Multi-Factor Credential Hashing for Asymmetric Brute-Force Attack Resistance

Vivek Nair
UC Berkeley
vcn@berkeley.edu

Dawn Song
UC Berkeley
dawnsong@berkeley.edu

Abstract—**Since the introduction of bcrypt in 1999, adaptive password hashing functions, whereby brute-force resistance increases symmetrically with computational difficulty for legitimate users, have been our most powerful post-breach countermeasure against credential disclosure. Unfortunately, the relatively low tolerance of users to added latency places an upper bound on the deployment of this technique in most applications. In this paper, we present a multi-factor credential hashing function (MFCHF) that incorporates the additional entropy of multi-factor authentication into password hashes to provide asymmetric resistance to brute-force attacks. MFCHF provides full backward compatibility with existing authentication software (e.g., Google Authenticator) and hardware (e.g., YubiKeys), with support for common usability features like factor recovery. The result is a 10^6 to 10^{48} times increase in the difficulty of cracking hashed credentials, with little added latency or usability impact.**

1. Introduction

Despite unprecedented levels of information security spending [24], the frequency and severity of data breaches continue to experience exponential yearly growth [1]. While countless factors are at play, stolen or compromised credentials are both the largest individual *cause* of data breaches, accounting for over 80% of breaches on record [16], [37], and the largest *result* of data breaches, being present in nearly 80% of stolen databases [7], [15]. Thus, data breaches effectively form a negative feedback loop, whereby stolen credentials from one data breach are often used to compromise several further systems. Preventing the extraction of plaintext credentials from stolen data is an important objective toward breaking this cycle.

Salted hashing of stored passwords has long been our strongest tool for preventing the disclosure of credentials after a data breach has occurred. Unfortunately, the low complexity of typical passwords [10] and the extreme hash rate of modern ASICs [6] make password hashes stored using standard cryptographic hash functions highly susceptible to brute-force attacks. However, "adaptive" hash functions like bcrypt [32] provide a convenient solution to this problem by allowing their computational complexity to increase over time in accordance with improvements in overall computational power. While more hardware-resistant hash functions like Argon2 [5] have since been introduced, the paradigm of adding artificial computational difficulty as the primary means of strengthening the resistance of password hashes has remained relatively unchanged since the introduction of bcrypt in 1999.

Today, the use of adaptive password hashing has proven necessary but insufficient to prevent password disclosure following data breaches, as most users' low tolerance for added latency [3] has effectively placed an upper bound on the extent to which such techniques can be utilized. As a result, some recent data breaches have seen credential disclosure ratios of nearly 50% despite using strong adaptive password hashes (see §2.2.2).

Unsurprisingly, coinciding with this alarming trend in the rate of password disclosure has been the widespread adoption of multi-factor authentication (MFA), which has become nearly ubiquitous in high-security applications to combat the risk of credential stuffing. As it stands, MFA increases the total entropy used to authenticate a user, but does so through mechanisms entirely independent of the primary authentication method. The goal of this paper is simply to incorporate the added entropy of MFA into password hashes to significantly increase their resistance to brute-force attacks without any added latency for users.

In this paper, we present techniques for building a multi-factor credential hashing function (MFCHF) that uses entropy from common authentication factors like HMAC-Based One-Time Password (HOTP) [38], Time-Based One-Time Password (TOTP) [39], and Out-Of-Band Authentication (OOBA) to strengthen password hashes without modifying existing client-side authentication hardware or software. Doing so in practice is non-trivial, as the dynamic nature of OTP factors is not readily conducive to their incorporation in a static hash. We overcome these limitations through a combination of novel cryptographic construction and techniques adapted from the field of multi-factor key derivation. The result is a dramatic improvement in brute-force attack resistance, with our experiments showing an MFCHF hash based on a password and HOTP would take over 8.5 years to crack compared to just 4.5 minutes with Argon2 alone (§7.3).

Contributions.
1) We performed an empirical study to demonstrate the real-world impact of hash function design on credential disclosure across over 4,000 prior data breaches (§2.2).
2) We describe the first known method to use entropy from common authentication factors like HOTP, TOTP, OOBA, and YubiKeys within credential hashes (§4).
3) Our scheme supports common usability features like validation windows (§4.2.1), factor persistence (§5.1), and account recovery (§5.2) with no loss in security.
4) We demonstrate the merits of our approach through a full implementation (§6), and perform an experimental evaluation of its real brute-force attack resistance (§7).

