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

Justin Cappos
New York University
jcappos@nyu.edu

Password file disclosures are a frequent problem for many companies, which makes their users the target of identify theft and similar attacks. This work provides a new general cryptographic technique to prevent an attacker from efficiently cracking individual passwords from a stolen password database. PolyPasswordHasher employs a threshold cryptosystem to protect password hashes so that they cannot be verified unless a threshold of them are known. (This is conceptually similar to encrypting the passwords with a key that is only recoverable when a threshold of passwords are known.) Even if the password file and all other data on disk is obtained by a malicious party, the attacker cannot crack any individual password without simultaneously guessing a large number of them correctly. PolyPasswordHasher is the first single server, software-only technique that increases the attacker's search space exponentially. The result is that even cracking small numbers of weak passwords is infeasible for an attacker.

PolyPasswordHasher achieves these properties with similar efficiency, storage, and memory requirements to existing salted hash schemes, performing tens of thousands of account authentications per second. When using the current best practice (of salting and hashing), cracking three passwords that are comprised of 6 random characters on a modern laptop would take under a hour. However, when protected with PolyPasswordHasher, cracking these passwords when using every computer in existence would take longer than the estimated age of the universe.

1 Introduction

Password file disclosure is a major security problem that has impacted dozens of organizations including Hotmail, LastFM, Formspring, ScribD, the New York Times, NVidia, Evernote, Billabong, Gawker, LinkedIn, Linode, ABC, Yahoo!, eHarmony, LivingSocial, and Twitter [48]. Security best practices advocate that the passwords should not be stored in plain text. Instead a user's password should be subjected to a salted hash and then stored. The (secure) hash acts as a one-way function to ensure that an attacker cannot trivially read the passwords. The salt is a random value that complicates the use of lookup tables to immediately crack weak passwords. Once password data is compromised, attackers have proven adept at quickly cracking large numbers of passwords. For example, in 2012 an attacker compromised 6.5 million LinkedIn accounts and within a week at least 60% of those passwords were cracked [43].

In situations where an attacker has root access to a running system, the attacker can intercept and read passwords from memory before any protections like salting and hashing apply, nullifying storage protections. However, while attackers have proven adept at stealing password databases, the most disclosed attack vectors, such as SQL injection [46, 48], do not require root access to a running authentication system. Thus our focus in this work is on protecting against an attacker that can read all data from persistent storage.

One possible way to protect a password hash database is to encrypt it using a key. However, when the system restarts, the database must be decrypted and if the key is stored on disk, the attacker can compromise it. At a high level, the idea behind this work is to store protection information such as the key in a way that it is recoverable only with a threshold of passwords. Thus the key does not reside on disk, but instead lies within the minds of the users (unbeknownst to them). By leveraging a threshold system, users organically provide the threshold of correct passwords (typically 2-4) needed when logging in.

To accomplish this, we devise a general cryptographic approach to validate the integrity of information based upon a novel technique called PolyHashing[1]. Previous schemes have used threshold cryptography to hide password data across multiple servers. PolyHashing uses threshold cryptography to obscure different stored

[1]The prefix 'Poly' in this context is used as a synonym for 'multi'. So the name PolyHashing refers to the need for multiple hashes.

hashes on a single server so that hashes can only be checked if a threshold are known. Thus, if an attacker does not know a threshold of correct hashes for the system it is not possible to validate any hash value. In essence, a PolyHashing store provides confidentially of all stored data until a certain amount of valid data is known. Once this threshold of data is known, a PolyHashing store then provides integrity checking for all future data.

Using PolyHashing, we built a system called PolyPasswordHasher that protects password hashes. PolyPasswordHasher ensures that a party cannot read or check any individual password in the password list, without knowing a configurable threshold number of passwords (often two to four). Thus when a system using PolyPasswordHasher restarts, a threshold of users must provide correct passwords before any may be authenticated. (We discuss an extension called *partial verification* that allows verification to occur immediately upon reboot while handling erroenously or maliciously incorrect passwors in Section 5.2.3.) Note that an extension also exists to cause untrusted users (such as Facebook or Gmail) to have their password not count toward the threshold for validation purposes.

Once the threshold is reached, it is possible to quickly and efficiently check all future password requests. In the situation where an attacker does learn a threshold of passwords, the remaining passwords are protected in as strong of a manner as existing salted hashing techniques. For an attacker that does not know a threshold of passwords, this makes password cracking infeasible.

PolyPasswordHasher only requires changes on one party (the server) and can be integrated smoothly into existing systems and verification processes. When adding PolyPasswordHasher to an existing system, no changes are visible to the user. In fact, existing client password login mechanisms do not need to be modified in any way. The only changes occur in the way that the encoded password data is generated, stored, and verified by the server.

PolyPasswordHasher relies on simple primitives that are efficient from a storage, memory, and computational standpoint. The additional storage cost is approximately one extra byte of data per user account over a salted hash. The memory overhead of PolyPasswordHasher is about a kilobyte of information, independent of the number of passwords stored. The computational primitives are fast and well known such as XOR, a salted hash function (SHA256), AES, and Shamir Secret Sharing [61]. As such, even when run on a three year old laptop, PolyPasswordHasher can process tens of thousands of authentication requests per second.

This work presents the first password protection scheme that 1) protects against an attacker that can read all persistent storage on the server including the complete password file, 2) uses only software changes on the server (no hardware changes, additional servers, or client changes), and 3) requires exponentially more effort for the attacker. As such, PolyPasswordHasher increases the difficulty of cracking passwords while remaining easy to deploy.

2 Threat Model

Suppose that a server wants to validate passwords for user accounts. We assume that:

- An attacker can read all data that is persisted on disk including the password file.
- The server can fail and need to be restarted by administrators. All state that is kept in memory is lost at this point. The server must restart using only the (attacker visible) data from disk. (While the basic technique needs a threshold of correct passwords to perform verification, Section 5.2.3 discusses an extension to immediately verify passwords upon restart.)
- The attacker can have a priori knowledge of the correct passwords for some number of user accounts. We assume that the administrator has configured the threshold to be larger than this value. (Note that a technique to have protected user accounts that do not count toward the threshold is described in Section 5.2.1.)
- The attacker **cannot** read arbitrary memory for all processes on the server. If an attacker can read arbitrary memory, then the attacker can observe the plain-text passwords as they are entered and so any form of secure password storage can be bypassed.

  The attacker cannot read arbitrary memory in many types of reported attacks including compromises of password file data stored on backup media, web server misconfiguration, or information obtained through SQL injection attacks — the primary vector for password file compromises [46, 48].
- Even if these guarantees are not met, the goal is to still retain strong security. Regardless of the attacker's knowledge, the protections provided in this work must never be worse than the best practices used today (salted hashes of passwords).

Limitations and strengths of alternative solutions that take this threat model into account are discussed in Section 6.

3 Background

This work requires two fundamental building blocks, a salted hash function and a (k,n)-threshold scheme.

While many different schemes could be utilized, for concreteness we will describe our system using SHA256 for the hash and Shamir Secret Sharing for the threshold scheme.

3.1 Salted SHA256

A cryptographic hash function, such as SHA256, is one where (with extremely high probability) each hash input produces a fixed size output where it is impractical to 1) find another value that produces the same output and 2) discover the input given an output. This is useful for situations like password checking because the hash of the password can be stored instead of the password. If the password file is compromised, then the attacker must be able to find the input for that output.

However, in practice many users choose the same set of passwords, allowing the easy construction of lookup tables with the hashes of common values [56]. To mitigate this problem, current best practice for password protection schemes is to also use a randomly generated value called a salt. The salt is created at the time the account is generated and this value is included in the password when it is hashed. In this way the same password will have different hash values for different users, because the salt will be different. The salt is typically stored in plaintext along with the hash so that the authentication function can use the same salt value when authenticating the password.

3.2 Shamir Secret Sharing

Shamir Secret Sharing [61] is an algorithm where a secret is divided into a set of n shares. These shares are typically provided to different parties. If a threshold k (specified when the secret is divided) of those are collected, the 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. 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.

For example, suppose one wishes to hide a secret 235 so that it can only be reconstructed if 3 shares are provided. The person hiding the secret can choose random terms and 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: $(1,185)$, $(2,183)$, $(3,229)$, $(4,67)$, etc. A party that has at least three shares can interpolate to reconstruct the full polynomial of $f(x)$ and thus the secret (235).

