prototype of our scheme by modifying the authentication mechanism in the FreeBSD operating system. The *pam_unix* Pluggable Authentication Module (PAM), which handles the user authentication process, is modified to incorporate our scheme. The design decision is driven by the simplicity of PAM modules as well as the preservation of expected behavior during user authentication. The effectiveness of the deception relies on the fact that the user authentication system appears no different than standard FreeBSD user authentication.

The system relies on two key components: the hardware dependent function $\mathbb{HDF}$ and the ersatzpassword generation function $Gen()$. We used the basic Yubico YubiHSM, a USB hardware security module, as our $\mathbb{HDF}$. Specifically, $\mathbb{HDF}$ is a HMAC-SHA1 with a fixed secret key (k) internally stored inside the HSM:[9]

$$\mathbb{HDF}(p \parallel u_i) := \text{HMAC-SHA1}_k(p \parallel u_i)$$

For the ersatzpassword generation, $Gen()$, we implemented the List-Based approach described in section 4.2.2. This choice was mainly driven by the fact that we can pre-select ersatzpasswords and have more accurate measurements. The code can be easily modified to choose any ersatzpassword generation algorithm. As a proof of concept, we used a list of six-character dictionary words;[10] a password is selected from the dictionary of 15,788 and used as the ersatzpassword during user account initialization.

This realization of the $Gen()$ functions achieves plausibility because all the ersatzpasswords in the file are an example of a weak password used by a real user. It is also typo-resilient as we can measure the Levenshtein distance between the original passwords and the ersatz and make sure they are far enough [15]. Additionally, it achieves non-deducability as the ersatzpassword does not depend in any

[9]A PUF would have been preferable, but we do not have access to one.

[10]Obtained from `http://www.poslarchive.com/math/scrabble/lists/common-6.html`

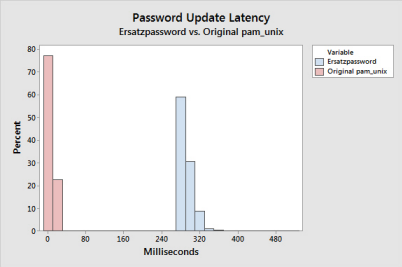

Figure 1: Comparison of update latency in the modified and standard FreeBSD.

way on the real password. We can simply pre-process all the passwords in the file to make sure they adhere to the organization's password policy. Finally, ersatzpasswords are crackable by definition because they are dictionary words.

5.2 Analysis

We analyze two authentication processes when comparing our implementation of the new authentication scheme with the standard FreeBSD mechanism. First, we compared the latency for adding a new user into the system and the latency for authenticating a valid user. Second, the storage of cryptographic hashes of the user's password must appear and behave as in a typical FreeBSD operating system. In addition to maintaining the fidelity of accurate user authentication, user password hashes must also work with conventional password cracking tools such as John the Ripper to ensure plausibility. We conducted our analysis on a FreeBSD virtual machine with a single core clocked at 2.7 Ghz.

5.2.1 *Password Update and Authentication Latency*

To evaluate the performance of our authentication module, we compared its latency with the standard `pam_unix` module found in FreeBSD. Two measurements were considered: the latency to update an existing password, and the latency to authenticate a user. The password was fixed to "password" for all experiments. Additionally, the evaluation also considered the latency of using "ersatz" for the ersatzpassword. The evaluation consisted of running `pam_chauthtok` and `pam_authenticate` as found in `passwd` and `login`. Password update and authentication latencies were sampled 1000 times independently on an idle FreeBSD virtual machine.

As shown in figure 1, the median latency time to update a user's password for our ersatz system is 287.3 ms while the latency on a standard FreeBSD system is 8.8 ms. These results indicate that further optimization is needed to reduce the latency for our module to match the expected behavior of the standard FreeBSD `pam_unix` module.

A similar pattern is observed when comparing authentication latency. Figure 2 illustrates the latencies in system response observed when providing a valid password and an ersatzpassword in our system in comparison with the latency when providing a valid password in a conventional FreeBSD system. Note that the latency difference between our system and the conventional system are similar to the password update latency. The median system latency for authentication in our system is 277.76 ms when providing the correct password and 281.95 ms when providing the ersatzpassword. The median latency for authenticating a valid user on a standard FreeBSD system is 5.14 ms.

Step	# of instructions	Percentage
Initialize	77,737,691	68.47%
Generate Ersatz	6,816	0.006%
Create Salt	273,322	0.24%
Hash Password	28,551,342	25.5%
Close	5,325,039	4.75%

Table 1: Number of instructions for creating a new user under the ersatz `pam_unix` module

We note that there are a number of reasons for the observed performance difference. The YubiHSM APIs are written in python and the implementation of our scheme is written in C as a modified `pam_unix` module. A call from C to Python has an impact on system performance. To validate our concern, we used `pmcstat`[11] to profile our modified `pam_unix` module. The results from `pmcstat` showed that the largest bottleneck is found in the `libpython2.7.so` library. Specifically, the bottleneck is `PyEval_FrameEx`, which interprets and executes bytecodes from a given frame. Another bottleneck is `PyObject_Malloc`, which is indirectly called when converting a C string to a Python string. Such conversion is needed in our modified `pam_unix` module when initializing the YubiHSM and generating a salt or the hash.

