PolyPasswordHasher: Protecting Passwords In The Event Of A Password File Disclosure

Justin Cappos

New York University
jcappos@nyu.edu

Santiago Torres

New York University
santiago@nyu.edu

***Abstract*—Over the years, we have witnessed various password-hash database breaches that have affected small and large companies, with a diversity of users and budgets. The industry standard, salted hashing (and even key stretching), has proven to be insufficient protection against attackers who now have access to clusters of GPU-powered password crackers. Although there are various proposals for better securing password storage, most do not offer the same adoption model (software-only, server-side) as salted hashing, which may impede adoption.**

In this paper, we present PolyPasswordHasher, a software-only, server-side password storage mechanism that requires minimal additional work for the server, but exponentially increases the attacker's effort. PolyPasswordHasher uses a threshold cryptosystem to interrelate stored password data so that passwords cannot be individually cracked. Our analysis shows that PolyPasswordHasher is memory and storage efficient, hard to crack, and easy to implement. In many realistic scenarios, cracking a PolyPasswordHasher-enabled database would be infeasible even for an adversary with millions of computers.

***Keywords*—password, password-hashing, authentication, password cracking, cryptography.**

I. Introduction

Password database thefts are increasingly common and continue to put users at risk. Both individuals and businesses want passwords to be stored efficiently, yet in a way that minimizes the impact of a password database disclosure. Major corporations, in particular, have encountered significant challenges in trying to protect their customers' and clients' identities and accounts. Reflecting the reach of the problem, a hacker on a Russian forum was selling a database with more than a billion password hashes obtained from different password databases, primarily through SQL injection [1]. Some companies and organizations that have been successfully hacked include: the Social Security Administration, Hotmail, LastFM, Formspring, ScribD, the New York Times, Nvidia, Evernote, LinkedIn, Billabong, Gawker, Linode, ABC, Yahoo!, eHarmony, LivingSocial, and Twitter [1]. Globally – despite ongoing efforts to enhance password security and adhere to best practices – password database breaches continue.

Notwithstanding new technologies such as authentication tags, biometric technologies, and NFC-capable devices, the most prevalent user authentication method remains the user password. Security experts have long advocated that user passwords not be stored in plain text but rather, that they be reduced to salted hashes before storing. A salted hash consists of a secure hash and a salt. The secure hash acts as a one-way function that ensures that an attacker cannot easily read the plain-text passwords from disk. Salting inserts a random value that complicates the use of tables that allow hackers to immediately look up passwords to match most hashes. Storing passwords with a salted hash is widely considered to be the best practice because of the level of protection offered and the ease of implementation (e.g., salted hashing requires no additional hardware or client software).

However, storing passwords with a salted hash is not a panacea. That is, when stored password data has been compromised, attackers have proven themselves adept at quickly cracking large numbers of passwords, even when salted-password hashes comprise the stored data. For example, Troy Hunt, a security researcher, showed that cracking 60% of a 40,000-entry salted-SHA1 database can be done in a couple of hours, using a couple of GPUs and an average computer [2].

This paper presents PolyPasswordHasher, a new technique that makes cracking individual passwords infeasible because the stored password-entries are interdependent. We leverage cryptographic hashing and threshold cryptography to combine password hash data with shares so that users unknowingly protect each other's password data. With a small amount of additional work by the server, PolyPasswordHasher increases, by many orders of magnitude, the time an attacker needs to crack passwords.

PolyPasswordHasher is designed to be easy for organizations to adopt. Much like salted hashing, PolyPasswordHasher is a software-only, single-server enhancement that can be deployed on a server without any changes to clients. PolyPasswordHasher relies on simple primitives that are efficient from a storage, memory, and computational standpoint. It integrates with other forms of authentication including: OAuth, hardware tokens, two factor authentication, and fingerprint authentication. PolyPasswordHasher is also easy to implement; two outside developers, whom we had no prior contact with, independently built implementations of PolyPasswordHasher, in different programming languages. Deploying PolyPasswordHasher is a straightforward software installation on a server.

The contributions of this paper are as follows:

- We describe our design of PolyPasswordHasher, a server-side, software-only scheme that protects against an attacker who is able to read all persistent storage on a server, including the complete password file.
- We demonstrate the PolyPasswordHasher has similar performance, storage, and memory requirements to deployed systems.

- We describe a mechanism to shield some user accounts, who may be untrustworthy or choose poor passwords, as long as the passwords from a set of protector accounts are not cracked.
- We analyze different practical scenarios and analyze the properties and limitations of PolyPasswordHasher in these environments.

This paper is organized into eight sections. Sections II and III introduce the threat model and the theoretical background of password storage mechanisms. Sections IV and V provide the design specification and implementation of PolyPasswordHasher. Section VI evaluates PolyPasswordHasher's suitability and resilience to cracking in a variety of scenarios; it also provides information about the implementations performance for different kinds of operations. We describe the relation to PolyPasswordHasher with other related work in Section VII. Finally, Section VIII summarizes and anaylzes our findings described in the paper.

II. Threat Model

We base our threat model on salted hashing, which is the most widely deployed protection scheme. In salted hashing, the user provides a username and password to the server. If a user is locally at the server, this is typically done using a keyboard attached to the device. If a user is remote, the password is usually provided over an encrypted channel, such as HTTPS or SSH. Either way, the username and password are input to the server in plain-text.

For a server to employ salted hashing, the only requirement is that there is software on the server that implements both salting and hashing. Notably, there are no hardware requirements for the server (e.g., hardware tokens or additional servers) and no client software is needed.

We assume that:

- An attacker can read all data that is persisted on disk, including the password database. The attacker cannot read arbitrary memory on the server. As with any scheme that does not require client changes, if an attacker can read arbitrary memory, the attacker can observe the passwords in plain-text. (Fortunately, most reported password database disclosures did not involve an attacker with access to arbitrary memory [1], [3].)
- The server will restart periodically. All data kept in memory is lost at this point and the server must restart using only the data on disk – which is attacker visible.
- The attacker may have a priori knowledge of correct passwords for some user accounts – for example, on sites that allow outside users to register accounts.
- We assume that well known and widely used cryptographic primitives, such as SHA256 and AES, are not breakable by the attacker.

Alternative solutions that use this threat model as well as alternative threat models are discussed in Related Work (Section VII).

III. Background

We briefly review the relevant properties of cryptographic (k, n)-threshold schemes. Although the specific (k, n)-threshold scheme that is used is not fundamental to our work, we describe Shamir Secret Sharing [4], which we use to develop explicit examples within the text.

The (k, n)-threshold scheme

Threshold schemes protect secret information (usually a key) by deriving n different shares from this information. A threshold scheme describes how any k shares (from a set of n total shares) can be used to recover an original secret. The number of needed shares, k, is called the threshold. If fewer than k shares are known, no information about the secret is provided.

Shamir Secret Sharing is an algorithm that describes how a secret is divided into a set of n shares. If a threshold k of shares are input (specified when the secret is divided), the original secret can be reconstructed. To hide a secret, Shamir Secret Sharing computes $k-1$ random coefficients for a $k-1$ degree polynomial $f(x)$ in a finite field (commonly GF-256 or GF-65536). The kth term (commonly the constant term) contains the secret. To compute a share, a value between 1 and the order of the field is chosen. The polynomial is evaluated with x equal to the share value, where the terms x and $f(x)$ are used as the share. To reconstruct the secret from at least k shares, a party can interpolate the values in the finite field to find the constant term (i.e., the secret). In practice, interpolation is often computationally optimized so that only the constant term is recovered.

Suppose that a secret, 235, is to be hidden so that it can only be reconstructed if three shares are provided. Because the threshold is three, two random terms are first generated (24 and 182) to build a GF-256 polynomial, such as $f(x) = 24x^2+182x+235$. Shares can then be generated by computing x and $f(x)$, such as: (1, 92), (2, 148), (3, 37), (4, 69), etc. A party that has at least three shares can interpolate to reconstruct the full polynomial of $f(x)$ and thus, the secret (235).

It is possible to generate additional shares after recovering the secret, (e.g., if Lagrange interpolation is performed during reconstruction). This makes share recovery slightly more computationally complex but also makes it possible (and efficient) to generate additional shares simply by evaluating f(x) for the specified share. This means that all shares do not need to be created initially — they may instead be added (or recovered) after the secret has been reconstructed.

In many cases, the secret will be larger than the size of the finite field. A large secret can be stored by breaking it into segments that are the size of the finite field (often one or two bytes) and applying the above technique separately to each segment. The same share number, x, is typically used for each share $f_i(x)$. This simplifies — and effectively hides — the fact that a secret has only a limited size. An integrity check can be added to detect whether an incorrect share has been provided. As was previously described, when given a set of any k distinct shares, whether valid or invalid, Shamir Secret Sharing will produce a polynomial of the appropriate length. This means that if any share is invalid, its resulting polynomial will be incorrect. To avoid this problem, implementations of Shamir

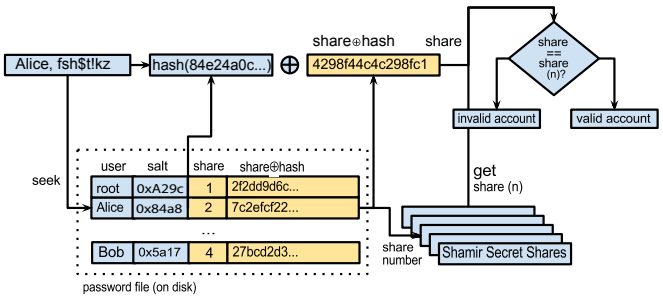

Figure 1: Account verification.