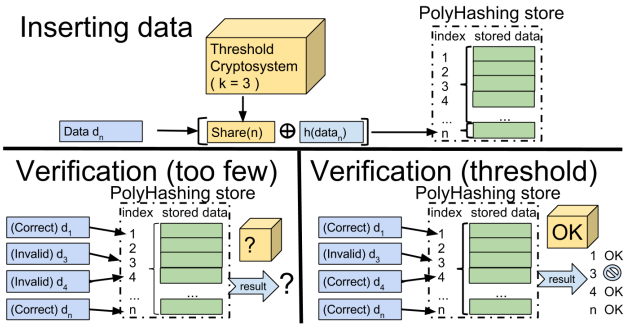

Figure 1: This figure demonstrates inserting data and integrity verification for PolyHashing. In the case of data verification, the caller performs PolyHashing verification on values simultaneously. Only if at least the threshold of correct values are provided (3), can the threshold cryptosystem be decoded and any value be checked. Note that the caller does not know whether data values are correct or invalid when calling the verification routine. Only the data in the dashed box is persisted on disk.

If one requires the ability to generate additional shares after the original share creation, as in this work, a different recontruction procedure is applied. To do this, Lagrange interpolation of the secret during reconstruction is performed instead of simply computing the last term. This makes share recovery slightly more computationally complex, but it is then possible (and efficient) to generate additional shares simply by evaluating $f(x)$ for the specified share.

In many cases, the desired secret will be larger than the size of the finite field. To store this secret, one can break it into pieces that are the size of the finite field (often one byte) and apply the above technique separate on each piece. In this case, the same share number x, is typically used for each share $f_i(x)$.

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 if any share is invalid, the resulting polynomial will be incorrect. To avoid this issue, typical implementations of Shamir Secret Sharing also store an integrity check to detect if an incorrect share has been provided. This may be done by including some portion of the secret (typically a few bytes at the end) that represent a hash of the decoded value. While the techniques in this paper function identically regardless of whether or not this hash exists, we assume its existence to prevent erroneous data from being used as valid.

4 PolyHashing: Integrity Verification

PolyHashing is a general technique for concurrently verifying the integrity of a set of values.The purpose of PolyHashing is much like using a traditional hashing function and storing the hash of a value $h(v_i)$ in the store at a specific index i. However, with PolyHashing, the hash values are blinded so that integrity verification may only be performed if a threshold k of the values are provided. If less than the threshold of correct $h(v)$ items are provided, **no information about which values are correct is leaked.**

These properties are provided by leveraging a threshold cryptosystem (Figure 1). Each entry in a PolyHashing store is XORed with a share. If a threshold of correct values are provided, then the attacker can recover enough shares to recover the secret. By using interpolation, all shares to be computed and thus any hash can be validated. As such, a PolyHashing store has information-theoretic confidentiality when less than the threshold of values are known, but can validate the integrity of values individually if at least k of them are known.

When a blank PolyHashing store is created, the threshold k is given. A cryptographically random value is used to seed all shares in the threshold cryptosystem. For example, when using Shamir Secret Sharing for PolyHashing, the constant term (which is typically the stored secret) is randomly generated. In this case, the purpose of the threshold cryptosystem is not to store a secret, but instead to act like a one-time pad to obscure the hashes that are stored. As such the length of shares generated by the threshold cryptosystem should be equal to the length of the hashes.

To add a secret to the store (the top of Figure 1), one will compute an index i (to later locate which secret in the store corresponds to this entry) and the hash of the data. The share value can be stored in the table, or alternatively i can also be used as the share number in the Shamir Secret Store for computing the $f(x)$ values of the shares. To store an item, the party will add an entry that includes the identifier, share number, and the XOR of the $f(x)$ values with the hash.

If a party possesses the PolyHashing store, they cannot validate any hash value without at least k correct hash values (the bottom left of Figure 1) because they do not have enough hashes to recover the threshold cryptosystem's shares. When a party has k or more hash values (bottom right of Figure 1), they can uncover all shares of the threshold cryptosystem and validate all hash values. As a result, other requests can be validated by computing the hash of the value $h(v_i)$, XORing the ith share, and comparing the result with the stored data in the PolyHashing store.

Note that the threshold cryptosystem's shares are not stored on disk or otherwise transmitted. Once enough correct values are known by a party, by XORing the $h(v)$ values with the data in the PolyHashing store, one can recover a threshold of shares. This allows the party to reconstruct all data in the threshold cryptosystem's store. By keeping this information only in memory on a running system, an attacker that can read the disk contents but does not know a threshold of correct values cannot validate the correctness of values individually.

More formally, given an information-theoretic (k,n)-threshold cryptosystem with shares $S = \{s_0, ...s_n\}$ that stores values $H = \{h(d_0), ...h(d_n)\}$. Suppose that $length(s_i) = length(h(x))$. A PolyHashing store Z has items $Z = \{z_0, ...z_n\}$ such that $z_i = s_i \oplus h(d_i)$.

Theorem: Z provides information-theoretic privacy for unknown items in H when $k-1$ items from H are known.

Proof: Suppose that $h(d_j)$ is an item that is not in the set of $k-1$ known items. Since S is a (k,n)-threshold cryptosystem and the observer knows only $k-1$ items, they only know $k-1$ shares of S. Thus they cannot recover S from the known shares of S. Therefore s_j has information-theoretic privacy. Since z_j is the result of XORing s_j with another value of the same length, s_j acts like a one-time pad for $h(d_j)$. Thus, z_j reveals no information about $h(d_j)$. However, since the choice of j was arbitrary, the same argument applies to all of the $n-(k-1)$ unknown items in H and clearly no information about $h(d_j)$ is revealed by any z_i where $i \neq j$. ■

5 PolyPasswordHasher: Password Vertification

This section describes one possible use of PolyHashing, protecting password data stored on disk. The resulting system is called PolyPasswordHasher. For ease of explanation, we describe PolyPasswordHasher using the hashing function (SHA256) and threshold cryptography (Shamir Secret Sharing) used in its implementation. Passwords may only be checked by PolyPasswordHasher if a threshold of correct passwords are known. After describing the core algorithm for achieving this, this section discusses domain specific extensions to this technique that handle thresholdless passwords (those which do not count toward the threshold) and alternative authentication mechanisms, such as private key authentication, biometrics, and single sign-in systems.

5.1 Core PolyPasswordHasher Algorithm

Much like other password storage schemes, PolyPasswordHasher stores password data to allow users to login using a username and password. However, unlike other schemes, if the encrypted password file is stolen, the users' passwords are safe and cannot be individually cracked unless at least a threshold of them are known. This is true even if all data stored on disk is stolen by an

attacker.

With PolyPasswordHasher, a PolyHashing store is created to store user login data. PolyPasswordHasher makes a few minor additions to the default PolyHashing scheme (top half of Figure 2). First of all, similar to popular login systems, the store is keyed by username. The corresponding share number (that the index above derives) is stored explicitly in the database. In this way, if an entry in the database is deleted or added, the username to share number mapping remains to allow recovery. Since the set of used passwords is quite small and predictable, we use a salt [40].

When a system using a PolyPasswordHasher store restarts, no authentication can be performed. First an initial threshold of passwords must be provided before any can be checked. When the system restarts, a set of users will try to authenticate in a normal way. When sufficiently many correct passwords are obtained (discussed below), the server can reconstruct the random coefficients and authenticate users. Once the initial set of correct authentications are performed, all subsequent authentications are fast and efficient.

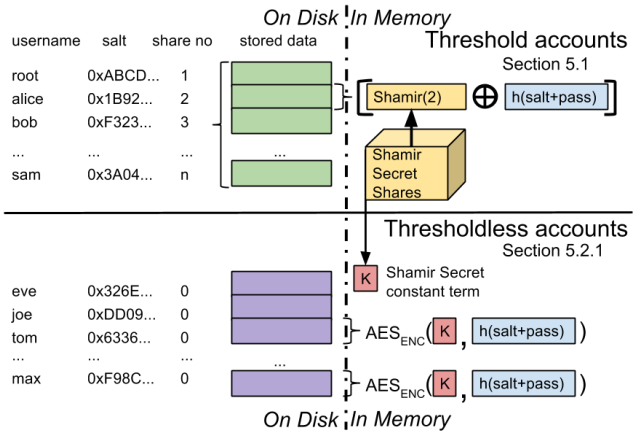

Figure 2: The top portion of this figure demonstrates the data stored by the basic PolyPasswordHasher algorithm. The bottom portion shows how thresholdless accounts have the salted hash of their password encrypted by an AES key. This AES key is stored as the secret for the threshold accounts and can only be recovered by the system if a threshold of administrator passwords is known.