To investigate other potential bottlenecks, we used `valgrind`[12] with the `callgrind` toolset to compare the number of instructions executed in the ersatz `pam_unix` module and the standard freeBSD `pam_unix` module. For each module, we looked at the amount of time it takes to enroll a new user in the system. For the ersatz `pam_unix` module, this includes initializing and closing the YubiHSM module in addition to generating an ersatz password, creating a salt value, and hashing. Table 1 contains the number of instructions for each function needed to create a new user. Initiating and closing the session with YubiHSM accounts for more than 70% of the instructions. In comparison to the standard freeBSD `pam_unix` module, creating a salt takes 24,171 instructions and generating a password takes 28,202,224 instructions. Generating a salt in the ersatz *pam_unix* module takes roughly 10 times longer than in the standard freeBSD `pam_unix` module. The total number of instructions to enroll a new user in the ersatz `pam_unix` module takes about 4.96 times more than the standard freeBSD `pam_unix` module.

We also investigated the I/O overhead for both modules with `ktrace`, which enables kernel trace logging in freeBSD. In the ersatz `pam_unix` module, about 280ms are spent waiting for I/O, while in the `pam_unix` module only 0.008ms is spent waiting for I/O. These numbers indicate that the overhead to communicate with the YubiHSM accounts for the largest bottleneck in our ersatz `pam_unix` module.

The main reason for the performance deficiencies above is that we are using a basic $\mathbb{HDF}$ function, namely the YubiHSM, which is not optimized for performance. A built-in device rather than a USB device should provide a speed improvement and reduce the I/O overhead. We believe that a combination of optimizations might bring the times close enough that it would not be obvious to an observer what might be in use on the system. However we believe that the latency, as it currently stands, could be deemed insignificant

[11] `https://wiki.freebsd.org/PmcTools`
[12] `http://valgrind.org/`

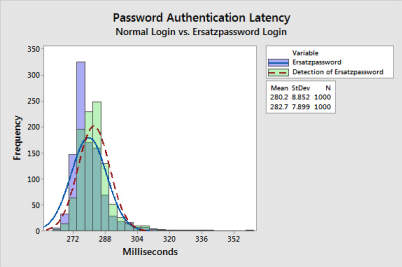

Figure 2: User authentication latency in our scheme using a valid and an ersatz passwords.

under certain deployments.

We did not evaluate the performance of recovering passwords in the case of $\mathbb{HDF}$ failure. This is part of the planned future work.

5.2.2 *Crackable Ersatz Hashes*

To demonstrate that our scheme produces crackable hashes, we generated 1000 hashes with the real passwords of `password1, password2, ···, password1000` and ersatzpasswords randomly selected from our list of six-character dictionary words. We ran *John the Ripper* on all 1000 hashes created by our scheme. John the Ripper successfully cracked all 1000 hashes and retrieved all the ersatzpasswords.

We note that studying the overall plausibility of a cracked ersatzpasswords file is an area of planned future research.

6. CONCLUSION

Passwords are widely regarded as one of the weakest points in securing any digital system. They come with their inherent weaknesses in how they are chosen, stored, memorized and managed. In this paper, we presented a scheme that addresses the wide-spread threat of stealing hashed password files and cracking them offline to impersonate user accounts to further infiltrate computer systems. Our scheme makes it impossible for an adversary to recover user passwords from their hashed format without physical access to the targeted machine. We show we can instantaneously protect any system without the involvement of its users. Furthermore, we discussed how we can deceive an attacker who steals the hashed users' password file by presenting him with ersatzpasswords that work as decoy passwords triggering an alarm when used to access the system. We discussed how to generate these passwords and their properties. Finally, we implemented our scheme discussing the design decisions and the performance analysis. Our goal with the deployment of our scheme, we can end the possibility of cracking user passwords and, at the same time, detect any exfiltration and cracking attempt on users' hashed password file.

7. ACKNOWLEDGMENTS

The authors would like to extend their thanks to Dan Trinkle and Keith Watson for their time, discussion, and ideas they provided. Portions of this work were supported by National Science Foundation Grants CPS-1329979, Science and Technology Center CCF-0939370; by an NPRP grant from the Qatar National Research Fund; and by sponsors of the Center for Education and Research in Information Assurance and Security. The statements made herein are solely the responsibility of the authors.

8. AVAILABILITY

The implementation discussed in this paper can be accessed at GitHub at `https://github.com/cngutierr/ErsatzPassword`.[13]

9. REFERENCES

[1] M. Bercovitch, M. Renford, L. Hasson, A. Shabtai, L. Rokach, and Y. Elovici. HoneyGen: An Automated Honeytokens Generator. In *Intelligence and Security Informatics (ISI), 2011 IEEE International Conference on*, pages 131–136. IEEE, 2011.

