Password Hardening Based on Keystroke Dynamics

Fabian Monrose Michael K. Reiter Susanne Wetzel

Bell Labs, Lucent Technologies
Murray Hill, NJ, USA
{fabian,reiter,sgwetzel}@research.bell-labs.com

Abstract

We present a novel approach to improving the security of passwords In our approach, the legitimate user's typing patterns (e.g , durations of keystrokes, and latencies between keystrokes) are combined with the user's password to generate a *hardened password* that is convincingly more secure than conventional passwords against both online and offline attackers. In addition, our scheme automatically adapts to gradual changes in a user's typing patterns while maintaining the same hardened password across multiple logins, for use in file encryption or other applications requiring a long-term secret key Using empirical data and a prototype implementation of our scheme, we give evidence that our approach is viable in practice, in terms of ease of use, improved security, and performance

1 Introduction

Textual passwords have been the primary means of authenticating users to computers since the introduction of access controls in computer systems Passwords remain the dominant user authentication technology today, despite the fact that they have been shown to be a fairly weak mechanism for authenticating users Studies have shown that users tend to choose passwords that can be broken by an exhaustive search of a relatively small subset of all possible passwords. In one case study of 14,000 Unix passwords, almost 25% of the passwords were found by searching for words from a carefully formed "dictionary" of only 3×10^6 words [10] (see also [21, 4, 27, 29]) This high success rate is not unusual despite the fact that there are roughly 2×10^{14} 8-character passwords consisting of digits and upper and lower case letters alone

In this paper, we propose a technique for improving the security of password-based applications by incorporating biometric information into the password Specifically, our technique generates a *hardened password* based on both the password characters and the user's typing patterns when typing the password. This hardened password can be tested for login purposes or used as a cryptographic key for file encryption, virtual private network access, etc. An attacker who obtains all stored system information for password verification (the analog of the `/etc/passwd` file in a typical Unix environment) is faced with a convincingly more difficult task to exhaustively search for the hardened password than in a traditional password scheme Moreover, an attacker who learns the user's textual password (e g., by observing it being typed) must type it like the legitimate user to log into an account protected by our scheme

There are several challenges to realizing this goal. The first is to identify features of a user's typing patterns (e.g , latencies between keystrokes, or duration of keystrokes) that the user reliably repeats (approximately) when typing her password The second is to use these features when the user types her password to generate the correct hardened password At the same time, however, the attacker who captures system information used to generate or verify hardened passwords should be unable to determine which features are relevant to generating a user's hardened password, since revealing this information could reveal information about the characters related to that password feature. For example, suppose the attacker learns that the latency between the first and second keystrokes is a feature that is reliably repeated by the user and thus is used to generate her hardened password Then this may reveal information about the first and second characters of the text password, since due to keyboard dynamics, some digraphs are more amenable to reliable latency repetitions than others.

Our approach effectively hides information about which of a user's features are relevant to generating her hardened password, even from an attacker that captures all system information. At the same time, it employs novel techniques to impose an additional (multiplicative) work factor on the attacker who attempts to exhaustively search the password space. Using empirical data, we evaluate both this work factor and the reliability with which legitimate users can generate their hardened passwords Our empirical studies demonstrate various choices of parameters that yield both increased security and sufficient ease of use

Our scheme is very attractive for use in practice. Unlike other biometric authentication procedures (e.g., fingerprint recognition, retina or iris scans), our approach is unintrusive and works with off-the-shelf keyboards. Our scheme initially is as secure as a "normal" password scheme and then adapts to the user's typing patterns over time, gradually hardening the password with biometric information Moreover, while fully able to adapt to gradual changes in user typing patterns, our scheme can be used to generate the *same* hardened password indefinitely, despite changes in the user's typing patterns. Therefore, the hardened password can be used, e.g , to encrypt files, without needing to decrypt and re-encrypt files with a new hardened password on each login.