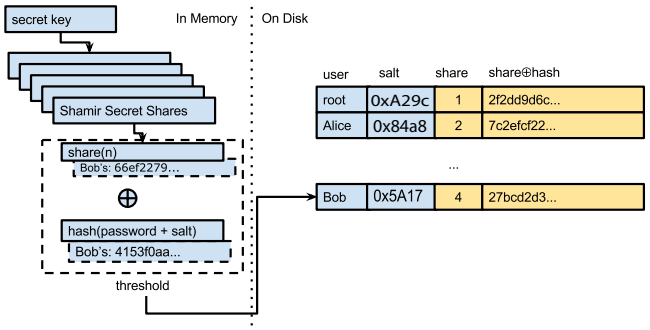

Figure 2: The basic PolyPasswordHasher algorithm showing how an entry is created

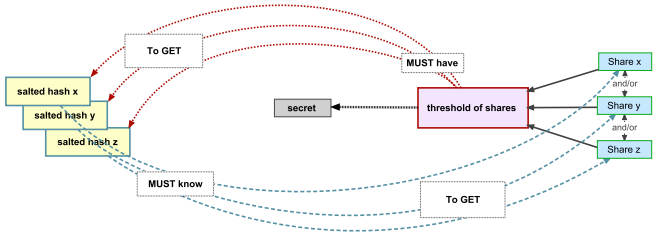

Figure 3: An attacker: (1) Cannot obtain salted hashes (needed to crack passwords) without a threshold of shares AND (2) Cannot obtain a share without knowing the salted hash (password) for a protector account.

Secret Sharing typically store an integrity check by appending the hash of the secret to the secret; this check detects if an incorrect share has been provided. When the shares are reconstructed, the additional integrity check provides verification that the correct shares were given.

IV. PolyPasswordHasher: A New Technique for Password Verification

The goal of PolyPasswordHasher is to make cracking individual passwords infeasible. It provides a way of preventing an attacker from validating a password hash. At its core, PolyPasswordHasher aims to protect password hashes by combining a share (derived using a threshold cryptosystem) with a salted password hash and then storing this combined value in the password database. Neither the share nor the password hash is stored on disk and, as we discuss in more detail below, an attacker cannot recover either piece with only the password database. Cracking a password that is stored by PolyPasswordHasher requires that the attacker know a threshold of passwords; this effectively makes the passwords in a database interdependent. Our goal is to ensure that so long as an attacker does not know a threshold of passwords, no password in a database can be cracked.

To make passwords interrelate, PolyPasswordHasher functions differently from a salted hash. A typical salted hash database stores a Username, Salt, and a Salted Hash. A PolyPasswordHasher database, rather than storing a salted hash, stores the secure hash XORed with the share (share⊕hash). The resulting PolyPasswordHasher database also holds an extra field called the share number. This field indicates which share was XORed with which salted hash.

When prompted to validate a password, the server XORs the salted hash with the stored data and determines whether the result is a valid share of the threshold cryptosystem. For example, in Figure 1, Alice has provided her password ('fsh$t!kz') and the username 'Alice'. The server and computes the salted hash of her password ('4298f44d...') and reconstructs Alice's share ('3e773b6f...') using her share number (2). The server XORs the share and Alice's salted password hash together and compares them to the value stored in the share⊕hash field in the password database ('7cefcf22...'). If they match, then the password provided was correct.

Account creation involves creating a share and XORing it with the salted hash of the password before storing it on disk For example, in Figure 2, Bob registers an account with his password ('Tr4mP0l1ne') and username ('Bob'). The server computes a salt ('0x5a17') and the salted hash of Bob's password ('4153f0aa...'). The server knows the secret and knows Bob's share number, 4. It computes Bob's share ('66ef2279...') and this value is XORed with Bob's passwords salted hash; that value is compared to the value stored in the database ('27bcd2d3...'). If these match, then the password provided was correct.

In PolyPasswordHasher, each share protects a salted hash and each salted hash protects a share, unless a threshold of shares is known (as is shown in Figure 3). Suppose that an attacker has obtained the password database and knows some set of account passwords (x,y,z). For each known password, (illustrated in Figure 4) the hacker can compute the salted hash and XOR this with the database entry to obtain the corresponding share. If the attacker does not have a threshold of shares, the attacker cannot generate a share for another account (e.g., 'share a'). As a result, the attacker cannot access share a's password's salted hash and cannot crack a's password. Or, suppose that a server has a threshold of correct passwords. The server now has enough information to validate a threshold of shares and can use those to recover the secret. The server could then reconstruct any share and thus, recover the salted hash for any account's password – an important step in validating passwords. Because shares protect passwords, an attacker who does not have an adequate number of correct passwords (and thus shares) cannot feasibly crack passwords individually.

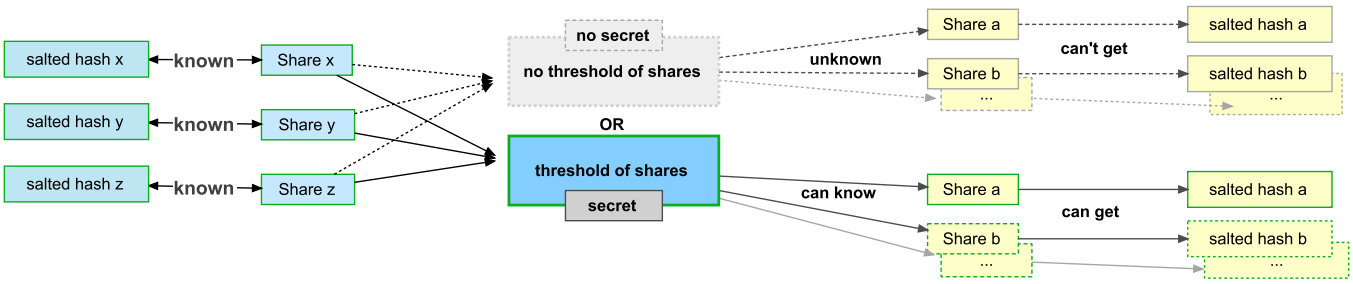

Figure 4: If an attacker knows some number of shares he can get the corresponding salted hashes; if he knows the salted hashes, he can get the corresponding shares; and, if he has a threshold of these, he can get the secret.

The basic scheme, as outlined so far, covers PolyPasswordHashers core functions. In the remaining subsections we address how PolyPasswordHasher checks passwords without having a threshold of correct passwords (such as after a restart) and how it ensures that all accounts, even those that could be created by an attacker, are protected. To more fully describe the full PolyPasswordHasher algorithm, we begin, in Section IV-A, with how PolyPasswordHasher functions in situations where the server has already validated a threshold of correct passwords and how, after reaching this threshold, the server proceeds to login normal users, in a phase we call normal operation. Following this, Section IV-B discusses how PolyPasswordHasher performs differently when a threshold of correct passwords have not yet been provided and needs to acquire that threshold (e.g. after a reboot). Section IV-C discusses how the system transitions between bootstrapping and normal operation and in Section IV-D, we describe how unlikely situations, such as the loss of a large number of account passwords, are handled.

A. *Normal operation of PolyPasswordHasher*

Given that a server with a threshold of shares can effectively recover any salted hash, if every account protected a share, every password would play an important role in protecting the security of the database. However, not all accounts should necessarily be trusted to protect shares. For example, a forum may allow any user to register an account and any party, including an attacker, to register any number of user accounts. If these accounts each protected a share, an attacker could easily crack the password database.

PolyPasswordHasher enables one group of passwords, which we call protector accounts, to protect the remaining passwords in the password database; these we call shielded accounts. Bob's account (who we started following in the previous example) is an example of a protector account (Figure 5). An administrator will define a threshold value of protector accounts so that if an attacker does not know a threshold of protector passwords, Bob's salted hash cannot be recovered. Bob's password hash protects his share and in turn, Bob's share helps to protect the other shares (such as shares 1,2), which in turn protect their corresponding password hashes (for root and Alice). These shares serves to protect the full password database, including shielded accounts (Trudy and Luke).

Since a shielded account does not protect a share, it is not XORed with one. Nonetheless, these accounts are still protected by shares, or more precisely, the secret. PolyPasswordHasher uses the secret as an encryption key and with it, encrypts the password's salted hash. For these accounts, the resulting value (but not the secret or salted hash) is stored in the password database. For example, in Figure 5, Trudy's password hash (a9e32543...) is encrypted using the secret (9d380eb6...); the result (8198a5fd...) is stored in the database. There are two important points to note. First, if an attacker knows another shielded account's password (e.g., Luke's), this does not substantially help with cracking Trudy's password. The attacker would need to break the symmetric encryption algorithm. Also, knowing Luke's password would not substantially help the attacker crack the secret or other shares in the password. Once again, the attacker would need to break the symmetric encryption algorithm. Thus, shielded accounts cannot be effectively cracked unless an attacker knows a threshold of protector account passwords.

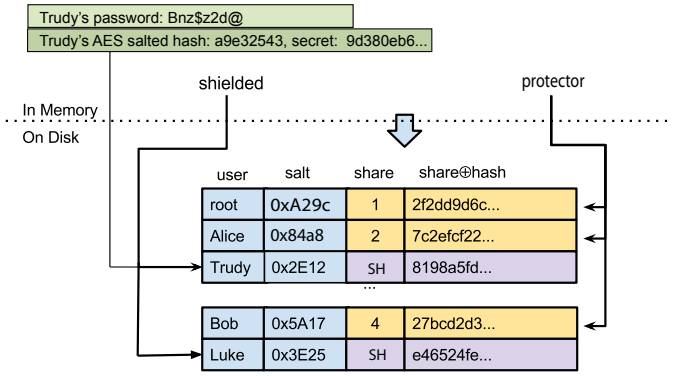

Figure 5: A PolyPasswordHasher store with protector and shielded accounts. Shielded accounts are displayed with a share number of SH.

1) PolyPasswordHasher's algorithm: Algorithm 1 details the processes of creating a user accounts. The relevant operations for normal operation (lines 2-13) are discussed in this section. Bootstrapping operations (lines 14-23) are deferred to the next section.