5.1.1 What Is Stored

As the top half of Figure 2 shows, the server stores a file (or database) that contains records of the form: <username, salt, sharenumber, (share XOR hash)>. As may be expected, the username is the user's login. The username, salt, and sharenumber will all be known to an attacker that steals the database. However, as with PolyHashing, the share and hash are effectively random unless a threshold of shares (and thus hashes of passwords) are known.

Note that the random coefficients for Shamir are private and are not stored on disk. They are instead reconstructed in memory through Lagrange interpolation assuming an adequate number of shares are known.

5.1.2 Checking A Password

Assuming the server has enough valid passwords (and thus shares), the server can thus recover all shares. The server computes the salted hash of the user's password and XORs it with the last entry from the table (share XOR hash) to obtain the share. If the share is valid, the password is correct. The share will consist of a set of x values and $f(x)$ results. To compute validity, one can evaluate $f(x)$ at the respective values and check to see if they match.

Note that, a single account could optionally provide access to multiple shares. To do this, there can be several entries with the same username that unlock using the same password, however each entry will have a different salt and share number which leads to different stored data. In this case, the same password is hashed with different salt and share numbers which effectively re-randomizes each component of the stored data.

5.1.3 Creating An Account

As is shown in the top half of Figure 2, to add an account entry, one must have four items: username, salt, share number, and stored data. The server randomly generates a salt, chooses an unused share number, and uses the password's hash and share to compute the shared data. For example, if a user `alice` needs an account, the system will find an unused share number, such as 2 in the above example. Then the system computes a random salt, prepends it to `alice`'s password and computes the hash of this value. This hash is XORed with the Shamir Secret Share's share number 2 and is stored as the fourth field of the PolyPasswordHasher store. The username, salt, share number, and stored data are written to the PolyPasswordHasher store.

5.1.4 Changing A Password

Changing a password is a simple process for an unlocked store. Optionally, the procedure for account change may require validating the existing password first before allowing replacement. The remaining steps required to change a password are nearly identical to account creation. The actual password change simply involves creating a new entry with the same username and sharenumber, but a different salt. Since the salt and password have changed, the final field will also change. Once the new password entry is computed, it can replace the old one.

If the PolyPasswordHasher database is also likely to

be compromised, then it makes sense to additionally change the cryptosystem's secret information. The reason is as follows. Suppose an attacker compromises the password database repeatedly and knows the password for different accounts over a period of time. From this information, the attacker may be able to recover enough shares from the cryptosystem even if those accounts were not active at the same time. To combat this, one can easily change the cryptosystem's source of randomness by computing a new random cryptosystem and XORing both the old and new shares with each stored data entry. This effectively removes the prior cryptosystem data while simultaneously adding randomness from the new cryptosystem. In this case, no user passwords are changed, so this operation is not visible to the users in any way.

5.1.5 Unlocking a PolyPasswordHasher Store

To recover the random coefficients, the server takes a series of logins that have more than the threshold number of entries. The server can either batch login requests and issue them all at once to achieve this or simply wait for administrators with a sufficient threshold to attempt to login. Then the password hashes are computed and the shares are recovered. Assuming that the provided information is valid, the cryptosystem will be able to use a technique like Lagrange interpolation to recover all shares.

If more than the threshold number of shares are provided and some may be invalid, individual user accounts (which may have incorrect logins) can be removed from the threshold to test their validity so long as at least k entries are validated. Furthermore, Section 5.2.3 describes an extension to partially validate passwords and discard a large percentage of incorrect values even before the threshold is reached.

5.2 PolyPasswordHasher Extensions

There are three major limitations of PolyPasswordHasher as described in the previous section. First, for some systems like Facebook or gmail, an attacker may control a huge number of accounts. In practice there are really two types of accounts: those that should count toward the threshold (likely administrators or power users) and those that should not. This section describes an extension to support "thresholdless" accounts that do not count toward the threshold. Second, there is not a way to handle non-password authentication schemes like biometrics, private key authentication, etc. This section discusses how to integrate into existing mechanisms. Third, the system must have a threshold of correct passwords before any can be authenticated. This may cause logins to be delayed for an unacceptable time after a restart. This section discusses an extension that leaks partial information about salted password hashes, but allows immediate authentication upon restart. As a result, PolyPasswordHasher still provides an exponential security increase, but no longer must wait for account logins upon restart.

5.2.1 Thresholdless Accounts

A desirable property is the ability to handle user accounts that should not count toward the threshold. For example, gmail and Facebook have a huge number of users that are untrusted and can create their accounts automatically. None of these user accounts, no matter how many, should be allowed to count toward the threshold for verifying other accounts.

We will support accounts that should not count toward the threshold by computing their salted hash and then encrypting this data with a symmetric cryptographic cipher (AES). If the server knows the AES key, it can clearly authenticate thresholdless users. However, if the AES key is stored on disk, an attacker that compromises the password file could compromise the AES key too.

To protect the AES key while stored on disk we will use a slightly modified version of the PolyHashing store algorithm. This store is identical to the main algorithm except that the lowest order term (the constant term) of the polynomial represents a portion of the encryption key, as is shown in the bottom half of Figure 2. (The key is stored in the same manner as traditional Shamir Secret Sharing where the constant term of the Polynomial encodes the secret.) A party that knows the threshold of administrator passwords can XOR those values to recover sufficient shares to reconstruct the polynomial (and thus the key). By the properties of Shamir Secret Sharing, the key has information-theoretic privacy from an attacker unless they possess a threshold of account passwords.

Creation of a thresholdless account involves a similar series of steps to other PolyPasswordHasher passwords. As before the system obtains the salted hash for the password. However, the salted cryptographic hash of the password is encrypted with AES instead of XORing it with a share. To indicate this is a thresholdless account, the share field can be set to 0 to indicate that this is not a normal PolyPasswordHasher store. (Note that the share 0 will not be used normally in Shamir secret sharing.) When the user attempts to login, if a large enough threshold of passwords has been entered, the AES key is available to decrypt the hash, allowing the user to be authenticated. However, if the stored threshold data is disclosed, it is not useful to an attacker unless they can recover the AES key by knowing a threshold of administrator passwords.

If desired by an administrator, accounts can be switched between thresholdless and threshold without user intervention. In both the case of threshold and

thresholdless accounts, a server with a threshold of shares knows the information necessary to recover the salted secure hash. This salted secure hash can then be re-encoded in the alternative manner and the new entry can be stored.

5.2.2 Handling Alternative Authentication Mechanisms

Passwords are not the only mechanism for logging into modern systems. For example, it is common for users to use a private key to login 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 login for *thresholdless* user accounts is trivial because it requires no changes to the system. Non-password login can be handled using the existing login mechanisms without any modification. This is especially important for techniques like OAuth and OpenID which are almost certainly not going to be used to protect administrator accounts (which count toward the threshold), but will be desirable to average users.

For administrator accounts or other accounts which are expected to contribute to the threshold, one must devise other techniques. Fortunately there is a significant amount of prior work on secure remote authentication and techniques for using this to hide secrets [17, 78]. This can be used to protect the Shamir Secret Share(s) for the account. It is possible to view the authentication as decrypting information using a secret stored by a remote party (as in fact many such systems are implemented). We can encrypt the Shamir Secret Share for the user with their key and then check that the resulting x and $f(x)$ are correct for the user.

For example, suppose that an administrator has a private key that they use to login (as is common with SSH private key authentication or smart card based authentication). Instead of XORing the Shamir share with the salted hash of the password, it can be encrypted with the user's public key. When the administrator presents their key to login, this key is used to decrypt the original share. This process can be repeated multiple times if the user has multiple shares. The share can then be used the same as any other share to unlock a PolyPasswordHasher data store.

For authentication of threshold account using other techniques such as biometrics, there has been prior work on deriving a private key from this (noisy) data [34]. Once a key is derived from the biometric data, the scheme will function identically to the private key authentication scheme discussed above.