The main limitation of our scheme is that a user whose typing patterns change substantially between consecutive instances of typing her password may be unable to generate

her correct hardened password and thus, e g , might be unable to log in The most common circumstance in which this could happen is if the user attempts to log in using a different style keyboard than her regular one, which can cause a dramatic change in the user's typing patterns. In light of this, applications for which our scheme is ideally suited are access to virtual private networks from laptop computers, and file or disk encryption on laptop computers Laptops provide a single, persistently available keyboard at which the user can type her password, which is the ideal situation for repeated generation of her hardened password Moreover, with the rising rate of laptop thefts (e g , see [22]), these applications demand security better than that provided by traditional passwords

2 Related work

The motivation for using keystroke features to harden passwords comes from years of research validating the hypothesis that user keystroke features both are highly repeatable and different between users (e g , [6, 28, 14, 15, 1, 9, 20, 24]). Prior work has anticipated utilizing keystroke information in the user login process (e g , [9]), and indeed products implementing this are being marketed today (e g , see `http://www.biopassword.com/`) All such prior schemes work by storing a model of user keystroke behavior in the system, and then comparing user keystroke behavior during password entry to this model Thus, while they are useful to defend against an online attacker who attempts to log into the system directly, they provide no additional protection against an offline attacker who captures system information related to user authentication and then conducts an offline dictionary attack to find the password (e.g , to then decrypt files encrypted under the password). On the contrary, the captured model of the legitimate user's keystroke behavior can leak information about the password to such an attacker, as discussed in Section 1 Thus, our work improves on these schemes in two ways. First, our method is the first to offer stronger security against *both* online and offline attackers. Second, our scheme is the first to generate a repeatable secret based on the password and keystroke dynamics that is stronger than the password itself and that can be used in applications other than login, such as file encryption

The only work of which we are aware that previously proposed generating a repeatable key based on biometric information is [3] In this scheme, a user carries a portable storage device containing (i) error correcting parameters to decode readings of the biometric (e.g , an iris scan) with a limited number of errors to a "canonical" reading for that user, and (ii) a one-way hash of that canonical reading for verification purposes Moreover, they further proposed a scheme in which the canonical biometric reading for that user is hashed together with a password Their techniques, however, are inappropriate for our goals because the stored error correcting parameters, if captured, reveal information about the canonical form of the biometric for the user. For this reason, their approach requires a biometric with substantial entropy. e g , they considered iris scans offering an estimated 173 bits of entropy, so that the remaining entropy after exposure of the error correcting parameters (they estimated 147 bits of remaining entropy) was still sufficiently large for their application. In our case, the measurable keystroke features for an 8-character password are relatively few (at most 15 on standard keyboards), and indeed in our scheme, the password's entropy will generally dominate the entropy available from keystroke features. Thus, exposing error-correcting parameters in our setting would substantially diminish the available entropy from keystroke features, almost to the point of negating their utility Moreover, exposing information about the keystroke features can, in turn, expose information about the password itself (as discussed in Section 1) This makes the careful utilization of keystroke features critical in our setting, whereas in their setting, the biometrics they considered were presumed independent of the password chosen.

Our method to harden user passwords has conceptual similarities to password "salting" for user login Salting is a method in which the user's password is prepended with a random number (the "salt") of s bits in length before hashing the password and comparing the result to a previously stored value [21, 16] As a result, the search space of an attacker is increased by a factor of 2^s if the attacker does not have access to the salts. However, the correct salt either must be stored in the system or found by exhaustive search at login time Intuitively, the scheme that we propose in this paper can be used to improve this approach, by determining some or all of the salt bits using the user's typing features. In addition, an advantage of our approach over salting is that our scheme can be effective against an online attacker who learns the legitimate user's password (e.g , by observing the user type it) and who then attempts to log in as that user.

Finally, we note that several other research efforts on password security have focused on detecting the unauthorized modification of system information related to password authentication (e g , the attacker adds a new account with a password it knows, or changes the password of an existing account) [13, 12, 8] Here we do not focus on this threat model, though our hardened passwords can be directly combined with these techniques to provide security against this attacker, as well

3 Preliminaries

The hardened passwords generated in our scheme have many potential uses, including user login, file encryption, and authentication to virtual private networks However, for concreteness of exposition, in the rest of this paper we focus on the generation and use of hardened passwords for the purposes of user login Extending our discussion to these other applications is straightforward.

We assume a computer system with a set A of user accounts Access to each user account is regulated by a login program that challenges the user for an account name and password. Using the user's input and some stored information for the account a that the user is trying to access, the login program either accepts or rejects the attempt to log into a. Like in computer systems today, the characters that the user types into the password field are a factor in the determination to accept or reject the login. For the rest of this paper, we denote by pwd_a the correct string of characters for the password field when logging into account a. That is, pwd_a denotes the correct text password as typically used in computer systems today.