Creating Accounts. Adding an account works as is shown in Algorithm 1. As the algorithm describes, the username, salt,

Algorithm 1 Account creation pseudocode.

```
1: function CREATEACCOUNT(username, salt, saltedPasswordHash, isProtectorAccount)
2:   // check whether we are under normal operation
3:   if normalOperation then // Section IV and IV-A
4:     if isProtectorAccount then
5:       // Obtain a share from the share cryptosystem
6:       shareNumber, share = SecretShares.getShare()
7:       // Combine the share with the hash
8:       passwordEntry = share ⊕ saltedHash
9:       shareID = shareNumber
10:    else // Shielded Account (Section IV-A)
11:      Key = SecretShares.getSecret()
12:      passwordEntry = AES.encrypt(saltedPasswordHash, Key)
13:      shareID = SHIELDED
14:  else // Bootstrapping (Section IV-B)
15:    if isProtectorAccount then
16:      raise AccountCreationError
17:    else
18:      shareID = BOOTSTRAP
19:    passwordEntry = saltedPasswordHash
20:  // Isolated Validation (Section IV-B)
21:  isolatedCheckBits = saltedPasswordHash.getSuffix(IC_BITS)
22:  passwordEntry += isolatedCheckBits
23:  store (username, salt, shareID, passwordEntry)
```

Algorithm 2 Account verification pseudocode.

```
1: function VERIFYACCOUNT(username, saltedPasswordHash, shareID, passwordEntry)
2:   if normalOperation then
3:     if isProtector(shareID) then// Section IV
4:       share = passwordHash ⊕ passwordEntry
5:       return SecretShares.computeShare(shareID) == share
6:     else// shielded account, Section IV-A
7:       // Encrypt the obtained hash and compare
8:       Key = SecretShares.getSecret()
9:       encryptedHash = AES.encrypt(passwordHash, Key)
10:      return passwordEntry == encryptedHash
11:  else// Bootstrapping, Section IV-B
12:    // verify account created during bootstrap
13:    if shareID == BOOTSTRAP then
14:      return passwordHash == passwordEntry
15:    if not passwordHash.endsWith(isolatedCheckBits) then
16:      return False
17:    if isProtector(shareID) then
18:      share = passwordHash ⊕ passwordEntry
19:      SecretShares.cacheShare(share)
20:    return True
```

the passwords salted hash, and a boolean are used to indicate whether or not the account should be a protector account. If the account is a protector account, an unused share is found (line 6) and the salted hash is XORed with it (line 8) before the share number and resulting value are stored (line 23).

Since a shielded account does not protect a share, those accounts are instead encrypted with the secret (lines 11-12). The share field is set to a special value that indicates a shielded account (line 13). As before, this information is then stored in the database (line 23).

It is important to prevent users, particularly protector accounts, from using bad passwords. Similar to many deployed systems, PolyPasswordHasher employs simple techniques to weed out extremely bad passwords. When creating an account, users input a password they want to use, but PolyPasswordHasher checks this password to ensure that it is not too weak (e.g., "letmein" or "password"). This is done by checking the requested password against a list of commonly used passwords (the 64K most popular). The proposed password will be rejected if it is on that list. PolyPasswordHasher also enforces constraints on the password length and composition (number of lowercase and uppercase letters, numbers, and symbols). The purpose is not to ensure that passwords are immensely strong, but to prevent the use of protector account passwords that are trivial to guess. These constraints are already common in many systems, such as those that aim to prevent passwords from being cracked by brute force over the network.

Process for Verifying Passwords. Assuming that the server holds the right number of valid passwords (and thus shares), it will be able to recover the remaining shares, as shown in Algorithm 2. To verify a protector account (lines 4-5), the server first computes the salted hash of the users password and XORs it with the passwordEntry. If the passwordEntry XORed with the salted hash is the share, then the password is correct.

For a shielded account, password verification differs because the salted hash is encrypted instead of being XORed with a share. Lines 8-10 of Algorithm 2 describe how to compare the provided password hash. The password hash is encrypted with the secret and, if the encrypted value matches the share⊕hash field, the correct password was provided.

Changing a user's password / password recovery Changing a password is a similar process to that used in a salted hash system, for both protector accounts and shielded accounts. Similar to existing systems, the procedure for changing a password may require validating the existing password before allowing a replacement to be generated. Changing the password involves creating a new entry with the same username and share number. (This is nearly identical to account creation.) A new salt is generated and hashed along with the new password. Since the hash has changed, the fourth field will also change. Once the new password entry is computed, it replaces the original stored data. The user may then log in normally with their new account.

B. Bootstrapping After a Reboot

When a system restarts, PolyPasswordHasher cannot validate or create accounts as it normally would, because a threshold of valid accounts has not been reached. Because PolyPasswordHasher stores shares in memory, not on disk, these are lost during reboot. This means that a server does not know the secret and cannot compute arbitrary shares. As a result, when bootstrapping, neither protector account nor shielded accounts may be verified or created using the methods described above; account creation and verification procedures are different when the system is bootstrapping.

1) Bootstrap account creation: New accounts may also be created during bootstrapping. To do this, the new account is added to the database along with the salted hash. While the system is bootstrapping, the new account is available to use. In the interim, these passwords will be created (line 19 of Algorithm 1) and validated (lines 18-19 of Algorithm 2) in the same manner as passwords stored in a system that uses salted hashes.

Although it would be easy to support protector account creation during bootstrap, PolyPasswordHasher does not do so (line 16 of Algorithm 1). The reason is that an attacker who can read the password database would also be able to read the salted password hash. If the attacker can later read the password database during normal operation, the attacker could use that salted password hash to recover the share.

2) Isolated Validation: When started, a PolyPasswordHasher system does not have enough protector passwords (and thus, shares) to recover the secret. At this point it is not possible to validate accounts following the same process as PolyPasswordHasher's normal operation (IV-A). Here we describe how isolated validation can be used to check logins even without the secret. Isolated validation is a process that leaks a configurable number of bits of the salted hash by using a slow hash algorithm. It implements a mechanism wherein it is possible, but extremely unlikely, that an attacker could access an account using an incorrect password. In the next section we discuss the isolated validation mechanism, but defer discussion of its algorithm until Section VI-E.

As it bootstraps, PolyPasswordHasher collects shares from protector logins. The number of logins required to recover the secret will have been configured by the system administrator, with the threshold normally set to a low number (e.g., three). Once the threshold has been reached, PolyPasswordHasher will finish bootstrapping (Section IV-B). Meanwhile, before this threshold is reached, isolated validation makes use of an isolated-check bits field to authenticate user passwords.

In this scheme, illustrated in the upper right corner of Figure 6, the password database contains isolated-check bits. These bits are used to verify logins before a threshold is reached and the same process is used for both protector and shielded accounts. The user's password is hashed using the isolated validation hash function. This function returns a small number of bits of the hash (such as 24 bits of a SHA256 hash) and typically involves many iterations of a secure hash function. If the isolated-check bits field of the password database match the isolated validation hash functions output (line 12 of Algorithm 2), the user is allowed to log in.

For example, Figure 7 illustrates verification while the system is bootstrapping. Bootstrap accounts, such as Wilson's account are validated in an identical way to a salted hash system. Wilson's password is salted and hashed and this is compared with the value stored in the database. If it matches, Wilson is logged in.

Protector accounts and shielded accounts, like those of Trudy or Alice may also log in while the system is bootstrapping (Figure 7) using isolated validation. The isolated validation hash function is computed over the provided password and the isolated-check bits field is checked. If these values match, the user is allowed to log in. Bootstrap accounts, such as Wilson's account are validated in an identical way to a salted hash system. Wilson's password is salted and hashed and this is compared with the value stored in the database. If it matches, Wilson is logged in.

Isolated validation represents a tradeoff for administrators. Using isolated validation makes the system available immediately after a reboot. However, when using a small number of isolated-check bits, there is the potential for an incorrect

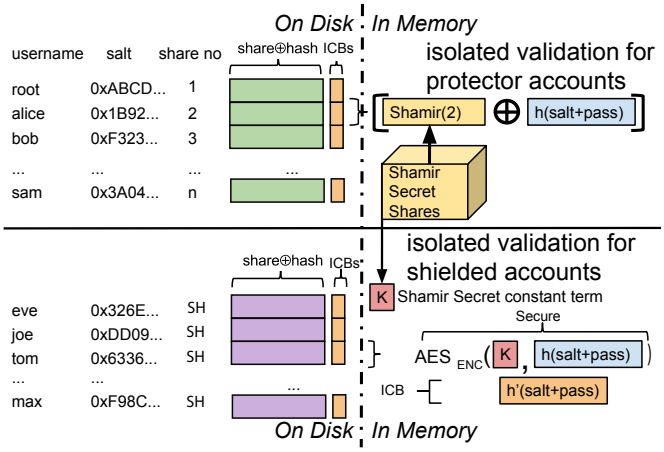

Figure 6: This figure shows validation using isolated validation. The isolated-check bits are stored on disk. This allows verification of accounts before a threshold of correct passwords is provided.

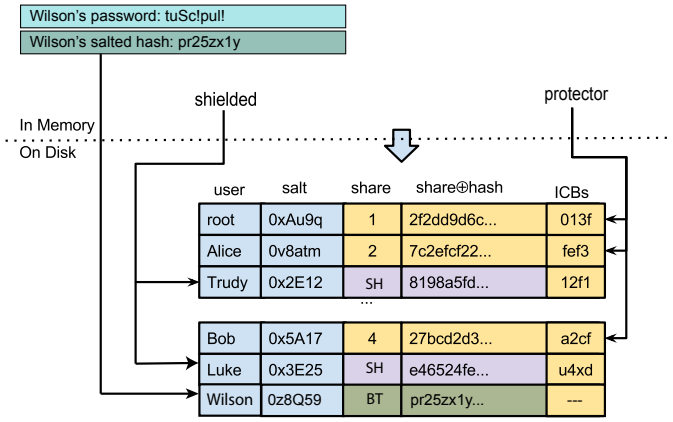

Figure 7: Wilson's account was created during bootstrap, so a regular salted hash is stored instead.

authentication during the bootstrapping phase. Alternatively a large number of isolated-check bits impacts the confidentiality of the password database in the event of a theft, by making the passwords easier to crack. Thus in some scenarios different settings are appropriate, possibly even for protector accounts and shielded accounts in the same database. We explore this tradeoff more in Section VI-E

C. Transitioning From Bootstraping to Normal Operation

As the system bootstraps, the server batches shares from protector accounts (line 14 of Algorithm 2). After the system has a threshold of shares, it is possible to recover the secret. The server performs Lagrange interpolation, which allows the server to not only recover the secret but also, to generate arbitrary shares (both are needed to check shielded accounts).