2. Background & Motivation

Although over $150 billion is now spent annually on information security [24], data breaches continue to experience exponential growth. In 2021 alone, there were over 4,000 publicly disclosed breaches containing over 22 billion records [22]. Thus, while the majority of current security research focuses on data breach prevention, the occurrence of thousands of data breaches per year should still be considered somewhat inevitable in today's security landscape. With the average cost of each data breach at an all-time high of $4.35 million [16], minimizing risk in the event of a data breach is equally deserving of research attention. Indeed, the *presume breach* tenet of the widely accepted zero-trust security framework compels implementers to give due consideration to harm reduction in the event that a data breach does occur [36].

While data breaches may contain a plethora of sensitive and personally identifiable information, password disclosure is amongst the most ubiquitous risks due to the prevalence of password-based authentication in user-facing systems. Increasingly, the consequences of password disclosure extend not only to the directly affected systems, but also to unrelated third-party systems via credential stuffing due to widespread cross-site password reuse. Hindering the ability of adversaries to obtain user credentials from breached databases thus remains a significant focus of post-breach risk mitigation efforts.

2.1. Password Hashing

Password hashing is today considered the primary countermeasure to leaking passwords in the event of a data breach. Rather than storing passwords in plaintext, systems are configured to store a cryptographic hash of passwords corresponding to each user. Upon login, credentials presented by users are hashed and compared to the stored value. Since hash functions are non-reversible, the storage of password hashes avoids directly disclosing passwords in the event of a data breach.

Unfortunately, absent any further security considerations, hashed passwords are still highly susceptible to brute-force and dictionary attacks. While the average password contains just 40.54 bits of entropy [10], modern hardware allows adversaries to calculate trillions of hashes per second for popular functions like SHA256 [6], allowing password hashes to be reversed ("cracked") and the plaintext password to be revealed, often in a matter of seconds. Moreover, the deterministic nature of hash functions causes password reuse across users to be immediately evident even without cracking their hashes.

2.1.1. Salting. The practice of salting, whereby a randomly-generated "salt" value is used along with a password as input to a hash function and is then stored alongside the password, is considered best practice for all applications where password hashing is used. Its immediate consequence is adding a degree of non-determinism to the hashing process, such that even users with identical passwords will have different password hashes, thereby significantly slowing the process of cracking said hashes. In some instances, a further fixed random value, known as a "pepper," is also used as an input to the hash function and is stored separately from the primary database.

2.1.2. Brute-Force Resistance. Even with the use of a salt, most cryptographic hash functions are optimized for computational efficiency, allowing brute-force attacks to proceed with relative speed. By contrast, most purpose-built password hashing functions are designed with a degree of intentional computational inefficiency, which significantly slows the pace of brute-force attacks at the cost of a slightly longer time to verify login attempts. In so-called "adaptive" password hashing functions, a variable cost parameter can be used to tune the function's computational difficulty and increase it over time as computing power improves. Fig. 1 shows how changing the cost parameter of bcrypt [32], a popular adaptive password hashing function, affects the cracking time of an attacker and the verification time of a user.[1]
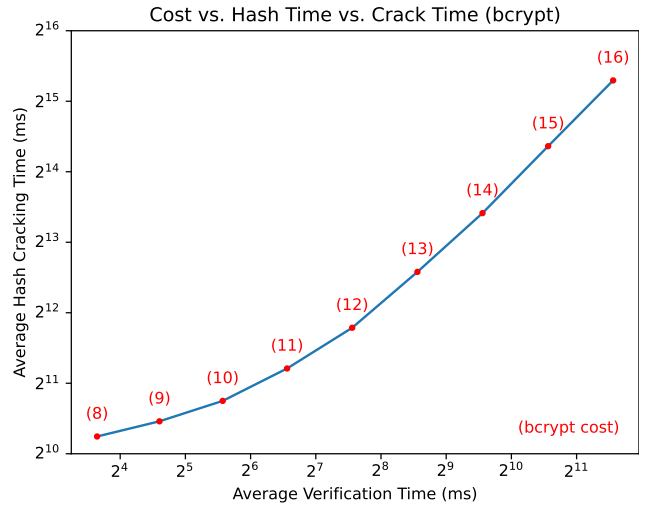

Figure 1: Effect of bcrypt cost parameter on hash cracking time of an adversary and verification time for a user.