In our architecture, typing pwd_a is necessary but not sufficient to access a. Rather, the login program combines the characters typed in the password field with keystroke features to form a hardened password that is tested to determine whether login is successful. The correct hardened password for account a is denoted hpwd_a. The login program will fail to generate hpwd_a if either something other than pwd_a is entered in the password field or if the user's

typing patterns significantly differ from the typing patterns displayed in previous successful logins to the account. Here we present our scheme in a way that maintains hpwd_a constant across logins, even despite gradual shifts in the user's typing patterns, so that hpwd_a can also be used for longer-term purposes (e.g., file encryption). However, our scheme can be easily tuned to change hpwd_a after each successful login, if desired.

3.1 Features

In order to generate hpwd_a from pwd_a and the (legitimate) user's typing patterns, the login program measures a set of features whenever a user types a password. Empirically we will examine the use of keystroke duration and latency between keystrokes as features of interest, but other features (e.g., force of keystrokes) could be used if they can be measured by the login program. Abstractly, we represent a feature by a function $\phi : A \times \mathbb{N} \to \mathbb{R}^+$ where $\phi(a, \ell)$ is the measurement of that feature during the ℓ-th (successful or unsuccessful) login attempt to account a. For example, if the feature ϕ denotes the latency between the first and second keystrokes, then $\phi(a, 6)$ is that latency on the sixth attempt to log into a. Let m denote the number of features that are measured during logins, and let $\phi_1, \ldots, \phi_m$ denote their respective functions.

Central to our scheme is the notion of a *distinguishing feature*. For each feature ϕ_i, let $t_i \in \mathbb{R}^+$ be a fixed parameter of the system. Also, let $\mu_{a,i}$ and $\sigma_{a,i}$ be the mean and standard deviation of the measurements $\phi_i(a, j_1), \ldots, \phi_i(a, j_h)$ where $j_1, \ldots, j_h$ are the last h successful logins to the account a and $h \in \mathbb{N}$ is a fixed parameter of the system. We say that ϕ_i is a distinguishing feature for the account (after these last h successful logins) if $|\mu_{a,i} - t_i| > k\sigma_{a,i}$ where $k \in \mathbb{R}^+$ is a parameter of the system. If ϕ_i is a distinguishing feature for the account a, then either $t_i > \mu_{a,i} + k\sigma_{a,i}$, i.e., the user consistently measures below t_i on this feature, or $t_i < \mu_{a,i} - k\sigma_{a,i}$, i.e., the user consistently measures above t_i on this feature.

3.2 Security goals

In our login architecture, the system stores information per account that is accessed by the login program to verify attempts to log in. This information is necessarily based on pwd_a and hpwd_a, but will not include either of these values themselves. This is similar to Unix systems, for example, where the `/etc/passwd` file contains the salt for that password and the result of encrypting a fixed string with a key generated from the password and salt. In our login architecture, the information stored per account will be more extensive but will still be relatively small.

The primary attacker with which we are concerned is an "offline" attacker who captures this information stored in the system, and then uses this information in an offline effort to find hpwd_a (and pwd_a). A first and basic requirement is that any such attack be at least as difficult as exhaustively searching for pwd_a in a traditional Unix setting where the attacker has `/etc/passwd`. In particular, if the user chooses pwd_a to be difficult for an attacker to find using a dictionary attack, then hpwd_a will be at least as secure in our scheme.

A more ambitious goal of our scheme is to increase the work that the attacker must undertake by a considerable amount even if pwd_a is chosen poorly, i.e., in a way that is susceptible to a dictionary attack. The amount of additional work that the attacker must undertake in our scheme generally grows with the number of distinguishing features for the account (when the attacker captured the system information). On one extreme, if there are no distinguishing features for the account, then the attacker can find pwd_a and hpwd_a in roughly the same amount of time as the attacker would take to find pwd_a in a traditional Unix setting. On the other extreme, if all m features are distinguishing for the account, then the attacker's task can be slowed by a multiplicative factor up to 2^m. In Section 7, we describe an empirical analysis that sheds light on what this slowdown factor is likely to be in practice. In addition, we show how our scheme can be combined with salting techniques, and so the slowdown factor that our scheme achieves is over and above any benefits that salting offers.

A second attacker that we defend against with our scheme is an "online" attacker who learns pwd_a (e.g., by observing it being typed in) and then attempts to log in using it. Our scheme makes this no easier and typically harder for this attacker to succeed in logging in.