Recall that accounts created during the bootstrap phase contain BOOTSTRAP for their share number and have a salted hash stored in the database. When the system recovers the

secret and transitions to normal operation, these accounts will have their password hashes encrypted with the secret and their share number field set to SHIELDED. This transforms these accounts to the same state as shielded accounts that are created in normal operation.

Also, all account logins that were processed during bootstrapping are now checked to be certain that the correct passwords were provided. This step is needed because it is possible that an attacker could find a password that is invalid and yet, matches the isolated-check bits. (We examine the feasibility and computational cost of this in Section VI) The salted password hash for shielded account logins are cached in memory and the passwords are verified once a threshold is reached.

For a protector account, if a provided password hash matches the isolated-check bits, but the password is not correct, the recovered share is not invalid. The incorrect share will be detected when a threshold of shares are obtained because the integrity check on the secret will fail. If this integrity check fails, the administrator is notified. The system can still enter normal operation once a threshold of valid shares are obtained.

D. Handling Rare Events

At an administrator's behest, accounts may be switched between shielded and protector without user intervention. To do this, the server must be in normal operation. The server can then recover the salted secure hash. This salted secure hash can then be re-encoded (using a share for an account that becomes a protector account; or encrypted, for an account that becomes shielded) and the new entry can be stored. This transitions the account from a protector to shielded (or vice versa).

Recovering data if all protector accounts are lost. If not enough known threshold users can be verified, the salted hashes are lost forever and accounts cannot be validated using the technique we described in Section IV-B2. However, this does not mean that the system is unusable because isolated validation allows users to log in. Furthermore, mechanisms like root password recovery that are done through the console will still work, and will allow any data on the system to be accessed.

Trusting a protector account with multiple shares. A single account may optionally provide access to multiple shares. To do this, a single user can have multiple rows in the table. Each row will have a different share⊕hash entry and thus protect a different share. Each entry must also have a different salt to ensure that a different hash value is XORed with each share. When the user provides their password, the value can be used to recover the share in each row by XORing the salted password hash with each share.

Detecting an isolated-check bits match of an incorrect password. If the system is in normal operation, it will detect that the password does not match. However, PolyPasswordHasher will always check the isolated-check bits for an entered password (for clarity, not depicted in Algorithm 2). If the password is incorrect, but the isolated-check bits match, this indicates that an attacker has almost certainly stolen the password database but has not (yet) cracked the password. This generates an alert to the administrator to notify her of the likely breach.

Language	Lines of Code	Author
C	598	Local1
Python	391 (+83)	Local2
Django	786	Local1 + External1
Ruby	437	External2
PHP*	N/A	External3

Table I: Different implementations of PolyPasswordHasher for different Languages and Frameworks, the PHP implementation has not been publicly released

V. Implementation and Limitations

Our reference implementation for PolyPasswordHasher is available with an MIT license at https://polypasswordhasher.poly.edu. It utilizes a 16 byte salt, with SHA256, to compute password hashes. The Shamir Secret Sharing routines utilize GF256 as the underlying field (by encoding each byte as a separate share). The code base for PolyPasswordHasher is 120 lines of Python code. This code handles the functionality described in Section IV, including support for protector, bootstrapping, and shielded accounts; reading/writing a database to disk; changing passwords; and detecting isolated validation of incorrect passwords. PolyPasswordHasher relies on mathematical libraries such as GF256 operations, Lagrange interpolation, and polynomial math code – 391 lines of Python and 83 lines of C code. Additionally, the implementation uses standard python libraries for functionalities such as SHA256 and AES.

Several outside developers, unbeknownst to us (until they shared their work with us) implemented versions of PolyPasswordHasher in Ruby and PHP. They created their implementations based on our Python reference implementation and specification that is available in our GitHub repository. Table I describes these outside PPH implementations. Other external developers have uploaded fixes and improvements to our code when they wanted it to be compatible with their environment.

A. Handling Alternate Authentication Mechanisms

Passwords are not the only mechanism for logging into modern systems. For example, it is common for users to have a private key to log in over ssh, biometric-based authentication holds substantial promise, smart cards are often used to hide user credentials, and single sign-in systems like OAuth and OpenID are commonplace and provided by many major websites. Any practical protection mechanism needs to operate in conjunction with such techniques.

Handling non-password logins for shielded user accounts is trivial because it requires no changes to the system. These non-password logins can be handled using existing login mechanisms, without any modification.

For protector accounts, it is possible to view the authentication process as decrypting information using a secret that has been stored by a remote party (as in fact many such systems are implemented). Essentially, we can encrypt the Shamir Secret Share with the users key and check that the resulting share is correct. For example, suppose that an administrator has a public/private keypair that is used to log in (as is common with SSH private key or smart card based authentications).

Instead of XORing the Shamir Share with the salted hash of the password, the share can be encrypted with the users private key. When the administrator presents her public key to login, it decrypts the original share.

Prior work on deriving a private key from (noisy) data [34] is relevant when authenticating protector accounts using other techniques, such as biometrics. For example, once a key has been derived from biometric data, PolyPasswordHasher will function in a way that is identical to the private key authentication scheme discussed in the previous paragraph.

B. Deployment

One key issue with deploying PolyPasswordHasher is deciding which accounts should be protector accounts and which should be shielded. General guidelines include that protector accounts should be assigned to users who choose strong passwords. Ideally these are users who will log in soon after a reboot, which will minimize the amount of time needed to bootstrap. Although different servers may have different use patterns, in our experience, system administrators usually meet the above requirements. We examined logs from servers at our institution and found that system administrators typically login a few minutes after a system is restarted (likely because they will check the system to make sure it is operating correctly). Based on these practices, system administrators tend to be ideal candidates for protector accounts.

The threshold setting is another important value for deployment. In our experience, a threshold value of 2 - 5 is sufficient, even for systems that process thousands of password requests. As we demonstrate in the evaluation section, even a threshold of 2 increases password strength immensely.

Using PolyPasswordHasher results in very minor changes to existing password authentication systems. The client tools for password authentication do not change in any way. In fact, because PolyPasswordHasher only impacts password storage, it is invisible to clients. For administrators, the only change is that when an account is created, the administrator can specify whether or not an account counts toward the threshold.

The file format for password storage is similar to existing systems. In systems that use a database, the share number and isolated-check bits can simply be inserted as new columns in the table. Many servers (such as Linux, Mac, and BSD) use the /etc/shadow or /etc/master.password colon delimited formats. As it is used today, the password field contains both the salt and hash but these different portions of the field are not delineated. With PolyPasswordHasher, additional data (such as the share number and isolated-check bits) can also be added at a known position within the password field (e.g., the beginning or end); doing so will lengthen this variable length, opaque field.

VI. Evaluation

To evaluate PolyPasswordHasher, we evaluated the feasibility of its use in different scenarios. This includes issues ranging from the efficiency of the algorithm, to the expected security benefits from different configurations and deployment scenarios. We frame our evaluation around the following questions:

- How long does PolyPasswordHasher take for different operations?
- What is the storage cost of PolyPasswordHasher?
- How much memory is needed for PolyPasswordHasher?
- What are the security properties of PolyPasswordHasher?
    - What happens if users pick weak passwords?
    - What isolated-check bits settings should be used?
    - How does the threshold affect the cracking time?
    - What if a poor value is chosen for the threshold?

A. What is the time cost of PolyPasswordHasher?

To evaluate time costs, we examined how long it took PolyPasswordHasher to process account verifications, new accounts, password changes, initializations of a password store, and transitioning from bootstrapping to normal operation. We measured the processing speed and performance of PolyPasswordHasher using an early-2011 MacBook Pro with 4GB of RAM and a 2.3 GHz Intel Core i5 processor. All operations reflect the mean verification time across 100 runs and were performed with the password file already present in memory. For benchmarking purposes, each action was performed sequentially despite being embarrassingly parallelizable.

Figure 8 shows the time taken by different operations (discussed below). Unless noted, the time cost of an operation did not depend on other factors, such as the number of accounts in the password database.

a) Time to verify an account: The mean time to verify a protector account was around 60μs independent of the threshold value. For example, a threshold of eight allowed a server to verify more than 16K user accounts per second. Verifying a shielded account took approximately 29μs and the server processed about 35K such actions per second. Bootstrap accounts, which use salted hashing, took just over 3μs on the same platform.

The values presented do not include key stretching. (Except for the case of isolated validation, key stretching is unlikely to be used for individual account verification with PolyPasswordHasher.)

b) Time to create accounts or process password changes: Account creation time was similar to the time needed to verify accounts. Depending on the threshold value, the average creation time varied from 77μs to 85μs. A PolyPasswordHasher store with a threshold value of eight created more than 12K accounts per second. Given that there are a maximum of 255 protector accounts that can be created (at least with a PolyPasswordHasher implementation that uses GF256), protector account creation time is not a performance concern. Shielded accounts were created independent of the threshold, in about 25μs. Based on this, we calculated that approximately 40K thresholdless accounts can be created each second. This is similar to the time it takes to generate a salted SHA256 hash for a password (16μs). Changing a password only requires

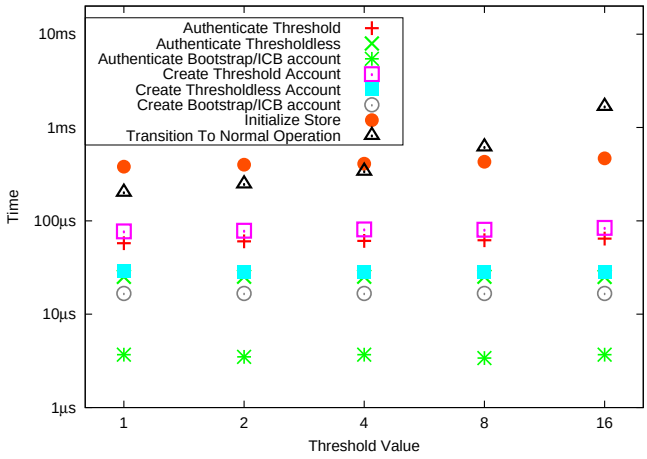

Figure 8: Time needed per operations in PolyPasswordHasher. Isolated-check bits verification and transition to normal operation times are listed without key stretching.