As illustrated in Fig. 1, a roughly linear relationship exists between verification time and hash cracking time for adaptive password hashing functions. In this paper, we categorize this relationship as *symmetric resistance*, as increasing the difficulty of brute-force attacks necessarily accompanies a corresponding increase in the verification time for legitimate users. On the other hand, a technical improvement that increased the difficulty of cracking password hashes without impacting the verification time of a legitimate user would provide *asymmetric resistance*.

A variety of password hashing mechanisms have been proposed with various approaches for providing symmetric resistance. In 2015, Argon2 [5] was selected as the winner of a competition to determine the best hash function for hardware-resistant password hashing.

2.2. Breach Statistics

While the choice of password hashing method theoretically plays an important role in preventing hash cracking after a data breach, how does the choice of hash function impact password disclosure in practice? We conducted a small measurement study to evaluate how password hashing has impacted the rate of password disclosure across

---

1. Simulated cracking 10 salted bcrypt hashes corresponding to the top 100 passwords at each cost parameter. The hardware used to produce this graph and all other benchmarks in this paper is described in §B.

thousands of real data breaches. To do so, we analyzed data from Hashmob, a "password research and recovery" community in which users around the world collaborate to crack hashes from over 4,000 verified data breaches comprising over 2.5 billion hashed credentials [13].
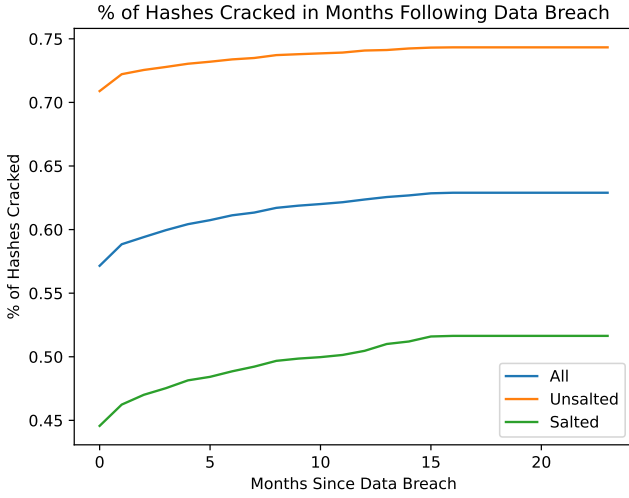

Figure 2: Progressive cracking of salted and unsalted hashes in the months following a data breach.

2.2.1. Password Cracking Time. Fig. 2 illustrates the progression of a typical brute-force attack over time using Hashmob monthly progress data. In an average data breach, 57% of password hashes are cracked within the first month, likely corresponding to those passwords which can easily be located via dictionary attacks. The rate of password cracking plateaus after the first year at around 74% of hashes cracked, with no cracking attempts being reported more than 18 months after a data breach.
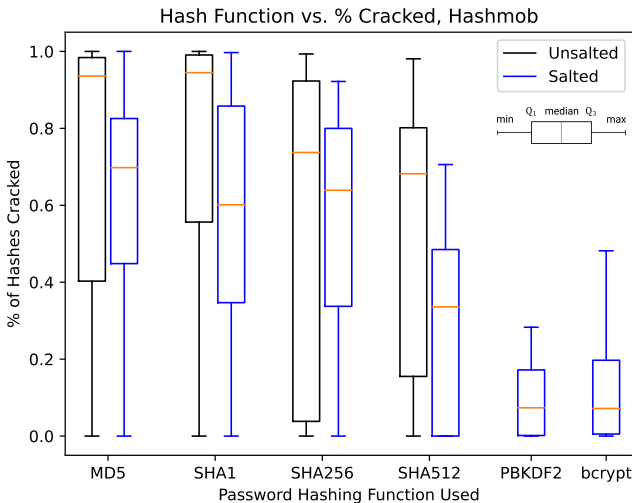

Figure 3: Portion of hashes eventually cracked for hash types seen in data breaches on Hashmob (box plot).