4 Overview

In this section we give an overview of our technique for generating hpwd_a from pwd_a and user keystroke features. When the account a is initialized, the initialization program chooses the value of hpwd_a at random from $\mathbb{Z}_q$, where q is a fixed, sufficiently large prime number, e.g., a q of length 160 bits should suffice. The initialization program then creates $2m$ shares $\{s_i^0, s_i^1\}_{1 \le i \le m}$ of hpwd_a using a secret sharing scheme such that for any $b \in \{0,1\}^m$, the shares $\{s_i^{b(i)}\}_{1 \le i \le m}$ can be used to reconstruct hpwd_a. (Here, $b(i)$ is the i-th bit of b.) These shares are arranged in an "instruction table".

	$< t_i$	$\ge t_i$
1	s_1^0	s_1^1
2	s_2^0	s_2^1
$\vdots$	$\vdots$	$\vdots$
m	s_m^0	s_m^1

The initialization program encrypts each element of both columns (i.e., the "$< t_i$" and "$\ge t_i$" columns) with pwd_a. This (encrypted) table is stored in the system. In the ℓ-th login attempt to a, the login program uses the entered password text pwd' to decrypt the elements of the table, which will result in the previously stored values only if $\mathsf{pwd}_a = \mathsf{pwd}'$. For each feature ϕ_i, the value of $\phi_i(a, \ell)$ indicates which of the two values in the i-th row should be used in the reconstruction to find hpwd_a: if $\phi_i(a, \ell) < t_i$, then the value in the first column is used, and otherwise the value in the second column is used. In the first logins after initialization, the value in either the first or second column works equally well. However, as distinguishing features ϕ_i for this account develop over time, the login program perturbs the value in the second column of row i if $\mu_{a,i} < t_i$ and perturbs the value in the first column of row i otherwise. So, the reconstruction to find hpwd_a in the future will succeed only when future measurements of features are consistent with the user's previous distinguished features.

In this way, our scheme helps defend against an online attacker who learns (or tries to guess) pwd_a and then attempts to log into a using it. Unless this attacker can mimic the legitimate user's keystroke behavior for the account's distinguishing features, the attacker will fail in logging into the account. Moreover, numerous prior studies have shown that

keystroke dynamics tend to differ significantly from user to user (see Section 2), and so typically the online attacker will fail in his attempts to log into a. Thus, the security analysis in the rest of this paper will focus on the offline attacker.

Not any secret sharing scheme satisfying the properties described above will suffice for our technique, since to defend against an offline attacker, the shares must be of a form that does not easily reveal if a guessed password pwd' successfully decrypts the table. In the following sections, we present instances of our technique using two different sharing schemes.

Our scheme can be easily combined with salting to further improve security. A natural place to include a salt is in the validation of hpwd_a just after reconstructing it. For example, when hpwd_a is generated during a login, it could be prepended with a salt before hashing it and testing against a previously stored hash value. The salt can be stored as is typically done today, or may not be stored so that the system must exhaustively search for it [16]. In this case, the extra salt results in an additional work factor that the offline attacker must overcome.

5 An instance using polynomials

In this section, we describe an instance of the technique of Section 4 using Shamir's secret sharing scheme [25]. In this scheme, hpwd_a is shared by choosing a random polynomial $f_a \in \mathbb{Z}_q[x]$ of degree $m-1$ such that $f_a(0) = \mathsf{hpwd}_a$. The shares are points on this polynomial. We present the method in two steps, by first describing a simpler variation and then extending it in Section 5.4 to be more secure against an offline attack.

5.1 Stored data structures and initialization

Let G be a pseudorandom function family [23] such that for any key K and any input x, $G_K(x)$ is a pseudorandom element of $\mathbb{Z}_q^*$.[1] In practice, a likely implementation of G would be $G_K(x) = F(K,x)$ where F is a one-way function, e.g., SHA-1 [26]. There are two data structures stored in the system per account.