PolyPasswordHasher to perform the same operation it uses to create an account (and potentially also authenticate the old password).

c) Time needed to initialize a password database: The time to create a database varied and was dependent on the threshold value used. However, varying the threshold from 2 to 16, varied the creation time from 380μs to 460μs. This time cost is largely due to the need to generate cryptographically-suitable random numbers; but this operation is only performed once, when a new password file is created.

d) Time needed to transition to normal operation: When the server restarts, random coefficients are computed from the set of provided shares, with full interpolation. The time needed to complete this operation varied between 202μs and 1.6ms, because the threshold changes in relation to changes in the number of polynomials, as the store size increases. Note that small thresholds were processed quickly — a store with a threshold of 8 was unlocked in 617μs. There is also the time needed to do the integrity check on the recovered secret.

Computing the random coefficients and computing the integrity of the recovered secret (via a cryptographic hash) needs to be done once per restart for a normal server. However, an attacker attempting to crack the database will recompute the random coefficients and compute the hash once per guess of protector account values. We therefore recommend that the integrity check use enough iterations of the hash function to increase the computational time needed to at least 100ms. This slows the attacker down per guess (a frequent operation), while only delaying the server slightly one time per reboot.

Recommendation 1: The integrity check on the secret that is performed when transitioning to normal operation should take at least 100ms to complete.

B. What is the Storage Cost of PolyPasswordHasher?

Storing passwords with PolyPasswordHasher requires that additional information – a one byte share number – be stored for each account. This adds one byte of storage space for each account, beyond the cost of current hash techniques. The one byte has minimal impact on the disk space needed to store production password databases (Table II).

Password source	Original space	Salted Hashes space	PPH No IV	PPH (16 ICB)	PPH (24 ICB)
RockYou	134MB	260MB	265MB	275MB	280MB
eHarmony*	51.6MB	100MB	102MB	106MB	108MB
Formspring*	27.3MB	34.8MB	35.2MB	36.0 MB	36.4MB
Gawker	75.2MB	119MB	120MB	122MB	123MB
LinkedIn*	252MB	424MB	430MB	442MB	448MB
Sony	2.98MB	4.95MB	5.00MB	5.10MB	5.15MB
Yahoo	17.8MB	35.0MB	35.4MB	36.2MB	36.6MB

Table II: Disk space needed to store leaked password databases in different formats. * Denotes breaches in which only the salted hash portion of the database was released

If the isolated-check bits field is used, the amount of information increases by the size of the field. For example, with a value of 24 isolated-check bits, the total cost increases by four bytes (one for the share number and three for the isolated-check bits field).

C. How much memory is needed for PolyPasswordHasher?

PolyPasswordHasher uses more memory than does a salted hash solution in normal operation. The server must store the polynomial coefficients for the Shamir Secret Share (which includes the secret). However, the total size of this data is relatively small — the threshold value (2-5) multiplied by the length of the share$\oplus$hash field (32 bytes). As this value is likely to be a few hundred bytes in practical deployments, (which is smaller than the PolyPasswordHasher code will be in memory), this should not pose a problem in practice.

During bootstrapping, additional memory is also used. While shares are being acquired, they are kept in memory. This has a similar cost to storing the polynomial coefficients for the Shamir Secret Share. However, if isolated validation is used, the more substantial cost is storing authentication information so that it can be verified against the complete entry to ensure that the previous login was correct. The memory cost of caching this information will correspond to the number of logins before bootstrapping is finished (i.e., share$\oplus$hash * Log-ins). The total memory cost is still rather small, even for a server that processes many authentications. For example, a server that processes ten thousand logins while bootstrapping needs only 320KB of memory to store that information. Thus the memory costs remain small, even for heavily used systems.

D. What happens if users choose extremely weak passwords?

If extremely weak passwords are used for a threshold of protector accounts (like administrators), PolyPasswordHasher will not provide strong protection. If there are only a few bits of entropy in the password, the search space will still be small, but exponentially larger. For example, if the attacker knows that three protector accounts each chose one of 10 weak passwords, the attacker could sweep the search space in 1000 guesses (100 seconds, given a 100ms time to verify the secret).

While this is much better than the 30 guesses an attacker would need with salted hashing, this protection is obviously still very weak. It is thus critical that protector accounts are secured with strong passwords. PolyPasswordHasher performs very poorly when weak passwords are used for protector accounts. Even with extremely weak passwords, shielded accounts will have some protection, so long as the protector accounts use strong passwords.

However, independent of any storage technology, extremely weak passwords are susceptible to guessing by an external party without having to steal the password database. Therefore, this is not an area that password database storage technologies can address.

Fortunately, research has shown that users can be helped to choose passwords that have a significant degree of randomness. This can be done by requiring a combination of passwords that are a certain length (8 characters), combinations of character types (letters, numbers, symbols), and by blacklisting common passwords [5]. Extremely weak passwords are susceptible to guessing by an external party (without database access); as best practice, sites should block them from use [6], [5], [7], [8]. The combination of length, diverse character types, and blacklisting common passwords has been shown to substantially increase the time needed to crack passwords.

Recommendation 2: Protector accounts should have as much entropy as a 6-character-long random password.

E. What isolated-check bits setting should be used?

Isolated validation changes the properties of PolyPasswordHasher, depending on the size of the isolated-check bits field.

- When isolated validation is disabled, a user may not log in until a threshold of protector account passwords have been provided. This makes PolyPasswordHasher unavailable during the bootstrap phase.
- When the size of the isolated-check bits field is very small, it is possible that authentication errors will be made during bootstrapping. For example, if the size of the isolated-check bits field is set to 1 bit (an implausible and extreme example), then an attacker typing a random password has a 50% chance of the isolated-check bits matching and being allowed in.
- As the size of the isolated-check bits field grows, the probability that an attacker will succeed with online cracking decreases. However, when the size of the isolated-check bits field is very large, if an attacker steals the database, the confidentiality benefits of PolyPasswordHasher are weakened. For example, in the extreme case that the isolated-check bits field is the size of the entire salted password hash (an implausible example), then an attacker can use the slow, isolated validation hash to individually crack passwords. In this case, from a security standpoint, PolyPasswordHasher is effectively equivalent to key stretching. The primary benefit of PolyPasswordHasher in this case is that it does authenticate accounts much more quickly in normal operation than does key stretching.

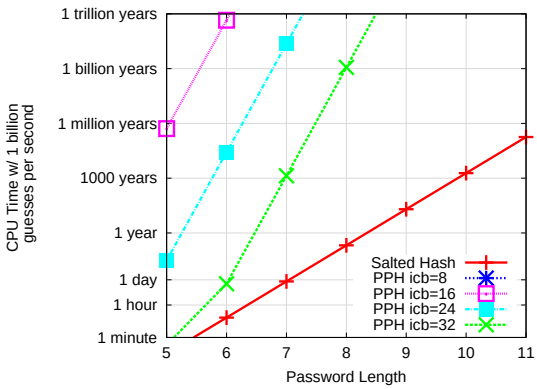

Figure 9: Time it takes to crack a PolyPasswordHasher store with different values for isolated-check bits and a threshold of 3. The line for isolated-check bits =8 does not fit within the axes of the graph.

Given that the size of the isolated-check bits field and the threshold is configurable, the security properties of PolyPasswordHasher vary. When integrating PolyPasswordHasher into a system, an administrator can leverage this adaptability to better meet his expectations. In the next subsection we explain how different configurations of PolyPasswordHasher apply to different use cases.

1) Immediate Availability For Many Unknown Clients: In some situations, after reboot a server needs to be immediately available to a potentially huge number of untrusted clients. For example, a web forum or a social media site is likely to have this property. Availability is paramount, for a large, diverse, and constantly changing user base.

If an attacker has his attack detected and his IP address blocked, this may not be a major concern or deterrent. Therefore, attackers may try to do things such as brute force passwords, even if this is likely to lead to detection.

There is also a risk of users trying incorrect passwords during the bootstrap phase to try to break into an account. Most existing web frameworks deal with this by restricting the number of password attempts from an IP address over a period of time (10 authentications per second is common). However, a motivated attacker with the ability to connect from many IP addresses (e.g., a botnet operator), could still be able try a substantial number of passwords. However, using a isolated-check bits value of 24 bits will cause the attacker's password search space to be as large as the usual number of attempts for successful online cracking scenarios.

The downside is that a large isolated-check bits field will make it easier for an attacker to crack the protector accounts. As shown in Figure 10, as the size of the isolated-check bits field grows, the cracking time for a threshold of three starts to resemble the time for salted hashes. For 24 isolated-check bits, the time to crack a six-character equivalent random password is around 270420 years. In summary, the isolated-check bits setting reduces the password entropy. To provide adequate protection, either passwords with the entropy of 8 character random passwords should be used, or, as will be discussed later, the threshold should be increased.

Recommendation 3: In situations where a large number

guessing probability	Attempts		
	ICB=8	ICB=16	ICB=24
25%	76 (25.4%)	18,767 (24.9%)	4,804,150 (24.9%)
50%	177 (49.9%)	45,295 (49.9%)	11,595,559 (49.9%)
75%	354 (74.9%)	90,590 (74.9%)	23,191,185 (74.9%)

Table III: Number of attempts required to find a isolated-check bits collision

of unknown clients will authenticate, use a setting of 24 isolated-check bits.

2) Immediate Availability For Known Clients: In many cases, the set of expected clients is limited and known. This includes situations like a institutional file server or mail server. In this case, the total set of possible clients is limited and it is assumed that an attacker cannot control a large number of IP addresses / clients.

In this case, an attacker who brute forces accounts is much less of a concern because he cannot obtain the IP addresses needed and failed attempts should be flagged and investigated by the security team regardless. Thus, setting isolated-check bits to be 16 requires that an attacker attempt to crack 45295 passwords to have a probability of finding a collision of about 50%, as shown in Table III. This configuration is ideal for cases in which a lock-out policy can be enforced. However, even if an attacker was able to find a collision during bootstrapping, PolyPasswordHasher would be able to detect it upon recombination. If an attacker were to find a collision when PolyPasswordHasher is performing on normal operation, then the break-in attempt would also notify the administrators.