2.2.2. Password Disclosure Ratio. Having established that the vast majority of password-cracking activity occurs within the first year following a data breach, we analyzed all Hashmob data breaches posted at least one year ago to determine what effect the choice of hash function has on the percent of passwords eventually cracked by the

community. The results for the six most popular password hashing methods are shown in Fig. 3. The full results of our analysis of 4,259 data breaches are given in §C.

2.2.3. Key Findings. We present the empirical findings of this section to emphasize that research into brute-force resistant password hashing methods has value that is not just theoretical but also clearly observable in the password cracking trends across thousands of actual data breaches. In particular, the beneficial effect of salting is clearly evident, with 51.6% of salted hashes eventually being cracked, in comparison with 74.3% of unsalted hashes. Even more clear is the impact of adaptive password hashing functions, with less than 15% of bcrypt and PBKDF2 hashes being cracked even a full year after a data breach.

With an average per-record liability of $164 [16], reducing the likely rate of password disclosure can have a significant impact on the level of risk associated with a data breach. Overall, stronger hashing methods have the effect of increasing the average time between a data breach and credential disclosure, thereby giving companies more opportunity to detect and respond to the data breach, and of reducing the percentage of hashes eventually cracked, thereby potentially reducing the overall liability associated with the breach. Adaptive hash functions have clearly already had a massive positive impact in this regard.

Unfortunately, users are also highly sensitive to the latency of applications, with loading times of over 400 ms resulting in significantly decreased website traffic and conversion rate [3]. Thus, there is effectively an upper limit on the extent to which symmetric brute-force resistance can be deployed without affecting usability. We thus argue that methods for achieving asymmetric resistance should be a significant focus of research in this area.

2.3. Multi-Factor Authentication

The recent widespread adoption of multi-factor authentication (MFA) is owed in no small part to the incidence of password disclosure in data breaches, not because it currently serves to increase the difficulty of obtaining user passwords, but rather because it is largely implemented in response to the threat of credential stuffing attacks that result from leaked credentials.

While a variety of MFA mechanisms are currently in use, the most popular MFA methods in use today are one-time passwords (OTPs), such as HOTP and TOTP, and challenge-response mechanisms, such as HMAC-SHA1 (as used by YubiKeys). The effect of including these factors in the login process is basically to increase the overall entropy used to authenticate, as both the password and secondary factor are used together to verify the user. Below, we provide some background information on each of these schemes to clarify their integration with MFCHF.

2.3.1. HMAC-SHA1. HMAC-SHA1 challenge-response authentication is an instance of ISO/IEC 9798-2 2-Pass Unilateral Authentication via Cryptographic Check Function (CCF) [11], where the selected CCF is a Hash-based Message Authentication Code (HMAC) [19] and Secure Hash Algorithm 1 (SHA1) [9] is chosen as the underlying hash function. In a typical implementation, a client and server will share a 20-byte secret key. The

login process proceeds as shown below, where $\mathsf{HS1}(\mathsf{k}, \mathsf{m})$ denotes HMAC-SHA1 with key k and message m:

1) server $\to$ client: challenge $\in [0, 2^{160})$
2) client $\to$ server: response $= \mathsf{HS1}(\mathsf{key}, \mathsf{challenge})$
3) server: *accept* iff response $= \mathsf{HS1}(\mathsf{key}, \mathsf{challenge})$

Despite the deprecation of SHA-1 [26] due to a lack of collision resistance, HMAC-SHA1 remains a secure [4] and popular option for authentication due to its hardware-based support in products like YubiKey [40], and forms the basis of most HOTP and TOTP implementations.

2.3.2. HOTP. HMAC-based one-time password (HOTP) [38] is a 1-pass authentication mechanism that is typically based on HMAC-SHA1.[2] It replaces the challenge and response mechanism with a shared counter value that starts at 0 and increments upon each successful login:

1) client $\to$ server: OTP $= \mathsf{HS1}(\mathsf{key}, \mathsf{counter}) \% 10^6$
2) server: *accept* iff OTP $= \mathsf{HS1}(\mathsf{key}, \mathsf{counter}) \% 10^6$
3) client, server: increment counter

The elimination of a random challenge and the reduction of the response size to a small number of decimal digits (usually 6) has made HOTP a popular choice for smartphone-based 2FA apps like Google Authenticator.