- An *instruction table* that contains "instructions" regarding how feature measurements are to be used to generate hpwd_a. More specifically, this instruction table contains an entry of the form $<i, \alpha_{ai}, \beta_{ai}>$ for each feature ϕ_i. Here,

$$\begin{aligned} \alpha_{ai} &= y_{ai}^0 \cdot G_{\mathsf{pwd}_a}(2i) \bmod q \\ \beta_{ai} &= y_{ai}^1 \cdot G_{\mathsf{pwd}_a}(2i+1) \bmod q \end{aligned}$$

  and y_{ai}^0, y_{ai}^1 are elements of $\mathbb{Z}_q^*$. Initially (i.e., when the user first chooses pwd_a), all $2m$ values $\{y_{ai}^0, y_{ai}^1\}_{1 \le i \le m}$ are chosen such that all the points $\{(2i, y_{ai}^0), (2i+1, y_{ai}^1)\}_{1 \le i \le m}$ lie on a single, random polynomial $f_a \in \mathbb{Z}_q[x]$ of degree $m-1$ such that $f_a(0) = \mathsf{hpwd}_a$.

- An encrypted, constant-size *history file* that contains the measurements for all features over the last h successful logins to a for some fixed parameter h. More specifically, if since the last time pwd_a was changed, the login attempts $j_1, \ldots, j_\ell$ to a were successful, then this file contains $\phi_i(a,j)$ for each $1 \le i \le m$ and $j \in \{j_{\ell-h+1}, \ldots, j_\ell\}$. In addition, enough redundancy is added to this file so that when it is decrypted with the key under which it was previously encrypted, the fact that the file decrypted successfully can be recognized.

  This file is initialized with all values set to 0, and then is encrypted with hpwd_a using a symmetric cipher. The size of this file should remain constant over time (e.g., must be padded out when necessary), so that its size yields no information about how many successful logins there have been.

[1] That is, a polynomially-bounded adversary not knowing K cannot distinguish between $G_K(x)$ and a randomly chosen element of $\mathbb{Z}_q^*$, even if he is first allowed to examine $G_K(\hat{x})$ for many $\hat{x}$'s of his choice and is allowed to even pick x (as long as it is different from every $\hat{x}$ he previously asked about).

5.2 Logging in

The login program takes the following steps whenever the user attempts to log into a. Suppose that this is the ℓ-th attempt to log into a, and let pwd' denote the sequence of characters that the user typed. The login program takes the following steps.

1. For each ϕ_i, the login program uses pwd' to "decrypt" α_{ai} if $\phi_i(a,\ell) < t_i$, and uses pwd' to "decrypt" β_{ai} otherwise. Specifically, it assigns

$$(x_i, y_i) = \begin{cases} (2i,\ \alpha_{ai}\ \ G_{\mathsf{pwd}'}(2i)^{-1} \bmod q) & \text{if } \phi_i(a,\ell) < t_i \\ (2i+1,\ \beta_{ai}\ \ G_{\mathsf{pwd}'}(2i+1)^{-1} \bmod q) & \text{if } \phi_i(a,\ell) \ge t_i \end{cases}$$

   The login program now holds m points $\{(x_i, y_i)\}_{1 \le i \le m}$.

2. The login program sets

$$\mathsf{hpwd}' = \sum_{i=1}^{m} y_i \cdot \lambda_i \bmod q$$

   where

$$\lambda_i = \prod_{1 \le j \le m, j \ne i} \frac{x_j}{x_j - x_i}$$

   is the standard Lagrange coefficient for interpolation (e.g., see [19, p. 526]). It then decrypts the history file using hpwd'. If this decryption yields a properly-formed plaintext history file, then the login is deemed successful. (If the login were deemed unsuccessful, then the login procedure would halt here.)

3. The login program updates the data in the history file, computes the standard deviation σ_{ai} and mean μ_{ai} for each feature ϕ_i over the last h successful logins to a, encrypts the new history file with hpwd' (i.e., hpwd_a), and overwrites the old history file with this new encrypted history file.[2]

4. The login program generates a new random polynomial $f_a \in \mathbb{Z}_q[x]$ of degree $m-1$ such that $f_a(0) = \mathsf{hpwd}'$.

5. For each distinguishing feature ϕ_i, i.e., $|\mu_{ai} - t_i| > k\sigma_{ai}$, the login program chooses new random values $y_{ai}^0, y_{ai}^1 \in \mathbb{Z}_q^*$ subject to the following constraints:

$$\begin{aligned} \mu_{ai} < t_i &\ \Rightarrow\ f_a(2i) = y_{ai}^0 \wedge f_a(2i+1) \ne y_{ai}^1 \\ \mu_{ai} \ge t_i &\ \Rightarrow\ f_a(2i) \ne y_{ai}^0 \wedge f_a(2i+1) = y_{ai}^1 \end{aligned}$$

[2] For maximum security, this and the previous step should be performed without writing the plaintext history file to disk. Rather, the login program should hold the plaintext history in volatile storage only.