The bigger concern is an attacker (possibly an insider) stealing the database and brute forcing account passwords. By using a small number of isolated-check bits, such as 16, the resilience to cracking is very high. Even if there are three protector accounts with the entropy of a 6 character password, it would require $447.8 * 10^6$ thousand years of CPU effort to crack these accounts.

Recommendation 4: In situations where a restricted set of clients will try to authenticate, use a setting of 16 isolated-check bits.

3) When Short Periods of Unavailability are Acceptable: In some situations, temporary unavailability of a server (while bootstrapping) is not a major concern. For example, many services load balance requests across multiple systems for redundancy and performance. Administrators will stop and start instances as needed. When starting an instance, the first thing the administrators will typically do is log into the system to validate that it is working, before moving it into production. For such a system, the time spent in the bootstrapping phase is small and there is no user-perceived unavailability while the system is bootstrapping.

In this case, isolated validation can be completely disabled, maximizing the security benefit of PolyPasswordHasher. The CPU time it would take an attacker to crack the database is infeasibly high for practical situations. Even with a threshold of 3 and 5 character random accounts, the cracking time when isolated validation is disabled ($1.46 * 10^{21}$ CPU years) extends past the top of the y-axis in Figure 9. (This is more CPU time than would be provided by every computer working nonstop for the estimated age of the universe.) Thus, given our understanding of the security of SHA256 and AES, it is infeasible for any adversary to use the password database to crack passwords with the entropy of 5 character random passwords that are stored in this manner.

Equivalent Entropy	ICB	Threshold	Keyspace	Recombinations
32.45 (5 character-long, random password)	Disabled	3	$4.632 * 10^{29}$	$5.559 * 10^{31}$
		4	$3.584 * 10^{39}$	$7.528 * 10^{41}$
	16	3	$1.645 * 10^{13}$	$1.975 * 10^{17}$
		4	$1.943 * 10^{18}$	$4.080 * 10^{22}$
	24	3	$2.331 * 10^{10}$	$1.175 * 10^{10}$
		4	$7.619 * 10^{10}$	$9.484 * 10^{12}$
38.85 (6 charater-long, random password)	Disabled	3	$4.972 * 10^{35}$	$6.554 * 10^{37}$
		4	$2.919 * 10^{47}$	$9.635 * 10^{49}$
	16	3	$1.411 * 10^{21}$	$1.693 * 10^{23}$
		4	$1.582 * 10^{28}$	$3.24 * 10^{30}$
	24	3	$8.631 * 10^{13}$	$1.009 * 10^{16}$
		4	$3.684 * 10^{18}$	$7.738 * 10^{20}$
45.21 (7 character-long, random password)	Disabled	3	$3.405 * 10^{41}$	$5.619 * 10^{43}$
		4	$2.378 * 10^{55}$	$7.848 * 10^{57}$
	16	3	$1.209 * 10^{27}$	$1.558 * 10^{28}$
		4	$1.289 * 10^{36}$	$2.976 * 10^{50}$
	24	3	$7.211 * 10^{19}$	$8.654 * 10^{21}$
		4	$3.685 * 10^{26}$	$6.303 * 10^{28}$

Table IV: Equivalent keyspaces and number of recombinations for different PolyPasswordHasher configurations. Assuming 10 protector accounts

Recommendation 5: In situations where availability immediately after a reboot is not essential, isolated validation should be disabled.

F. How does the threshold affect the cracking time?

The threshold value is directly related to the cracking time by increasing the number of secret recombinations that an attacker needs to attempt before he can recover the secret. Also, on the usability side, it is also related to the time it takes for a system to transition from the bootstrap phase to normal operation.

We can see in Figure 10 that, when we fix the isolated-check bits number to 24, the threshold value exponentially increases the cracking time. (We choose a high number of isolated-check bits because the security is lower and thus, differences are easier to see on the graph.) If the protector accounts pick strong enough passwords, and the threshold is three, the cracking time increases by many orders of magnitude. The regular salted hashes are expected to be cracked in about 40 minutes, whereas, if using PolyPasswordHasher, these hashes would be cracked in $27,042$ years.

The reason why PolyPasswordHasher provides strong protection is that even when an attacker has to guess only a few passwords, in most cases PolyPasswordHasher's exponential increase in guessing time ($O(Vp)$ instead of $O(pV)$, will make

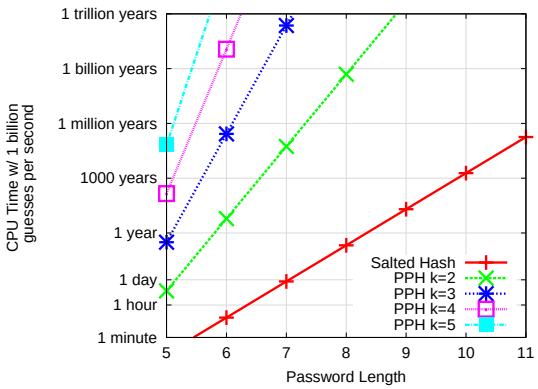

Figure 10: Time it takes to crack a PolyPasswordHasher store with different values of threshold and 24 isolated-check bits

guessing a password computationally infeasible (Figure 10). Furthermore, it is possible to much more aggressively apply a technique like key stretching for the integrity check of the reconstructed secret. This is because this operation is only done once (by the server), but is done for each guessed combination of passwords (by the attacker). This increases the attacker's time cost while extending the time the server spends, by a second or less, in bootstrapping .

Recommendation 6: Choose a threshold of 3 plus the number of protector account passwords an attacker could reasonably know, for strong protection.

G. What if a poor value is chosen for the threshold?

Choosing a poor threshold setting can minimize the protections of PolyPasswordHasher. If the threshold is too high, the system will primarily be in the bootstrapping phase while waiting for protector accounts to log in. Thus most of the accounts will be bootstrap accounts and only be protected by salted hashes. If an attacker steals this database, they can individually crack these passwords.

An organization might find that there are not enough protector account logins to get the system into normal operation. One way to mitigate this is to promote some shielded accounts that log in frequently to be protector accounts. This would allow an attacker who controls one or more of those accounts to have an easier time cracking the protector accounts, but causes bootstrapping accounts to be protected.

Recommendation 7: It is often better to have protector accounts that frequently log in (but who are untrusted), than those of trusted users who rarely log in.

If the threshold setting is too low, then an attacker can feasibly crack the protector accounts and gain the ability to individually crack accounts. This is particularly problematic when combined with a large isolated-check bits settion. The minimum threshold setting described in the previous settings should be used wherever possible. This relationship can be verified by consulting Table IV

VII. Related Work

Given the ubiquity of passwords, password security has become an important research area in both academia and industry. Hackers have also found password databases enticing and have devoted time and resources to figuring out new strategies for gaining access to passwords and associated user identities. Hackers persistence in developing ever more sophisticated cracking strategies has in turn focused researchers, who have come up with many promising solutions that each solve different portions of the problem.

In this section, we categorize related work according to their requirements for deployment. Solutions that use additional server hardware or servers require effort to deploy but tend to present very strong protection. Changes that involve the client also require effort to deploy (on clients), but in many cases are also very effective at protecting user information. There are relatively few solutions that only require changes on the server side and are software only. However, these solutions, like salted hashing and key stretching, tend to be very widely deployed. PolyPasswordHasher is the first solution of this type that increases attackers' cracking effort asymmetrically more than the increase to the server's verification time.

A. Systems that Require Extra Servers or Server-side Hardware

Multi-server Password Authentication. A wide variety of authentication schemes use multiple servers to store password data [9], [10], [11]. The assumption is that an attacker cannot compromise a threshold of the servers. In contrast, PolyPasswordHasher uses a threshold system to hide information in memory on a single server that can only be unlocked with a threshold of correct user passwords.

Prior work by Gwoboa [12] hides passwords using a trapdoor function (public key cryptography) and techniques from threshold cryptography. It can authenticate users with two hidden pieces of information, a user ID (likely not the user name, for security reasons) and the password. However, a major concern of the scheme is how the private key is stored on the server. The authors propose splitting it amongst multiple systems and using threshold cryptography.

Decoys. Recently, researchers have suggested an array of techniques that employ sets of extra password entries [13], [14]. For example, the Honeywords [14]system uses a separate server to hold information that authenticates the correct password entry. Once attackers obtain a password database they do not know which password entry is correct. Entering a password that matches a hash to the wrong password entry will trigger an alarm that notifies the administrator that there has been a password hash file breach. However, for this to work, there must be a separate, secure server to authenticate the index of that entry (a one byte value). An important attribute of HoneyWords is that it can also operate when the server is offline and will check passwords and detect breaches when the server comes back online. This is similar to the way that accounts validated during bootstrapping in PolyPasswordHasher are revalidated after a threshold of protector account passwords are provided.

Using a hardware database encryption key. Storing a key in trusted hardware (such as a USB dongle [15]) has security benefits, but also has deployment challenges. If the hardware stops working, the protected data is likely to be irrevocably lost. To have backup servers, it is essential to have identical

duplicate copies of the secure token. This complicates use in scenarios like cloud computing or even just a standard master / slave deployment.

Bounded Attacker. Di Crescenzo [16] proposed a scheme for protecting password data when an attacker can only read a bounded amount of data from storage. This entails that an organization configure network monitoring hardware and set up a separate server to process authentication requests. This may pose a deployment challenge for some organizations.

Single Sign-On. Single sign-on systems such as OpenID and OAuth have promise for organizations to securely offload authentication to a third party. This is convenient for users, but is far from ubiquitous for a variety of reasons [17]. Single sign-on systems have known security issues [18], [19], but overall can be effective when properly used. PolyPasswordHasher integrates cleanly with such systems and can trivially allow shielded accounts to verify with them. In addition, PolyPasswordHasher could be used by a Single Sign-On provider to securely store its password database.

B. Systems That Require Client Side Changes

Biometrics. Biometric authentication is a promising field for secure password authentication [20], [21], [22], [23], [24], [25], [26], [27], [28], with a substantial amount of work already done on how to store and authenticate users with biometric information. Like PolyPasswordHasher, some of these approaches use a threshold system to validate and authenticate users, in part to deal with noisy biometric data [29], [30]. Although PolyPasswordHasher also requires users to remember a password, it does not require client-side hardware. Moreover, prior work uses keystroke dynamics to change stored password data [31] and relies on reading timing information as users type their passwords during login. This method of analyzing timing of keystrokes provides promising protections, but requires changes to the client and server to correctly operate.