2.3.3. TOTP. Time-based one-time password (TOTP) [39] is an extension of HOTP that replaces the shared counter with a coarse timestamp. As with HOTP, it is typically based on HMAC-SHA1. Given the current UNIX time T, an initial time T_0, and a time interval T_X, the TOTP code at time T is equal to $\mathsf{HOTP}_\mathsf{K}(\lfloor (T - T_0)/T_X \rfloor)$. TOTP has the advantage of avoiding the counter desynchronization issues of HOTP.

HOTP, TOTP, and HMAC-SHA1 all essentially serve to supplement passwords with additional entropy derived from a key using HMAC, increasing the overall entropy used to authenticate a user. The goal of this paper is simply to take advantage of this added entropy in the password hashing process to increase the brute-force difficulty of the resulting hash while, assuming MFA was already in use, having no significant impact on the user experience, thus providing asymmetric brute-force resistance. Achieving this would require popular OTP authentication methods to be incorporated into the hashing mechanism without modifying the client-side functionality of these factors, such that the user experience remains largely unaffected.

Unfortunately, two major difficulties remain in the realization of this technique. Firstly, while passwords, and thus password hashes, remain fairly constant over time, OTPs are by definition intended for one-time use, and are thus expected to change upon each login. It may not be immediately clear how a static hash can be guaranteed to reflect the OTP corresponding to any given login request.

Secondly, while the server must retain the ability to validate all authentication factors, it can no longer centrally store secret information about those factors. For example, HOTP and TOTP codes are typically verified by storing a shared HMAC secret key in the database

along with a password hash, but doing so would defeat the purpose of using said OTP as part of the hash, as an adversary obtaining a copy of the database could easily reproduce the correct OTP and defeat any added difficulty.

Thus, while taking advantage of multi-factor authentication to increase the difficulty of cracking password hashes seems straightforward, doing so in practice is easier said than done and requires the design of new techniques.

2.4. MFKDF

The Multi-Factor Key Derivation Function (MFKDF) [25] is a recent improvement over password-based key derivation that incorporates multiple authentication factors into the key derivation process. Its construction provides an important building block for the creation of a multi-factor credential hashing function with support for commonly-used OTP authentication factors.

The MFKDF specification contains two major architectural components. The first component is the set of so-called "factor constructions," which convert a dynamic *factor witness*[3] (W) and public parameters (α) into static key material (σ). The public parameters require no security assumptions and can safely be stored in a database without concern for revealing information about the factors to potential adversaries. For some factors, these parameters must be updated upon each login ($\alpha_i \mapsto \alpha_{i+1}$). Constructions are given for a variety of popular authentication factors, including TOTP, HOTP, OOBA (e.g., Email/SMS), and HMAC-SHA1 (e.g., YubiKey).

The second major component of the MFKDF specification is the key derivation function itself, which adds a secret sharing layer to provide functionality such as threshold-based key derivation, advanced policy enforcement, and factor recovery. While useful in the key derivation setting, such functionality is not particularly relevant for building a multi-factor credential hashing function and adds unnecessary overhead in the form of needing to store excessive material like secret shares.

While the intended use case of MFKDF, namely client-side key derivation for end-to-end encryption, is very different from the goals of this paper, the MFKDF factor constructions are a key tool for solving the challenge of using dynamic OTP factors as an input to a static hash. The core technique of this paper builds atop said factor constructions to achieve multi-factor credential hashing with asymmetric resistance.

2.5. Summary

With the introduction of ultra-fast hashing ASICs for cryptocurrency mining, research into brute-force-resistant password hashing mechanisms has become more important than ever before. We hope the brief empirical study presented thus far serves to demonstrate that advancements in this area have a dramatic effect on the consequences of actual data breaches.

Clearly, the symmetric resistance provided by adaptive password hashing functions has already had a significant impact on password cracking, but the relative intolerance

---

2. HOTP can be constructed using other underlying hash functions, but popular implementations like Google Authenticator only support SHA-1. Per the HOTP specification, a 31-bit truncation function is applied to the HMAC output before determining the final OTP value.

3. The *witness* refers to the message used to authenticate (e.g., a 6-digit OTP), which is often not the same as the underlying shared secret.

of users to added latency places an upper limit on the potential use of these functions. Simultaneously, the adoption of multi-factor authentication has provided a key opportunity to incorporate additional entropy into credential hashes without increasing the computation time for legitimate users. We are thus motivated to explore a scheme that takes advantage of the additional entropy provided by MFA to provide asymmetric resistance to brute-force attacks by incorporating multiple authentication factors, rather than just passwords, into a single hash.