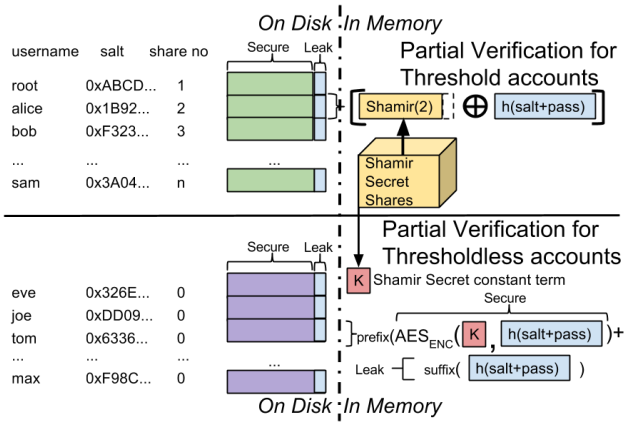

Figure 3: This figure shows validation using partial verification. A portion of suffix of the salted hash is stored on disk. This allows verification of accounts before a threshold of correct passwords is provided.

5.2.3 Partial Verification

When a PolyHashing store restarts, no values may be authenticated until a threshold are provided. If the time delay is not important and the rate at which values are provided is rapid, this may not pose a problem. However, in many scenarios, this delay may prove burdensome.

This issue can be addressed with a technique called *partial verification*. Partial verification has the password database leak partial information about the hash to allow verification before a threshold is reached. The core idea is similar for threshold accounts and thresholdless accounts, but for simplicity will be explained with threshold accounts first. If the Shamir Secret Share is chosen so that it is shorter than the hash value, this will leak some bytes of the salted hash. The 'partial' hash that is leaked can be used for verification purposes.

For example, suppose there is a 32 byte salted hash and a 30 byte secret (Figure 3). The secret will XOR with the first 30 bytes, effectively obscuring them. However, the remaining 2 bytes will consist of the suffix of the salted hash. When the system restarts, accounts can be validated using the last 2 bytes of the hash. Once sufficient accounts have logged in, the full hash can be recovered and account authorizations can be performed using the full hash as was described in the previous section.

Thresholdless accounts can also be validated before a threshold is reached if partial validation is used. The technique is similar to threshold accounts, where a portion of the suffix of the hash replaces some of the encrypted hash on disk. Validation can be performed with the suffix until a threshold of correct passwords are provided and the Shamir Secret Store (and thus the AES

key) are recovered.

Increasing the length of the hash which is leaked has a negative effect on the resulting data store security. First of all, suppose that l bits are leaked with via the hash. This allows an attacker that compromises the password hash database to discard 2^l possibilities from the password search space. Thus if there is a threshold of k and each password has n bits of entropy, the attacker can make k passes of cost 2^n to remove all but 2^{n-l} passwords. Following this, the attacker will then need to simultaneously guess from the remaining passwords which will cost an additional $2^{k*(n-l)}$ to crack the store. (Section 8.3 discusses the practical impact this has on password security.)

However, a small hash also has a negative ramification that incorrect passwords could log in before a threshold is reached. In fact, the probability of a random password working is $\frac{1}{2}^l$. Rate limiting the rate of password attempts can mitigate the risk from an attacker that does not know the salt. However, if an attacker has stolen the password database, they can quickly compute a value that will be correctly verified upon restart. However, once a threshold of passwords is provided, PolyPasswordHasher knows the full hash and can detect any such malicious logins. Much like other decoy schemes [33, 42], partial verification allow incorrect logins but later detects this occurrence and provides strong evidence that the password database was leaked.

6 Discussion

This section discusses strengths, limitations, and expected uses of PolyPasswordHasher as compared to other techniques.

How does PolyPasswordHasher compare to having an administrator enter a key at boot time to unencrypt the password database in memory?

With the administrator entering a key, the system cannot validate passwords upon startup until an administrator intervenes. With partial verification, PolyPasswordHasher can process user authentications immediately. Furthermore, if the administrators enter a key, all system administrators would likely share this key to be able to quickly act to restart the system. From a security design standpoint, it is considered bad practice to share passwords. With PolyPasswordHasher different administrators do not need to share a password or key. Each administrator may have their own password allowing access control to happen naturally as administrators join and leave the organization.

Why not store a key to decrypt the password database in hardware instead?

Storing a key in trusted hardware (such as a USB dongle [55]) has security benefits, but also has significant 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. On the other hand, PolyPasswordHasher is entirely software based and a backup system can be brought online in the same way that a normal server boots.

How does an attacker that can read arbitrary memory on the authentication server impact PolyPasswordHasher?

If an attacker can read arbitrary memory on a running server where the threshold of authentications has been met, the attacker can recover the Shamir Secret Store. This allows the attacker to recover the hashes for each password. An attacker could then crack passwords individually in the standard way. While these are not the majority of disclosed database compromises [46, 48], neither PolyPasswordHasher nor techniques like hardware key storage are significantly helpful in this situation.

Furthermore, new processor extensions such as Intel SGX [47] expressly provide protections to select portions of user space code so that even a root user, hypervisor, or the operating system kernel cannot read it. Servers running PolyPasswordHasher can leverage these extensions to protect the Shamir Secret Store.

However, if an attacker can read arbitrary memory, the attacker can gain access to plaintext passwords as they are provided by clients. This is because typically the client provides the server with the password, which the server then combines with the salt to get the hash. So without changes to the client software, any server-side password storage technique will leak plain-text passwords to an attacker that can read arbitrary memory on a running server.

How does PolyPasswordHasher change how users interact with existing password authentication systems?

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 the administrator's toolchain, the only change is that when an account is created, the administrator can specify if the account counts toward the threshold or not.

How does using PolyPasswordHasher integrate with existing password storage formats?

The file format for password storage is similar to existing systems. For systems which use a database, one can simply insert the share number as a new column in the table and change the authentication logic to include PolyPasswordHasher.

However, many servers (such as Linux, Mac and BSD)

use the `/etc/shadow` or `/etc/master.password` file colon delineated formats. The 'password' field contains both the salt and hash but these different portions of the field are not delineated. For PolyPasswordHasher, one can also add the share number at a known byte location within the 'password' field (such as the beginning or end), making this variable length, opaque string one byte longer.

If there is a single administrator, is PolyPasswordHasher useful?

There is value in PolyPasswordHasher whenever there are multiple accounts, even if only one is an administrator. If an attacker steals the password file, but does not know the password for the administrator account, the attacker cannot crack other user accounts. Assuming that the attacker does not know the administrator password from other sources, such as its use on multiple sites, the attacker cannot crack a thresholdless password without cracking the administrator password.

What threshold values are likely to be used?

Ignoring partial verification, k of n administrators (or power users) must login to unlock the PolyPasswordHasher store. The value of n is largely irrelevant except it must be smaller than the field size since there are only $n-1$ Shamir Secret shares. (Note n only includes accounts that should count against the threshold.) We believe that most organizations will have a small value for k, such as 1 to 5. As our results later show, when coupled with non-trivial passwords, even a small value of k increases password strength immensely.

When is PolyPasswordHasher most (and least) valuable?

When only a single user account exists, such as on a smartphone, there is no benefit to using PolyPasswordHasher on the local device. However, essentially all devices are networked and interact with servers. Web and cloud services benefit from PolyPasswordHasher as do any devices that support multiple user logins. Services that manage logins from multiple parties, substantially benefit from PolyPasswordHasher. This is because PolyPasswordHasher makes a password file compromise only impactful if a threshold of administrator passwords are already known.

What happens if so many threshold users forget their passwords that a threshold cannot be reached?

If there are not enough known threshold users, all password data in the password file is useless. However, this does not mean that the device is unusable. If used, partial verification will still allow users to log in. However, it will not be possible to create new accounts since Shamir Secret Shares cannot be generated and the AES key protecting the thresholdless accounts is not known. Furthermore, mechanisms like root password recovery through the console will still work, allowing any data on the system to be accessed.

What happens if users choose extremely weak passwords like 123456, letmein, and password?

Ignoring partial verification for the moment, if extremely weak passwords are used for a threshold of threshold accounts (like administrators), PolyPasswordHasher will not provide strong protection. If there are only a few bits of entropy in the password, while exponentially larger, the search space will still be small. It is thus critical that the administrators choose strong passwords. However, if a thresholdless account uses a weak password, the attacker must first crack a threshold of accounts, thus providing strong protection.

If partial verification is used, then some bytes of the password's hash are leaked. If an extremely weak password is used, this will be effectively leaked to the attacker. One should think of partial verification as reducing the password strength via the number of leaked bits. Thus a password with fewer bits of entropy than the partial verification length, can be cracked as though it is protected using today's best practices.

Independent of all of this, extremely weak passwords are susceptable to guessing by an external party (without database access) and as best practices sites should block them from use [5].