Authentication Using Tokens or Smart Cards. Much investigation has looked at authentication systems. These systems, which use hardware tokens, are popular in banking [32], [33], health services [34], as well as in more general contexts [35], [36]. These systems are extremely effective and are widely used to limit and protect access to classified systems, often by integrating a threshold cryptosystem to provide multifactor authentication using hardware tokens [37], [38]. There are even client devices that one can use either as a primary authentication source or as a second factor of authentication.

Password Managers. There are a myriad of password managers that help users choose and manage secure per-site passwords including LastPass, 1Password, and OnePass. These systems store password data and lock it with the user's credentials. However, these third party software (and often their servers) will know users' passwords, which opens the door to potential misuse. To mitigate this, several groups have proposed cryptographic techniques that will similarly take a user's password and generate secure, per site passwords [39], [40], [41]. These techniques are effective (and more secure) but can create passwords that are incompatible with a server's password policy. Password manager tools require client-side changes or support, whereas PolyPasswordHasher requires server-side changes only. Both can be used in conjunction safely for improved security.

Multiparty Computational Authentication. There are a variety of schemes that perform secure, remote authentication using computation by the client and server on legacy hardware [42], [43], [44], [45], [46], [47], [48], [49], [50], [51]. These schemes have significant, positive aspects such as (in some cases) requiring an attacker to be online to validate communications. However, they require multiparty protocols that require changes on clients and servers.

Non-password Authentication. Many researchers have proposed authentication based upon non-password items such as pictures [52]. In practice, these systems can face security limitations if users do not appropriately choose their authentication tokens [53]. Used with these authentication mechanisms, PolyPasswordHasher functions well for shielded accounts, requiring no changes to the system. However, unless a key or hash can be derived from such mechanisms, it may not be possible to use them to recover a protector account's share.

Key Exchange Schemes. There are also many systems that implement secure key exchange [54], such as Password Authenticated Key Exchange (PAKE) [55], [56], [57], [58], Encrypted Key Exchange (EKE) [59], [60], [61], and further enhancements [62]. These systems allow parties who share a password to securely find an encryption key to hide their communications. These systems provide excellent protection and can handle compromises to a system's memory, in many cases. However, unlike PolyPasswordHasher, they typically involve multi-round authentications and require changes to both the client and server.

Helping Users Choose Stronger Passwords. There have been many efforts to help users choose stronger, more memorable passwords and understand what constitutes a weak password [63], [64], [65], [66], [67], [68], [69]. Such efforts could be quite effective in protecting users, but users will need to adopt new password habits and strategies. We leverage studies by [5], [70], [7], [8] to understand how to add policies that improve entropy in protector account passwords.

Two-Factor Authentication. The use of two-factor authentication [71] is provided by some popular services (typically through an SMS or phone call). Two-factor authentication does not change PolyPasswordHasher's use in any way. Users can easily get the best of both protections with simultaneous deployment of each technology.

C. Server-Side, Software-Only

These mechanisms fall directly into the same threat model as PolyPasswordHasher.

Key Stretching. One way to mitigate the effectiveness of cracking salted hashes is to use key stretching techniques [72]. This involves performing multiple rounds of cryptographic operations to validate a hash (or key). But key stretching incurs a slowdown for the attacker, proportional to the number of iterations. Unfortunately, it also slows the server down by the same factor, which limits the effectiveness of the technique.

PolyPasswordHasher leverages key stretching in two parts of the algorithm in a way that poses a much higher cost to

the attacker than the server. PolyPasswordHasher may also use key stretching during isolated validation. This increases the authentication time only when bootstrapping, yet increases the time cost to the attacker for all cracking efforts. When the server is in normal operation, there is no slowdown. This presents an increase that is the same factor for the server and attacker, but the server only needs to pay the cost for a short period.

The more impactful use of key stretching is when PolyPasswordHasher performs an integrity check on the secret before transitioning to normal operation. This operation is done only once per reboot by the server, yet the attacker must perform this operation once per guess of protector account passwords. This causes an enormous, asymmetric increase in cracking time for the attacker.

VIII. Conclusion

This work presents PolyPasswordHasher, a technique for protecting user passwords in the event of password database disclosure. PolyPasswordHasher is the first technique that requires only software changes on the server and yet requires attackers to do asymmetrically more work to crack passwords than servers need to do to verify them. As a result, password cracking becomes infeasible for attackers in many cases.

PolyPasswordHasher is practical to deploy and is effective in practice. The performance of PolyPasswordHasher is similar to salted hashing. The memory and storage costs from using PolyPasswordHasher are negligible. So long as good password selection procedures are followed, protector accounts increase an attacker's cracking effort by many orders of magnitude. We show how configuring isolated validation in different ways can help to optimize PolyPasswordHasher for different configurations of services.

We have installed PolyPasswordHasher at our institution and are gaining practical experience to uncover any usability problems. Five different parties, two of whom had no prior contact with us, authored an implementation of PolyPasswordHasher. There are implementations for a variety of languages and web frameworks available with an MIT license at: https://polypasswordhasher.poly.edu.

References

[1] D. Mirante and J. Cappos, "Understanding Password Database Compromises," NYU Poly, Tech. Rep. TR–CSE–2013–02, Sep 2013.

[2] T. Hunt, "Our password hashing has no clothes," http://www.troyhunt.com/2012/06/our-password-hashing-has-no-clothes.html.

[3] B. K. Marshall, "PasswordResearch.com Authentication News: Passwords Found in the Wild for January 2013," http://blog.passwordresearch.com/2013/02/passwords-found-in-wild-for-january-2013.html.

[4] A. Shamir, "How to share a secret," *Communications of the ACM*, vol. 22, no. 11, pp. 612–613, 1979.

[5] 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*, ser. CCS '10. New York, NY, USA: ACM, 2010, pp. 162–175. [Online]. Available: http://doi.acm.org/10.1145/1866307.1866327

[6] "Hotmail banning common passwords to beef up security — Ars Technica," http://arstechnica.com/information-technology/2011/07/hotmail-banning-common-passwords-to-beef-up-security/.

[7] S. Komanduri, R. Shay, L. F. Cranor, C. Herley, and S. Schechter, "Telepathwords: Preventing weak passwords by reading users' minds," in *23rd USENIX Security Symposium (USENIX Security 14)*. San Diego, CA: USENIX Association, Aug. 2014, pp. 591–606. [Online]. Available: https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/komanduri

[8] P. Kelley, S. Komanduri, M. Mazurek, R. Shay, T. Vidas, L. Bauer, N. Christin, L. Cranor, and J. Lopez, "Guess again (and again and again): Measuring password strength by simulating password-cracking algorithms," in *Security and Privacy (SP), 2012 IEEE Symposium on*, May 2012, pp. 523–537.

[9] Z. Chai, Z. Cao, and R. Lu, "Threshold password authentication against guessing attacks in ad hoc networks," *Ad Hoc Networks*, vol. 5, no. 7, pp. 1046 – 1054, 2007. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1570870506000308

[10] A. Bagherzandi, S. Jarecki, N. Saxena, and Y. Lu, "Password-protected secret sharing," in *Proceedings of the 18th ACM conference on Computer and communications security*. ACM, 2011, pp. 433–444.

[11] J. Katz, P. MacKenzie, G. Taban, and V. Gligor, "Two-server password-only authenticated key exchange," in *Applied Cryptography and Network Security*. Springer, 2005, pp. 1–16.

[12] H. Gwoboa, "Password authentication without using a password table," *Information Processing Letters*, vol. 55, no. 5, pp. 247–250, 1995.

[13] 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*, ser. CCS '13. New York, NY, USA: ACM, 2013, pp. 187–198. [Online]. Available: http://doi.acm.org/10.1145/2508859.2516746

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

[15] "University of Cambridge Develops Potentially More Secure Password Storage System," http://it.slashdot.org/story/14/03/11/002206/university-of-cambridge-develops-potentially-more-secure-password-storage-system.

[16] G. Di Crescenzo, R. Lipton, and S. Walfish, "Perfectly secure password protocols in the bounded retrieval model," *Theory of Cryptography*, pp. 225–244, 2006.

[17] S.-T. Sun, Y. Boshmaf, K. Hawkey, and K. Beznosov, "A billion keys, but few locks: the crisis of web single sign-on," in *Proceedings of the 2010 workshop on New security paradigms*. ACM, 2010, pp. 61–72.

[18] "Security Issues – OpenID Review," https://sites.google.com/site/openidreview/issues.

[19] "Security Advisories – OAuth," http://oauth.net/advisories/.

[20] M. J. Atallah, K. B. Frikken, M. T. Goodrich, and R. Tamassia, "Secure biometric authentication for weak computational devices," in *Financial Cryptography and Data Security*. Springer, 2005, pp. 357–371.

[21] R. Snelick, U. Uludag, A. Mink, M. Indovina, and A. Jain, "Large-scale evaluation of multimodal biometric authentication using state-of-the-art systems," *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, vol. 27, no. 3, pp. 450–455, 2005.

[22] P. Tuyls and J. Goseling, "Capacity and examples of template-protecting biometric authentication systems," in *Biometric Authentication*. Springer, 2004, pp. 158–170.

[23] X. Boyen, Y. Dodis, J. Katz, R. Ostrovsky, and A. Smith, "Secure remote authentication using biometric data," in *Advances in Cryptology–EUROCRYPT 2005*. Springer, 2005, pp. 147–163.

[24] Z. Erkin, M. Franz, J. Guajardo, S. Katzenbeisser, I. Lagendijk, and T. Toft, "Privacy-preserving face recognition," in *Privacy Enhancing Technologies*. Springer, 2009, pp. 235–253.

[25] F. Kerschbaum, M. J. Atallah, D. MRaïhi, and J. R. Rice, "Private fingerprint verification without local storage," in *Biometric Authentication*. Springer, 2004, pp. 387–394.