While achieving such a scheme has long seemed out of reach, the introduction of MFKDF has provided a blueprint for realizing its implementation. In the following section, we will outline the desired security properties of our multi-factor credential hashing function, and will then proceed to describe a scheme satisfying these properties.

3. Problem Statement

In a typical password hashing deployment, a user registers an account with a server by providing a password, a hash of which is stored by the server in a database. Later, the user initiates a login process with a server by providing the password, which the server hashes and compares to the stored hash to authenticate the request. Secondary authentication factors are then independently verified.

In this paper, we'll present a multi-factor credential hashing approach that differs from typical password hashing by atomically verifying all of a user's credentials with a single hash. The purpose of this section is to present the problem setting and goals of this approach. These are largely the same as in standard password hashing, except that all factors are provided and verified simultaneously rather than sequentially, yielding a significant asymmetric improvement in brute-force attack resistance.

3.1. Deployment Setting

Our deployment setting consists of the following entities:
- One or more *users*, each possessing a password along with a secondary authentication factor.
- A *server*, which stores data about the factors of each user and uses it to authenticate user login requests.

3.2. Registration Process

In an initial setup process, a user establishes a password with the server. The server establishes a secondary authentication factor and internally stores a hash relating to these factors that can be used to later authenticate the user. Thus, the MFCHF SETUP function requires the following type definition:

$$\text{SETUP} : \mathsf{password} \mapsto \mathsf{hash}, \mathsf{mfainfo}$$

3.3. Login Process

During a login process, a user simultaneously provides the server with witnesses corresponding to all authentication factors. For example, they may provide a password along with a TOTP code (e.g., from Google Authenticator). The server must then use the data stored

during the registration process to validate all factors and authenticate the user. If authentication succeeds, the server may also update its stored hash to prepare for the next login. Thus, the type definition for the MFCHF VERIFY function is as follows:

$$\text{VERIFY} : \mathsf{password}, \mathsf{witness}, \mathsf{hash} \mapsto reject \text{ or}$$
$$\mapsto accept, \mathsf{hash}$$

3.4. Threat Model

We consider security under a *total data breach* threat model; that is, at some instant, an adversary receives a snapshot of all data stored on the server. The goal of the adversary is to use this data to obtain the underlying credentials of the user, via brute force, dictionary attack, or any other method available to the adversary in an offline capacity. To ensure a fair comparison with other password hashing schemes, the adversary is considered successful even if just the user's password is determined.

If given unlimited time, the adversary will be able to brute-force user credentials under any scheme where the stored data allows credentials to be verified. Therefore, the goal of our scheme is to increase the time and difficulty of attacking user credentials without increasing the time taken to verify a legitimate user (asymmetric resistance).

3.5. Security Goals

To summarize, a multi-factor credential hashing function consists of separate SETUP and VERIFY functions. The security goals of an MFCHF scheme are as follows:

1) **Correctness:** When provided with valid witnesses corresponding to a user's established authentication factors and hash, the server outputs *accept* with $p = 1$.

2) **Safety:** When at least one of a user's factor witnesses is invalid with respect to the established factors and hash, the server outputs *reject* except with $p = negl$.

3) **Asymmetric Resistance:** For a given fixed verification time, a multi-factor hash should be significantly more difficult to brute-force than a standard password hash.

4. Multi-Factor Credential Hashing

We now present four practical constructions for two-factor credential hashing schemes corresponding to authentication using a password and either HOTP, TOTP, OOBA, or HMAC-SHA1 (YubiKey) as a secondary factor. While these schemes can easily be extended to arbitrary n-factor variants, their construction is most straightforwardly presented using two factors, which is the most common use case. Furthermore, our four chosen secondary factors are by no means exhaustive with respect to the authentication methods that can be used with multi-factor credential hashing, and exist to serve as a template for implementing the proposed approach with other authentication methods one may wish to use in the future while simultaneously demonstrating the backward compatibility of MFCHF with several popular unmodified authentication factors.