7 Implementation

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 (encoding each byte as a separate share). The full code base for PolyPasswordHasher is 203 lines of Python code and 223 lines of C code[2]. PolyPasswordHasher uses Python's standard hashlib for SHA256 and the PyCrypto implementation of AES. The GF256 operations, Lagrange interpolation, and polynomial math code are written in C for performance reasons. All other code is written in Python. The main code for PolyPasswordHasher, which handles account creation, accessing a persisted store, writing a store to disk, checking passwords, and similar operations, is 87 lines of Python code. Adding support for thresholdless accounts involved adding 20 lines of Python code. Partial verification added 13 lines of Python code. The remaining Python is wrapper code for the C-based Shamir Secret Sharing implementation.

8 Evaluation

This section describes a performance evaluation of our prototype of PolyPasswordHasher. To understand the performance of PolyPasswordHasher, we performed a

[2] All line of code counts are according to SLOCcount [75]

Figure 4: Time for PolyPasswordHasher operations. The plots for thresholdless creation and verification overlap as do the plots for the threshold versions of each action.

series of microbenchmarks on an early-2011 MacBook Pro with 4GB of RAM, a 2.3 GHz Intel Core i5 processor.

All operations are the mean verification time across 100 runs and are performed with the password file already present in memory. For benchmarking purposes, each action is performed sequentially despite being embarrassingly parallelizable.

8.1 Performance Of A PolyPasswordHasher Store

Figure 4 shows the time taken by different operations (discussed below). Unless noted below, the time of the operations does not depend on other factors such as the number of accounts in the password database.

Account Verification. The mean verification time of a threshold account varies from 57μs to 163μs depending on the threshold. With a threshold of 8, this allows the verification of more than 16K user accounts per second.

Thresholdless accounts are verified in a time that is independent of the threshold size because it merely computes a salted hash and performs an AES encryption operation. Verifying a thresholdless account takes about 29μs, allowing about 35K such actions to be performed each second. In comparison, the current best practice of generating the SHA256 hash of a known salt is a little over 3μs on the same hardware, which allows hundreds of thousands of account authentications per second.

Account Creation / Password Change. The account creation time is similar to that of password verification. Depending on the threshold, the average verification time varies from 77μs to 190μs. A store with a threshold of 8 can create more than 12K accounts per second. Given there are a maximum of 255 threshold accounts that can be created (at least with a PolyPasswordHasher implementation that uses GF256), this clearly is not a performance concern.

Much like account verification, thresholdless accounts can be created in time that is independent of the threshold. A thresholdless account creation takes about 25μs. As a result, about 40K thresholdless accounts can be created each second. This is similar to the time it takes to generate a salted SHA256 hash for a provided password (16μs).

Note that changing the password requires that PolyPasswordHasher performs the same operation as account creation (potentially along with authentication of the old password).

Initializing a Store. The time to create a store varies depending on the threshold. This cost is dominated by generating cryptographically-suitable random numbers. By varying the threshold, the creation time varies from 380μs to 1.13ms. This operation is only performed when a new password file is created.

Password Source	Original Disk Space	Salted, Hashed Disk Space	PolyPasswordHasher Disk Space
eHarmony*	51.6MB	100MB	102MB
Formspring*	27.3MB	34.8MB	35.2MB
Gawker	75.2MB	119MB	120MB
LinkedIn*	252MB	424MB	430MB
Sony	2.98MB	4.95MB	5.00MB
Yahoo	17.8MB	35.0MB	35.4MB

Table 1: Disk space to store password data from different account disclosures. Entries with a * indicate only the password hash portion of the database was leaked. Other breaches include usernames and similar data.

Unlocking a PolyPasswordHasher Store. When the server restarts, the random coefficients are computed from the set of provided shares with full interpolation. The time needed varies between 202μs and 2.87s as the threshold changes since the number of polynomials changes as the store size grows. Note that small thresholds are very fast, with a store with a threshold of 8 being unlocked in 617μs.

This is fast enough that if some passwords are incorrectly entered, PolyPasswordHasher can (naïvely) detect them. For example, suppose that a threshold of 10 is used and that 14 passwords are provided and only 10 of them are correct. All possible combinations of 10 passwords can be checked by PolyPasswordHasher in 618ms. Partial verification will also discard most passwords that are entered incorrectly to prevent them from being used to attempt to unlock the password store.

8.2 Memory and Storage Costs of PolyPasswordHasher

Storing passwords with PolyPasswordHasher requires additional information be stored for each account. Namely, for each account, since each share is a value in GF256, there must be a one byte share number stored for each account. This represents an additional 1 byte of storage space for each account in addition to the cost of current hash techniques. This has a minimal impact on the disk space needed to store production password databases (Table 1). If the thresholdless accounts are stored in a separate file or are otherwise distinguished, then only threshold accounts require the extra byte of storage. This will result in a cost for those accounts that is identical to the salted, hashed scheme.

In addition, any account that has multiple entries (to count multiple times toward the threshold), requires an additional salt (16 bytes), hash (32 bytes), and share number (1 byte) for every entry after the first. Since there are at most 255 Shamir secret shares in GF256, this can

Figure 5: Time to crack a password store given 1 billion attempts per second. The k=1 case for PolyPasswordHasher is the same as the legacy salted hash scheme. The partial verification values show how leaking 2 bytes impacts cracking time. A threshold of 3 without partial verification falls well outside the axis of the graph.

be treated as a fixed cost of a few kilobytes.

There is additional memory cost because the server must cache the polynomial coefficients for a store. The total size of this data is the threshold value multiplied by the hash length. Thus, the in-memory threshold data is likely to be under a kilobyte in practice.When thresholdless accounts are used, the AES key used is the constant term of the polynomial coefficients and so does not incur additional cost. The use of partial verification has essentially no impact on the storage or memory requirements.

8.3 Feasibility Of Cracking Passwords

General Analysis. Even when an attacker needs to guess only a few passwords, in many cases PolyPasswordHasher's *exponential increase* in guessing time ($O(V^p)$ instead of $O(pV)$) makes guessing computationally infeasible (Figure 5). For example, suppose an attacker wants to guess three passwords that they know are each comprised of 6 randomly chosen characters[3]. Recent results have shown that a GPU can compute on the order of a billion password hashes a second [21, 80], allowing the attacker to search the key space for all three passwords in under an hour. Thus the current state of the art provides little protection in this case.

In the case of PolyPasswordHasher, the attacker must *simultaneously* guess all three passwords. This means the attacker needs to guess from 3.97×10^{35} values. This is roughly 23 orders of magnitude more effort. When PolyPasswordHasher is used with the same passwords and the GPU accelerated technique, searching the key space would take 1.25×10^{19} CPU years (does not fit on the axis of Figure 5). To put this number into context, ignoring smartphones there are about 900 million computers on the planet [15]. The estimated age of the universe is $(1.3798 \pm 0.0037) \times 10^{10}$ [72]. The time needed to crack three random 6 character passwords protected by PolyPasswordHasher is more CPU time than would be provided by every computer on the planet working nonstop for the estimated age of the universe!

Even a threshold of two is substantially stronger than existing best practices. Searching the key space would require over 17 million CPU years of effort.

[3]These passwords are extremely weak and we do not advocate their use. This example is used to illustrate the strength of PolyPasswordHasher even with weak passwords.

Partial verification. Partial verification allows an attacker to first reduce the search space for a specific account by eliminating accounts that do not match the leaked bits. If the attacker knows the password pattern, the attacker can first precompute all passwords that match that pattern and hash. This requires the same amount of effort ($k * 2^n$) as cracking passwords when stored with traditional best practices. If the attacker has sufficient space to store these passwords, the attacker then may use combinations of these precomputed passwords to try to unlock the store. This allows the attacker to search the space for the Shamir Secret Store in 2^{k*n-l} guesses. Each byte used for partial verification effectively reduces the strength of a random password by approximately 1.22 characters. However, the time needed to unlock the Shamir Secret Share still represents an exponential increase and dominates the overall cost.

For example, if 2 bytes are leaked for 6 random character passwords (Figure 5), the attacker will need to do 735 trillion operations for each of the three passwords and then 1.41×10^{21} operations to crack the password store. With one billion operations per second, sweeping the search space would still require 45 thousand CPU years (8 orders of magnitude more time than salting and hashing).