[26] M. Osadchy, B. Pinkas, A. Jarrous, and B. Moskovich, "Scifi-a system for secure face identification," in *Security and Privacy (SP), 2010 IEEE Symposium on*. IEEE, 2010, pp. 239–254.

[27] F. Monrose, M. K. Reiter, Q. Li, and S. Wetzel, "Cryptographic key generation from voice," in *Security and Privacy, 2001. S&P 2001. Proceedings. 2001 IEEE Symposium on*. IEEE, 2001, pp. 202–213.

[28] N. Sae-Bae, K. Ahmed, K. Isbister, and N. Memon, "Biometric-rich gestures: a novel approach to authentication on multi-touch devices," in *Proceedings of the 2012 ACM annual conference on human factors in computing systems*. ACM, 2012, pp. 977–986.

[29] A. Juels and M. Sudan, "A fuzzy vault scheme," *Designs, Codes and Cryptography*, vol. 38, no. 2, pp. 237–257, 2006.

[30] L. Ballard, S. Kamara, and M. K. Reiter, "The practical subtleties of biometric key generation," in *Proceedings of 17th Conference on Security Symposium*, 2008, pp. 61–74.

[31] F. Monrose and A. D. Rubin, "Keystroke dynamics as a biometric for authentication," *Future Generation Computer Systems*, vol. 16, no. 4, pp. 351–359, 2000.

[32] V. Deo, R. B. Seidensticker, and D. R. Simon, "Authentication system and method for smart card transactions," Feb. 24 1998, uS Patent 5,721,781.

[33] K.-H. Yeh, C. Su, N.-W. Lo, Y. Li, and Y.-X. Hung, "Two robust remote user authentication protocols using smart cards," *Journal of Systems and Software*, vol. 83, no. 12, pp. 2556–2565, 2010.

[34] G.-J. Ahn and D. Shin, "Towards scalable authentication in health services," in *Enabling Technologies: Infrastructure for Collaborative Enterprises, 2002. WET ICE 2002. Proceedings. Eleventh IEEE International Workshops on*. IEEE, 2002, pp. 83–88.

[35] H.-Y. Chien, J.-K. Jan, and Y.-M. Tseng, "An efficient and practical solution to remote authentication: smart card," *Computers & Security*, vol. 21, no. 4, pp. 372–375, 2002.

[36] W.-H. Yang and S.-P. Shieh, "Password authentication schemes with smart cards," *Computers & Security*, vol. 18, no. 8, pp. 727–733, 1999.

[37] *UbiComp '14 Adjunct: Proceedings of the 2014 ACM International Joint Conference on Pervasive and Ubiquitous Computing: Adjunct Publication*. New York, NY, USA: ACM, 2014.

[38] Yubico, "Yubikey hardware," https://www.yubico.com/products/yubikey-hardware/.

[39] J. A. Halderman, S. D. Schoen, N. Heninger, W. Clarkson, W. Paul, J. A. Calandrino, A. J. Feldman, J. Appelbaum, and E. W. Felten, "Lest we remember: cold-boot attacks on encryption keys," *Communications of the ACM*, vol. 52, no. 5, pp. 91–98, 2009.

[40] B. Ross, C. Jackson, N. Miyake, D. Boneh, and J. C. Mitchell, "Stronger password authentication using browser extensions," in *Proceedings of the 14th Usenix Security Symposium*, vol. 1998, 2005.

[41] J. A. Halderman, B. Waters, and E. W. Felten, "A convenient method for securely managing passwords," in *Proceedings of the 14th international conference on World Wide Web*. ACM, 2005, pp. 471–479.

[42] T. Wu *et al.*, "The secure remote password protocol," in *Internet Society Symposium on Network and Distributed System Security*, 1998.

[43] T. Lomas, L. Gong, J. Saltzer, and R. Needhamn, "Reducing risks from poorly chosen keys," in *Proceedings of the twelfth ACM symposium on Operating systems principles*, ser. SOSP '89. New York, NY, USA: ACM, 1989, pp. 14–18. [Online]. Available: http://doi.acm.org/10.1145/74850.74853

[44] H.-Y. Chien, J.-K. Jan, and Y.-M. Tseng, "A modified remote login authentication scheme based on geometric approach," *Journal of Systems and Software*, vol. 55, no. 3, pp. 287–290, 2001.

[45] J.-K. Jan and Y.-Y. Chen, "paramita wisdom password authentication scheme without verification tables," *Journal of Systems and Software*, vol. 42, no. 1, pp. 45–57, 1998.

[46] L. Gong, "Optimal authentication protocols resistant to password guessing attacks," in *Computer Security Foundations Workshop, 1995. Proceedings., Eighth IEEE*. IEEE, 1995, pp. 24–29.

[47] J. Camenisch, N. Casati, T. Gross, and V. Shoup, "Credential authenticated identification and key exchange," in *Advances in Cryptology–CRYPTO 2010*. Springer, 2010, pp. 255–276.

[48] J. Brainard, A. Juels, B. Kaliski, and M. Szydlo, "Nightingale: A new two-server approach for authentication with short secrets," in *12th USENIX Security Symp*. Citeseer, 2003.

[49] J. Katz, R. Ostrovsky, and M. Yung, "Efficient password-authenticated key exchange using human-memorable passwords," in *Advances in CryptologyEUROCRYPT 2001*. Springer, 2001, pp. 475–494.

[50] ——, "Forward secrecy in password-only key exchange protocols," in *Security in Communication Networks*. Springer, 2003, pp. 29–44.

[51] L. Gong, M. A. Lomas, R. M. Needham, and J. H. Saltzer, "Protecting poorly chosen secrets from guessing attacks," *Selected Areas in Communications, IEEE Journal on*, vol. 11, no. 5, pp. 648–656, 1993.

[52] R. Dhamija and A. Perrig, "Deja vu: A user study using images for authentication," in *Proceedings of the 9th USENIX Security Symposium*. Usenix Denver, CO, 2000, pp. 45–58.

[53] D. Davis, F. Monrose, and M. K. Reiter, "On user choice in graphical password schemes," in *13th USENIX Security Symposium*, vol. 9, 2004.

[54] V. Shoup, *On formal models for secure key exchange*. Citeseer, 1999.

[55] V. Boyko, P. MacKenzie, and S. Patel, "Provably secure password-authenticated key exchange using diffie-hellman," in *Advances in Cryptology-Eurocrypt 2000*. Springer, 2000, pp. 156–171.

[56] F. Shen, C. Mei, H. Jiang, and Z. Xu, "Towards secure and reliable data storage with multi-coefficient secret sharing," in *Computer and Information Technology (CIT), 2010 IEEE 10th International Conference on*. IEEE, 2010, pp. 797–802.

[57] M. M. Sathik and A. K. Selvi, "Secret sharing scheme for data encryption based on polynomial coefficient," in *Computing Communication and Networking Technologies (ICCCNT), 2010 International Conference on*. IEEE, 2010, pp. 1–5.

[58] D. P. Jablon, "Strong password-only authenticated key exchange," *ACM SIGCOMM Computer Communication Review*, vol. 26, no. 5, pp. 5–26, 1996.

[59] M. Steiner, G. Tsudik, and M. Waidner, "Refinement and extension of encrypted key exchange," *ACM SIGOPS Operating Systems Review*, vol. 29, no. 3, pp. 22–30, 1995.

[60] S. Lucks, "Open key exchange: How to defeat dictionary attacks without encrypting public keys," in *Security Protocols*. Springer, 1998, pp. 79–90.

[61] D. P. Jablon, "Extended password key exchange protocols immune to dictionary attack," in *Enabling Technologies: Infrastructure for Collaborative Enterprises, 1997., Proceedings Sixth IEEE workshops on*. IEEE, 1997, pp. 248–255.

[62] P. Wang, Y. Kim, V. Kher, and T. Kwon, "Strengthening password-based authentication protocols against online dictionary attacks," in *Applied Cryptography and Network Security*. Springer, 2005, pp. 17–32.

[63] U. Topkara, M. J. Atallah, and M. Topkara, "Passwords decay, words endure: Secure and re-usable multiple password mnemonics," in *Proceedings of the 2007 ACM symposium on Applied computing*. ACM, 2007, pp. 292–299.

[64] D. V. Klein, "Foiling the cracker: A survey of, and improvements to, password security," in *Proceedings of the 2nd USENIX Security Workshop*, 1990, pp. 5–14.

[65] M. Bishop and D. V Klein, "Improving system security via proactive password checking," *Computers & Security*, vol. 14, no. 3, pp. 233–249, 1995.

[66] S. Schechter, C. Herley, and M. Mitzenmacher, "Popularity is everything: a new approach to protecting passwords from statistical-guessing attacks," in *Proceedings of the 5th USENIX conference on Hot topics in security*. USENIX Association, 2010, pp. 1–8.

[67] S. Komanduri, R. Shay, P. G. Kelley, M. L. Mazurek, L. Bauer, N. Christin, L. F. Cranor, and S. Egelman, "Of passwords and people: measuring the effect of password-composition policies," in *Proceedings of the 2011 annual conference on Human factors in computing systems*. ACM, 2011, pp. 2595–2604.

[68] R. Shay, S. Komanduri, P. G. Kelley, P. G. Leon, M. L. Mazurek, L. Bauer, N. Christin, and L. F. Cranor, "Encountering stronger password requirements: user attitudes and behaviors," in *Proceedings of the Sixth Symposium on Usable Privacy and Security*. ACM, 2010, p. 2.

[69] "xkcd: Password Strength," http://xkcd.com/936/.

[70] D. Florencio and C. Herley, "A large-scale study of web password habits," in *Proceedings of the 16th international conference on the World Wide Web*. Association for Computing Machinery, Inc., May 2007, pp. 657–666. [Online]. Available: http://research.microsoft.com/apps/pubs/default.aspx?id=74164

[71] R. Di Pietro, G. Me, and M. A. Strangio, "A two-factor mobile authentication scheme for secure financial transactions," in *Mobile Business, 2005. ICMB 2005. International Conference on*. IEEE, 2005, pp. 28–34.

[72] J. Kelsey, B. Schneier, C. Hall, and D. Wagner, "Secure applications of low-entropy keys," in *Information Security*. Springer, 1998, pp. 121–134.