For all other features ϕ_i—i e , those for which $|\mu_{a,i} - t_i| \leq k\sigma_{a,i}$, or all features if there have been fewer than h successful logins to this account since initialization (see Section 3 1)—the login program sets $y^0_{a,i} = f_a(2i)$ and $y^1_{a,i} = f_a(2i+1)$

6 The login program replaces the instruction table with a new table with an entry of the form $<i, \alpha'_{a,i}, \beta'_{a,i}>$ for each feature ϕ_i. Here,

$$\alpha'_{a,i} = y^0_{a,i} \; G_{\mathsf{pwd}'}(2i) \bmod q$$
$$\beta'_{a,i} = y^1_{a,i} \; G_{\mathsf{pwd}'}(2i+1) \bmod q$$

where $y^0_{a,i}, y^1_{a,i}$ are the new values generated in the previous step

Step 4 above is particularly noteworthy for two reasons First, due to this step, the polynomial f_a is changed to a new random polynomial during each successful login This ensures that an attacker viewing the instruction table at two different times will gain no information about which features switched from distinguishing to non-distinguishing and vice-versa during the interim logins. That is, each time the attacker views an instruction table for an account, either all values will be the same since the last time (if there were no successful logins since the attacker last saw the table) or all values will be different. Second, though generated randomly, f_a is chosen so that $f_a(0) = \mathsf{hpwd}_a$ This ensures that hpwd_a remains constant across multiple logins

Step 5 is also noteworthy, since it shows that whether each feature is distinguishing is recomputed in each successful login So, a feature that was previously distinguishing can become undistinguishing and vice-versa This is the mechanism that enables our scheme to naturally adapt to gradual changes in the user's typing patterns over time

5.3 Security

Consider the "offline" attacker who obtains account a's history file and instruction table, and attempts to find the value of hpwd_a Presuming that the encryption of the history file using hpwd_a is secure, since the values $y^0_{a,i}, y^1_{a,i}$ are effectively encrypted under pwd_a, and since pwd_a is presumably chosen from a much smaller space than hpwd_a, the easiest way to find hpwd_a is to first find pwd_a Thus, to argue the benefits of this scheme, we have to show two things First, we have to show that finding pwd_a is not made easier in our scheme than it is in a typical environment where access is determined by testing the hash of the password against a previously stored hash value. Second, we have to show that the cost to the attacker of finding hpwd_a is generally greater by a significant multiplicative factor

That searching for pwd_a is not made easier in our scheme is clear The attacker has available only the instruction table and the encrypted history file. Since there is a row in the instruction table for each feature (not just those that are distinguishing for a), and since the contents of each row are pseudorandom values, the rows reveal no information about pwd_a And, all other data available to the attacker is encrypted with hpwd_a

The more interesting security consideration in this scheme is how much security it achieves over a traditional password scheme. Suppose that the attacker captured the history file and instruction table after $\ell \geq h$ successful logins to a, and let d be the number of distinguishing features for this account in the ℓ-th login When guessing a password pwd', the attacker can decrypt each field $\alpha_{a,i}$ and $\beta_{a,i}$ using pwd' to yield points $(2i, \hat{y}^0_{a,i})$ and $(2i+1, \hat{y}^1_{a,i})$, respectively, for $1 \leq i \leq m$ Note that $\hat{y}^0_{a,i} = y^0_{a,i}$ and $\hat{y}^1_{a,i} = y^1_{a,i}$, where $y^0_{a,i}, y^1_{a,i}$ are as generated in Step 5, if and (with overwhelming probability) only if $\mathsf{pwd}' = \mathsf{pwd}_a$. Therefore, there exists a bit string $b \in \{0,1\}^m$ such that $\{(2i + b(i), \hat{y}^{b(i)}_{a,i})\}_{1 \leq i \leq m}$ interpolates to a polynomial $\hat{f}$ with $\hat{f}(0) = \mathsf{hpwd}_a$, if and only if $\mathsf{pwd}' = \mathsf{pwd}_a$. Consequently, one approach that the attacker can take is to enumerate through all $b \in \{0,1\}^m$ and, for each $\hat{f}$ thus computed, see if $\hat{f}(0) = \mathsf{hpwd}_a$ (i e , if $\hat{f}(0)$ will decrypt the history file). This approach slows down the attacker's search for hpwd_a (and pwd_a) by a multiplicative factor of 2^m In practice, the slowdown that the attacker suffers may be substantially less because user typing patterns are not random. In Section 7, we use empirical data to quantify the degree of security achieved against this form of attack, and show that it is nevertheless substantial