Case Study. To explore how these results hold for real passwords, we performed an experiment using the password data dumped from the Sony account breaches [66]. Of the leaked passwords known to the authors (Table 1), this is the only data set which explicitly lists which accounts are administrator accounts versus normal users.

In the database dump, there is password data for both outside user accounts as well as accounts have administrator access. (There is also a database with testing accounts which we ignore.) The four administrator accounts have passwords with the estimated entropy in bits [74]: password@1 (5.322 bits), welkom@1 (26.553 bits), waderobsen (30.618 bits), and itsafullcyrcle (44.011 bits). Given a rate of checking a billion password hashes a second, the first three passwords can all be cracked in two seconds. The remaining password (and thus every administrator password) could be cracked in under 5 hours.

If PolyPasswordHasher were protecting these passwords, the ability to crack the passwords depends on the threshold. The effective password strength is that of the weakest passwords combined, times the probability of selecting those accounts in that order. A threshold of 3 will have an effective entropy of 67.078 bits, which will take nearly 5000 years to crack at 1 billion guesses a second.

With a threshold of 2, the effective entropy is 35.459 bits, which can be cracked in under a minute. However, if all administrators had chosen a password as strong as the

password waderobsen (which is considered a weak password by best practices [29]), then the password would have had an effective entropy of 64.820 bits, which will take more than a thousand CPU years to crack at 1 billion guesses per second. This underscores that administrators still should not choose immensely poor passwords like password@1. Independent of password storage, avoiding trivially guessable passwords is essential to prevent remote brute-force password cracking.

The thresholdless user passwords (many of which are extremely poor) cannot be cracked until a threshold of administrator passwords are compromised. This means that even if only administrators can be convinced to use strong passwords, PolyPasswordHasher provides substantial security benefits.

9 Related Work

There has been extensive work on password security stretching back many years [51, 40, 23]. Password database disclosure is a problem that, if anything, seems to be more prevalent as time goes on [14, 48, 46]. Password security has been the focus of much study with many promising solutions solving different portions of the problem [70].

Our work on PolyPasswordHasher is unique in that is the first password database protection scheme that:

i assumes the attacker can read all persistent storage,

ii requires only a software change on the server,

iii and requires exponentially more time for the attacker to crack passwords.

PolyPasswordHasher can be deployed with minimal server changes and without modifying clients at all.

Multi-server Password Authentication. There are a wide variety of authentication schemes that use multiple servers to store password data [11, 3, 35]. The assumption is that the attacker cannot compromise a threshold of the servers. In contrast, PolyPasswordHasher uses a single server but uses a threshold system to hide information that can only be unlocked with a threshold of correct user passwords.

Decoys. Recently, researchers have suggested multiple techniques that use a set of extra password entries [42, 33]. For example, the Honeywords [33] system uses a separate server holds information about which password entry is correct. If an attacker obtains the password database then they do not know which password entry is correct. Entering a password which matches the hash of a different password entry will trigger an alarm which notifies the administrator of a password hash file breach. However, for this to work, there must be a separate, secure server which authenticates the index of that entry in the file (a one byte value). The real value of HoneyWords is that it can also operate when that server is offline and / or check passwords at a later time to detect breaches later. PolyPasswordHasher utilizes ideas from these works in constructing the partial verification technique discussed in Section 5.2.3.

Key Stretching. One way to mitigate the effectiveness of password hash cracking is to use techniques for key stretching [38]. This involves performing multiple rounds of cryptographic operations to validate a key. This effectively slows down both the attacker's cracking of passwords and the user's authentication by the same factor. In contrast, PolyPasswordHasher results in an exponential increase in the amount of time needed by requiring multiple passwords to be simultaneously guessed. Key stretching is orthogonal to PolyPasswordHasher and could be trivially used in conjunction.

Bounded Attacker. Di Crescenzo [19] proposed a scheme for protecting password data when an attacker can only read a bounded amount of data from storage. This works by an organization configuriing network monitoring hardware and setting up a separate server to process authentication requests. In the widely published password file compromises, the attackers were able to steal complete password file data [48, 46]. In contrast, PolyPasswordHasher requires no network changes or monitoring and works even when an attacker has complete access to stored information, such as a disk backup.

Prior work by Gwoboa [26] 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. Clearly if all persisted data is known, this key (and thus all passwords) are at risk. In PolyPasswordHasher, as long as a threshold of passwords are not known, all persisted data can be stolen by an attacker without compromising user passwords.

Biometrics. Biometric authentication has substantial promise for secure authentication [2, 65, 71, 7, 22, 39, 54, 49, 58]. There has been a substantial amount of work on how to store and authenticate users with this information. Like PolyPasswordHasher, some of these systems use a threshold of information to validate and authenticate users, in part to deal with noisy biometric data [34, 4]. While users must still remember a password to use PolyPasswordHasher, it does not require client-side hardware.

Prior work uses keystroke dynamics to change stored password data [50]. This technique relies on reading timing information from when the user types their password into a site. This provides promising protections, but re-

quires changes to the client and server to correctly operate. In comparison, PolyPasswordHasher protects password hash information with no change to the client and minimal server changes.

Authentication Using Tokens or Smart Cards. Much authentication has looked at authentication in the context of banking [17, 79], health services [1], or a more general context [13, 78]. These systems are extremely effective and are widely used for banking and protecting access to classified systems. Unfortunately, these devices incur a per-user cost and thus are not often used in contexts where the user and server have no prior commercial or authentication relationship. PolyPasswordHasher can be applied to webmail systems and social networks where this relationship does not exist a priori.

Two-Factor Authentication. The use of two-factor authentication [20] is provided by some popular services (typically through a user's smartphone). The use of two-factor authentication does not change PolyPasswordHasher's use in any way. Users can easily get the best of both protections by a simultaneous deployment of each technology.

Multiparty Computational Authentication. There are a variety of schemes that perform secure, remote authentication using computation by the client and server on legacy hardware [76, 44, 12, 32, 24, 10, 9, 36, 37, 25]. 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 which require changes on clients and servers. They also do not function in non-distributed scenarios. PolyPasswordHasher works with no changes to clients, minimal visible changes to the administrator and operates on a single system.

Related Key Exchange Schemes. There are also many systems for secure key exchange [64] such as Password-Authenticated Key Exchange (PAKE) [8, 63, 59, 30], Encrypted Key Exchange (EKE) [67, 45, 31], and further enhancements [73]. These systems allow parties that share a password to securely find an encryption key to hide communications. These systems provide excellent protection and can handle compromises of the memory of systems in some cases. However, unlike PolyPasswordHasher, they typically involve multi-round authentication and require changes to both the client and server.

Helping Users Choose Stronger Passwords. There have been many efforts to help users to choose stronger, more memorable passwords or expose weak passwords [69, 40, 6, 60, 41, 62, 77]. These techniques can be very effective at protecting users, but must be adopted by users. PolyPasswordHasher provides an exponential improvement in protection for user passwords. Assuming threshold passwords are strong (which these methods assist with), PolyPasswordHasher strengthens the protection of stored user passwords. This is especially critical when partial verification is used.

Password Managers. There are a myriad of password managers that help users choose secure per-site passwords including LastPass, 1Password, and OnePass. These systems store password data and lock it using a user's credentials. As a result, it is the case that the third party software (and often their server) will know the user's password.

To mitigate this, several groups have proposed cryptographic techniques to take a user's password and generate secure, per site passwords [27, 57, 28]. These techniques are effective (and more secure) but can create passwords that are incompatible with the server's password policy.

These techniques require client side changes (only) while PolyPasswordHasher requires only server side changes. Both can (and should) be used safely in conjunction for improved security.

Single Sign-On. Single sign-on systems like OpenID and OAuth have promise for organizations to securely offload authentication to a third party. This proves convenient for users, but is far from ubiquitous for a variety of reasons [68]. These systems have some security issues [53, 52], but overall can be effective when properly used. PolyPasswordHasher integrates cleanly with non-administrator logins for such systems. In addition, PolyPasswordHasher can be used by the Single Sign-On provider to provide security to the authenticating users.

Non-password Authentication. Many researchers have proposed authentication based upon non-password items such as pictures [18]. In practice, these systems can have security limitations if users do not appropriately choose their authentication tokens [16]. For exotic authentication mechanisms like this, PolyPasswordHasher functions well for non-administrator accounts, requiring no changes to the system.