[2] D. Bojinov, Hristo and Bursztein, Elie and Boyen, Xavier and Boneh. Kamouflage : Loss-Resistant Password Management. In *Proceedings of the 15th European conference on Research in computer security*, pages 286—-302. Springer-Verlag, 2010.

[3] J. Bonneau. The science of guessing: Analyzing an anonymized corpus of 70 million passwords. In *Proceedings - IEEE Symposium on Security and Privacy*, pages 538–552, 2012.

[4] J. Bonneau, C. Herley, P. C. Van Oorschot, and F. Stajano. The quest to replace passwords: A framework for comparative evaluation of web authentication schemes. In *Proceedings - IEEE Symposium on Security and Privacy*, pages 553–567, 2012.

[5] J. Cappos and S. Torres. PolyPasswordHasher: Protecting Passwords In The Event Of A Password File Disclosure. Technical report, 2014.

[6] D. Cvrcek. Hardware Scrambling - No More Password Leaks. Technical report, 2014.

[7] A. Das, J. Bonneau, M. Caesar, N. Borisov, and X. Wang. The Tangled Web of Password Reuse. In *NDSS '14: The 2014 Network and Distributed System Security Symposium*, San Diego, CA, USA, 2014.

[8] Defense Information Systems Agency (DISA) for the Department of Defense (DOD). Application security and development: Security technical implementation guide (STIG). Technical report.

[9] M. DeLuca and J. Pepitone. eBay Warns Customers to Change Passwords After Database Hacked, 2014.

[10] C. Gaylord. LinkedIn, Last.fm, now Yahoo? Don't ignore news of a password breach.

[11] D. Gross. 50 million compromised in Evernote hack, Mar. 2013.

[12] A. Juels and R. L. Rivest. Honeywords: making password-cracking detectable. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pages 145–160. ACM, 2013.

[13] D. V. Klein. Foiling the Cracker; A Survey of, and Improvements to Unix Password Security. In *14th DoE Computer Security Group*, May 1991.

[14] G. Kontaxis, E. Athanasopoulos, G. Portokalidis, and A. D. Keromytis. SAuth: protecting user accounts from password database leaks. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pages 187–198. ACM, 2013.

[13]The performance analysis in this paper were done for commit number (d3efd8dec831909abe34fad3c858917f53d1c831).

[15] V. I. Levenshtein. Binary codes capable of correcting deletions, insertions, and reversals. *Soviet Physics Doklady*, 10(8):707–710, 1966.

[16] R. Morris and K. Thompson. Password security: a case history, 1979.

[17] N. Perlroth. Hackers in China Attacked The Times for Last 4 Months.

[18] S. Rao. Data and system security with failwords, 2005.

[19] S. Requirements. Hardware Security Module (HSM). *Security*, pages 1–26, 2009.

[20] B. Ross, C. Jackson, N. Miyake, D. Boneh, and J. C. Mitchell. Stronger password authentication using browser extensions. In *Proceedings of the 14th conference on USENIX Security Symposium - Volume 14*, page 2, 2005.

[21] S. Schechter, A. J. B. Brush, and S. Egelman. It's no secret Measuring the security and reliability of authentication via 'secret' questions. In *Proceedings - IEEE Symposium on Security and Privacy*, pages 375–390, 2009.

[22] A. Shamir. How to share a secret. *Commun. ACM*, 22(11):612–613, Nov. 1979.

[23] E. Spafford. More than Passive Defense, 2011.

[24] L. Spitzner. Honeytokens: The other honeypot, 2003.

[25] C. P. Stoll. The Cuckoo's Egg: Tracing a Spy Through the Maze of Computer Espionage, 1989.

[26] G. E. Suh and S. Devadas. Physical unclonable functions for device authentication and secret key generation. In *Proceedings - Design Automation Conference*, pages 9–14, 2007.

[27] M. Weir, S. Aggarwal, M. Collins, and H. Stern. Testing metrics for password creation policies by attacking large sets of revealed passwords. In *Proceedings of the 17th ACM conference on Computer and communications security*, pages 162–175. ACM, 2010.

[28] M. Weir, S. Aggarwal, B. De Medeiros, and B. Glodek. Password cracking using probabilistic context-free grammars. In *Proceedings - IEEE Symposium on Security and Privacy*, pages 391–405, 2009.

[29] C. Yue and H. Wang. BogusBiter: A transparent protection against phishing attacks. *ACM Transactions on Internet Technology (TOIT)*, 10(2):6, 2010.

[30] J. Yuill, M. Zappe, D. Denning, and F. Feer. Honeyfiles: deceptive files for intrusion detection. In *Information Assurance Workshop, 2004. Proceedings from the Fifth Annual IEEE SMC*, pages 116–122. IEEE, 2004.