However, the attacker has potentially more powerful attacks against this scheme using the $2m$ points $\{(2i, \hat{y}^0_{a,i}), (2i+1, \hat{y}^1_{a,i})\}_{1 \leq i \leq m}$, due to the following contrast On the one hand, if $\mathsf{pwd}' \neq \mathsf{pwd}_a$, then with overwhelming probability, no $m+1$ points will lie on a single degree $m-1$ polynomial, i e , each subset of m points interpolates to a different polynomial with a different y-intercept (not equal to hpwd_a). On the other hand, if $\mathsf{pwd}' = \mathsf{pwd}_a$, then there are $2m - d \geq m$ points that all lie on a polynomial f of degree $m-1$ (and $f(0) = \mathsf{hpwd}_a$), in particular if $d < m$, then there are at least $m+1$ points that all lie on some such f. Asymptotically (i.e , as m grows arbitrarily large), it is known that the second case can be distinguished from the first in $O(m^2)$ time if $d \leq (2-\sqrt{2})m \approx .585m$ using error-correcting techniques [7]. These techniques do not directly break our scheme, since our analysis in Section 7 suggests that for many reasonable values of k, d will typically be too large relative to m for these techniques to succeed (unless the attacker captures the account information before the account is used). Moreover, typically m will be too small in our scenario for these techniques to offer benefit over the exhaustive approach above. However, because these techniques might be improved with application-specific knowledge—e g , that in the second case, at least one of $(2i, \hat{y}^0_{a,i})$ and $(2i+1, \hat{y}^1_{a,i})$ lies on f—it is prudent to look for schemes that confound the use of error-correcting techniques. This is the goal of Section 5 4

5.4 A variation using exponentiation

In this section we present a minor variation of the scheme presented in Sections 5 1–5.2, to which we refer as the "original" scheme below. The scheme of this section is more secure in several ways that will be described below.

Let p be a large prime such that computing discrete logarithms modulo p is computationally intractable (e g., choose p of length 1024 bits) and such that q divides $p-1$. Also, let g be an element of order q in $\mathbb{Z}^*_p$ The main conceptual differences in this variation are that hpwd_a is defined to be $g^{f_a(0)} \bmod p$, and rather than storing $\alpha_{a,i}$ and $\beta_{a,i}$ in the instruction table, the values

$$\gamma_{a,i} = g^{\alpha_{a,i}} \bmod p$$
$$\delta_{a,i} = g^{\beta_{a,i}} \bmod p$$

are stored instead. Intuitively, since the attacker cannot compute discrete logarithms modulo p, this hides $y^0_{a,i}, y^1_{a,i}$ from him even if he guesses pwd_a.

There are a number of reasons to prefer this variation to the original in practice. First, this modified instruc-

DOCKET ALARM

Explore Litigation Insights

Docket Alarm provides insights to develop a more informed litigation strategy and the peace of mind of knowing you're on top of things.

Real-Time Litigation Alerts

Keep your litigation team up-to-date with **real-time alerts** and advanced team management tools built for the enterprise, all while greatly reducing PACER spend.

Our comprehensive service means we can handle Federal, State, and Administrative courts across the country.

Advanced Docket Research

With over 230 million records, Docket Alarm's cloud-native docket research platform finds what other services can't. Coverage includes Federal, State, plus PTAB, TTAB, ITC and NLRB decisions, all in one place.

Identify arguments that have been successful in the past with full text, pinpoint searching. Link to case law cited within any court document via Fastcase.

Analytics At Your Fingertips

Learn what happened the last time a particular judge, opposing counsel or company faced cases similar to yours.

Advanced out-of-the-box PTAB and TTAB analytics are always at your fingertips.

API

Docket Alarm offers a powerful API (application programming interface) to developers that want to integrate case filings into their apps.

LAW FIRMS
Build custom dashboards for your attorneys and clients with live data direct from the court.

Automate many repetitive legal tasks like conflict checks, document management, and marketing.

FINANCIAL INSTITUTIONS
Litigation and bankruptcy checks for companies and debtors.

E-DISCOVERY AND LEGAL VENDORS
Sync your system to PACER to automate legal marketing.