10 Conclusion

This work addresses an important problem in the security community by protecting user passwords in the event of password file disclosure. Previously an attacker that obtained a salted and hashed password file could easily crack passwords one at a time. Now, assuming the attacker must have access to a threshold of passwords to crack individual passwords. As a result, password cracking requires exponentially more work for the attacker and is infeasible in many cases.

Our implementation of PolyPasswordHasher demonstrates that the storage and performance properties are similar to systems that are widely used in practice. PolyPasswordHasher integrates naturally with alternative authentication mechanisms, tools, and techniques. In ongoing work, we are deploying PolyPasswordHasher in

several domains including a production web service used by thousands of users and a cloud computing infrastructure.

Our reference implementation is available with an MIT license at: https://polypasswordhasher.poly.edu.

11 Acknowledgments

PolyPasswordHasher benefitted tremendously from feedback and assistance by many outside parties. Foremost is Santiago Torres, who produced a C implementation based upon the specification. Lisa Hellerstein and Dennis Shasha provided excellent feedback on the core ideas at an early stage. We are indebted to Justin Quick, Mike Amling, Peter Maxwell, Christopher Taylor, Gregory Maxwell, Jim Manico, Justin Samuel, John Hoffman, Andy Lutomirski, as well as participants in the Password Hashing Competition for their excellent feedback, advice, and suggestions.

This work was supported by the New York State Center for Advanced Technology in Telecommunications.

References

[1] AHN, G.-J., AND SHIN, D. Towards scalable authentication in health services. In *Enabling Technologies: Infrastructure for Collaborative Enterprises, 2002. WET ICE 2002. Proceedings. Eleventh IEEE International Workshops on* (2002), IEEE, pp. 83–88.

[2] ATALLAH, M. J., FRIKKEN, K. B., GOODRICH, M. T., AND TAMASSIA, R. Secure biometric authentication for weak computational devices. In *Financial Cryptography and Data Security*. Springer, 2005, pp. 357–371.

[3] BAGHERZANDI, A., JARECKI, S., SAXENA, N., AND LU, Y. Password-protected secret sharing. In *Proceedings of the 18th ACM conference on Computer and communications security* (2011), ACM, pp. 433–444.

[4] BALLARD, L., KAMARA, S., AND REITER, M. K. The practical subtleties of biometric key generation. In *Proceedings of 17th Conference on Security Symposium* (2008), pp. 61–74.

[5] 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/.

[6] BISHOP, M., AND V KLEIN, D. Improving system security via proactive password checking. *Computers & Security 14*, 3 (1995), 233–249.

[7] BOYEN, X., DODIS, Y., KATZ, J., OSTROVSKY, R., AND SMITH, A. Secure remote authentication using biometric data. In *Advances in Cryptology–EUROCRYPT 2005*. Springer, 2005, pp. 147–163.

[8] BOYKO, V., MACKENZIE, P., AND PATEL, S. Provably secure password-authenticated key exchange using diffie-hellman. In *Advances in Cryptology-Eurocrypt 2000* (2000), Springer, pp. 156–171.

[9] BRAINARD, J., JUELS, A., KALISKI, B., AND SZYDLO, M. Nightingale: A new two-server approach for authentication with short secrets. In *12th USENIX Security Symp* (2003), Citeseer.

[10] CAMENISCH, J., CASATI, N., GROSS, T., AND SHOUP, V. Credential authenticated identification and key exchange. In *Advances in Cryptology–CRYPTO 2010*. Springer, 2010, pp. 255–276.

[11] CHAI, Z., CAO, Z., AND LU, R. Threshold password authentication against guessing attacks in ad hoc networks. *Ad Hoc Networks 5*, 7 (2007), 1046 – 1054.

[12] CHIEN, H.-Y., JAN, J.-K., AND TSENG, Y.-M. A modified remote login authentication scheme based on geometric approach. *Journal of Systems and Software 55*, 3 (2001), 287–290.

[13] CHIEN, H.-Y., JAN, J.-K., AND TSENG, Y.-M. An efficient and practical solution to remote authentication: smart card. *Computers & Security 21*, 4 (2002), 372–375.

[14] CLAIR, L. S., JOHANSEN, L., ENCK, W., PIRRETTI, M., TRAYNOR, P., MCDANIEL, P., AND JAEGER, T. Password exhaustion: Predicting the end of password usefulness. In *Information Systems Security*. Springer, 2006, pp. 37–55.

[15] Forecast: Devices by Operating System and User Type, Worldwide, 2010-2017, 3Q13 Update. http://www.gartner.com/resId=2596420.

[16] DAVIS, D., MONROSE, F., AND REITER, M. K. On user choice in graphical password schemes. In *13th USENIX Security Symposium* (2004), vol. 9.

[17] DEO, V., SEIDENSTICKER, R. B., AND SIMON, D. R. Authentication system and method for smart card transactions, Feb. 24 1998. US Patent 5,721,781.

[18] DHAMIJA, R., AND PERRIG, A. Deja vu: A user study using images for authentication. In *Proceedings of the 9th USENIX Security Symposium* (2000), Usenix Denver, CO, pp. 45–58.

[19] DI CRESCENZO, G., LIPTON, R., AND WALFISH, S. Perfectly secure password protocols in the bounded retrieval model. *Theory of Cryptography* (2006), 225–244.

[20] DI PIETRO, R., ME, G., AND STRANGIO, M. A. A two-factor mobile authentication scheme for secure financial transactions. In *Mobile Business, 2005. ICMB 2005. International Conference on* (2005), IEEE, pp. 28–34.

[21] A Complete Suite of ElcomSoft Password Recovery Tools. http://www.elcomsoft.com/eprb.html#gpu.

[22] ERKIN, Z., FRANZ, M., GUAJARDO, J., KATZENBEISSER, S., LAGENDIJK, I., AND TOFT, T. Privacy-preserving face recognition. In *Privacy Enhancing Technologies* (2009), Springer, pp. 235–253.

[23] FLORENCIO, D., AND HERLEY, C. A large-scale study of web password habits. In *Proceedings of the 16th international conference on World Wide Web* (2007), ACM, pp. 657–666.

[24] GONG, L. Optimal authentication protocols resistant to password guessing attacks. In *Computer Security Foundations Workshop, 1995. Proceedings., Eighth IEEE* (1995), IEEE, pp. 24–29.

[25] GONG, L., LOMAS, M. A., NEEDHAM, R. M., AND SALTZER, J. H. Protecting poorly chosen secrets from guessing attacks. *Selected Areas in Communications, IEEE Journal on 11*, 5 (1993), 648–656.

[26] GWOBOA, H. Password authentication without using a password table. *Information Processing Letters 55*, 5 (1995), 247–250.

[27] HALDERMAN, J. A., SCHOEN, S. D., HENINGER, N., CLARKSON, W., PAUL, W., CALANDRINO, J. A., FELDMAN, A. J., APPELBAUM, J., AND FELTEN, E. W. Lest we remember: cold-boot attacks on encryption keys. *Communications of the ACM 52*, 5 (2009), 91–98.

[28] HALDERMAN, J. A., WATERS, B., AND FELTEN, E. W. A convenient method for securely managing passwords. In *Proceedings of the 14th international conference on World Wide Web* (2005), ACM, pp. 471–479.

[29] INSTITUTE, G. T. R. Teraflop Troubles: The Power of Graphics Processing Units May Threaten the Worlds Password Security System. http://www.gtri.gatech.edu/casestudy/Teraflop-Troubles-Power-Graphics-Processing-Units-GPUs-Password-Security-System.

[30] JABLON, D. P. Strong password-only authenticated key exchange. *ACM SIGCOMM Computer Communication Review 26*, 5 (1996), 5–26.

[31] JABLON, D. P. Extended password key exchange protocols immune to dictionary attack. In *Enabling Technologies: Infrastructure for Collaborative Enterprises, 1997., Proceedings Sixth IEEE workshops on* (1997), IEEE, pp. 248–255.

[32] JAN, J.-K., AND CHEN, Y.-Y. paramita wisdom password authentication scheme without verification tables. *Journal of Systems and Software 42*, 1 (1998), 45–57.

[33] JUELS, A., AND RIVEST, R. L. Honeywords: making password-cracking detectable. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security* (2013), ACM, pp. 145–160.

[34] JUELS, A., AND SUDAN, M. A fuzzy vault scheme. *Designs, Codes and Cryptography 38*, 2 (2006), 237–257.

[35] KATZ, J., MACKENZIE, P., TABAN, G., AND GLIGOR, V. Two-server password-only authenticated key exchange. In *Applied Cryptography and Network Security* (2005), Springer, pp. 1–16.

[36] KATZ, J., OSTROVSKY, R., AND YUNG, M. Efficient password-authenticated key exchange using human-memorable passwords. In *Advances in CryptologyEUROCRYPT 2001*. Springer, 2001, pp. 475–494.

[37] KATZ, J., OSTROVSKY, R., AND YUNG, M. Forward secrecy in password-only key exchange protocols. In *Security in Communication Networks*. Springer, 2003, pp. 29–44.

[38] KELSEY, J., SCHNEIER, B., HALL, C., AND WAGNER, D. Secure applications of low-entropy keys. In *Information Security*. Springer, 1998, pp. 121–134.

[39] KERSCHBAUM, F., ATALLAH, M. J., MRAÏHI, D., AND RICE, J. R. Private fingerprint verification without local storage. In *Biometric Authentication*. Springer, 2004, pp. 387–394.

[40] KLEIN, D. V. Foiling the cracker: A survey of, and improvements to, password security. In *Proceedings of the 2nd USENIX Security Workshop* (1990), pp. 5–14.

[41] KOMANDURI, S., SHAY, R., KELLEY, P. G., MAZUREK, M. L., BAUER, L., CHRISTIN, N., CRANOR, L. F., AND EGELMAN, S. Of passwords and people: measuring the effect of password-composition policies. In *Proceedings of the 2011 annual conference on Human factors in computing systems* (2011), ACM, pp. 2595–2604.

[42] KONTAXIS, G., ATHANASOPOULOS, E., PORTOKALIDIS, G., AND KEROMYTIS, A. D. Sauth: Protecting user accounts from password database leaks. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security* (New York, NY, USA, 2013), CCS '13, ACM, pp. 187–198.

[43] Hackers crack more than 60% of breached LinkedIn passwords. http://www.computerworld.com/s/article/9227869/Hackers_crack_more_than_60_of_breached_LinkedIn_passwords.

[44] LOMAS, T., GONG, L., SALTZER, J., AND NEEDHAMN, R. Reducing risks from poorly chosen keys. In *Proceedings of the twelfth ACM symposium on Operating systems principles* (New York, NY, USA, 1989), SOSP '89, ACM, pp. 14–18.

[45] LUCKS, S. Open key exchange: How to defeat dictionary attacks without encrypting public keys. In *Security Protocols* (1998), Springer, pp. 79–90.

[46] MARSHALL, B. K. 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.

[47] MCKEEN, F., ALEXANDROVICH, I., BERENZON, A., ROZAS, C. V., SHAFI, H., SHANBHOGUE, V., AND SAVAGAONKAR, U. R. Innovative instructions and software model for isolated execution. *HASP 13* (2013), 10.

[48] MIRANTE, D., AND CAPPOS, J. Understanding Password Database Compromises. Tech. Rep. TR–CSE–2013–02, NYU Poly, Sep 2013.

[49] MONROSE, F., REITER, M. K., LI, Q., AND WETZEL, S. Cryptographic key generation from voice. In *Security and Privacy, 2001. S&P 2001. Proceedings. 2001 IEEE Symposium on* (2001), IEEE, pp. 202–213.

[50] MONROSE, F., AND RUBIN, A. D. Keystroke dynamics as a biometric for authentication. *Future Generation Computer Systems 16*, 4 (2000), 351–359.

[51] MORRIS, R., AND THOMPSON, K. Password security: A case history. *Communications of the ACM 22*, 11 (1979), 594–597.

[52] Security Advisories – OAuth. http://oauth.net/advisories/.

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

[54] OSADCHY, M., PINKAS, B., JARROUS, A., AND MOSKOVICH, B. Scifi-a system for secure face identification. In *Security and Privacy (SP), 2010 IEEE Symposium on* (2010), IEEE, pp. 239–254.

[55] 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.

[56] Rainbow Table – Wikipedia. http://en.wikipedia.org/wiki/Rainbow_table.

[57] ROSS, B., JACKSON, C., MIYAKE, N., BONEH, D., AND MITCHELL, J. C. Stronger password authentication using browser extensions. In *Proceedings of the 14th Usenix Security Symposium* (2005), vol. 1998.

[58] SAE-BAE, N., AHMED, K., ISBISTER, K., AND MEMON, N. 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* (2012), ACM, pp. 977–986.

[59] SATHIK, M. M., AND SELVI, A. K. Secret sharing scheme for data encryption based on polynomial coefficient. In *Computing Communication and Networking Technologies (ICCCNT), 2010 International Conference on* (2010), IEEE, pp. 1–5.

[60] SCHECHTER, S., HERLEY, C., AND MITZENMACHER, M. 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* (2010), USENIX Association, pp. 1–8.

[61] SHAMIR, A. How to share a secret. *Communications of the ACM 22*, 11 (1979), 612–613.

[62] SHAY, R., KOMANDURI, S., KELLEY, P. G., LEON, P. G., MAZUREK, M. L., BAUER, L., CHRISTIN, N., AND CRANOR, L. F. Encountering stronger password requirements: user attitudes and behaviors. In *Proceedings of the Sixth Symposium on Usable Privacy and Security* (2010), ACM, p. 2.

[63] SHEN, F., MEI, C., JIANG, H., AND XU, Z. Towards secure and reliable data storage with multi-coefficient secret sharing. In *Computer and Information Technology (CIT), 2010 IEEE 10th International Conference on* (2010), IEEE, pp. 797–802.

[64] SHOUP, V. *On formal models for secure key exchange*. Citeseer, 1999.

[65] SNELICK, R., ULUDAG, U., MINK, A., INDOVINA, M., AND JAIN, A. Large-scale evaluation of multimodal biometric authentication using state-of-the-art systems. *Pattern Analysis and Machine Intelligence, IEEE Transactions on 27*, 3 (2005), 450–455.

[66] Sony PlayStation suffers massive data breach — Reuters. http://www.reuters.com/article/2011/04/26/us-sony-stoldendata-idUSTRE73P6WB20110426.

[67] STEINER, M., TSUDIK, G., AND WAIDNER, M. Refinement and extension of encrypted key exchange. *ACM SIGOPS Operating Systems Review 29*, 3 (1995), 22–30.

[68] SUN, S.-T., BOSHMAF, Y., HAWKEY, K., AND BEZNOSOV, K. A billion keys, but few locks: the crisis of web single sign-on. In *Proceedings of the 2010 workshop on New security paradigms* (2010), ACM, pp. 61–72.

[69] TOPKARA, U., ATALLAH, M. J., AND TOPKARA, M. Passwords decay, words endure: Secure and re-usable multiple password mnemonics. In *Proceedings of the 2007 ACM symposium on Applied computing* (2007), ACM, pp. 292–299.

[70] TSAI, C.-S., LEE, C.-C., AND HWANG, M.-S. Password authentication schemes: current status and key issues. *International Journal of Network Security 3*, 2 (2006), 101–115.

[71] TUYLS, P., AND GOSELING, J. Capacity and examples of template-protecting biometric authentication systems. In *Biometric Authentication*. Springer, 2004, pp. 158–170.

[72] Age of the Universe – Wikipedia, the free encyclopedia. https://en.wikipedia.org/wiki/Age_of_the_universe.

[73] WANG, P., KIM, Y., KHER, V., AND KWON, T. Strengthening password-based authentication protocols against online dictionary attacks. In *Applied Cryptography and Network Security* (2005), Springer, pp. 17–32.

[74] WHEELER, D. DropBox Tech Blog — zxcvbn: realistic password strength estimation. https://tech.dropbox.com/2012/04/zxcvbn-realistic-password-strength-estimation/.

[75] WHEELER, D. SLOCCount. *Available from http://www.dwheeler.com/sloccount*.

[76] WU, T., ET AL. The secure remote password protocol. In *Internet Society Symposium on Network and Distributed System Security* (1998).

[77] xkcd: Password Strength. http://xkcd.com/936/.

[78] YANG, W.-H., AND SHIEH, S.-P. Password authentication schemes with smart cards. *Computers & Security 18*, 8 (1999), 727–733.

[79] YEH, K.-H., SU, C., LO, N.-W., LI, Y., AND HUNG, Y.-X. Two robust remote user authentication protocols using smart cards. *Journal of Systems and Software 83*, 12 (2010), 2556–2565.

[80] ZONENBERG, A. Distributed hash cracker: A cross-platform gpu-accelerated password recovery system. *Rensselaer Polytechnic Institute* (2009), 27.