4.1. General Blueprint

We begin with an overview of our general approach for multi-factor credential hashing. Fig. 5 shows a typical password hashing setup using a salted adaptive hash function. Even if MFA is in use, secondary factors are considered totally independent of password validation.
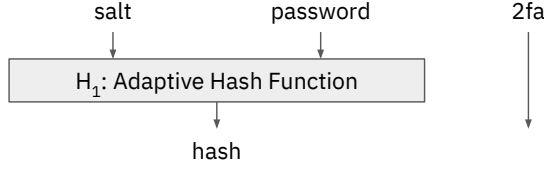

Figure 5: Standard adaptive password hashing scheme.

As discussed in §2.3, our goal is to incorporate the additional entropy of a secondary authentication factor into the hash function as though it were an additional hidden salt value, but we are unable to do so directly due to the dynamic nature of OTPs. A successful multi-factor credential hashing scheme must convert a dynamic secondary factor into another static input, and must further obscure the private material underlying that factor.
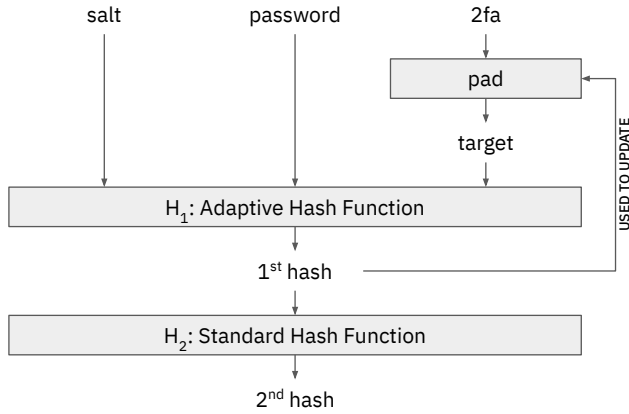

Figure 6: Blueprint for multi-factor credential hashing.

Fig. 6 illustrates the general strategy implemented by all of our MFCHF constructions. It incorporates three major insights: First, a one-time pad or modular offset can be used to convert dynamic OTPs into a static hash input without leaking information. Second, an intermediate hash value can be used as a key to hide secrets within the data stored on the server. Finally, those secrets can be recovered ephemerally upon each login to update the pad or offset.

The resulting feedback loop incorporates dynamic secondary factors into the hash without weakening them, and actually strengthens the underlying construction by not storing shared keys in plaintext. The forthcoming MFCHF constructions all start with this general blueprint and then introduce their own factor-specific optimizations.

4.2. MFCHF with HOTP (mfchf-hotp6)

Our first specific MFCHF construction uses HOTP, a factor that is illustrative of the general approach described in §4.1. The mfchf-hotp6 construction requires two underlying hash functions: H_1 is an adaptive password hash function such as Argon2, and H_2 is a standard cryptographic hash function such as SHA-256. We assume that a salt and HOTP key have been randomly chosen ahead of time. $HOTP(k, n)$ denotes the nth HOTP code under key k per RFC 4226 [38]. Throughout this paper, $\oplus$ denotes bitwise XOR and $\odot$ denotes concatenation. The SETUP function for mfchf-hotp6 then proceeds as follows:

1) Select a random target value in $[0, 10^6)$
2) Determine the first HOTP code: $first = HOTP(key, 1)$
3) Find the modular offset: $diff = (target - first) \% 10^6$
4) Compute 1st hash: $inner = H_1(password \odot target \odot salt)$
5) Blind HOTP key using 1st hash: $blind = key \oplus inner$
6) Compute 2nd hash: $outer = H_2(inner)$
7) Store $\{counter = 1, diff, blind, salt, outer\}$

Upon login, the user obtains an otp code from their HOTP application, which is provided to the server and used along with their password within the VERIFY function like so:

1) Recover target value: $target = (diff + otp) \% 10^6$
2) Compute 1st hash: $inner = H_1(password \odot target \odot salt)$
3) Output *reject* if $H_2(inner) \neq outer$, else continue
4) Increment counter value
5) Unblind HOTP key using 1st hash: $key = blind \oplus inner$
6) Determine the next OTP: $next = HOTP(key, counter)$
7) Find the modular offset: $diff = (target - next) \% 10^6$
8) Output *accept*, store $\{counter, diff, blind, salt, outer\}$

Because the original target value is recovered only if the otp value is correct, the correct password and OTP must be provided in order for the server to output *accept*. Thus, the MFCHF hash has the effect of simultaneously validating the password and HOTP factors. See Alg. 2 of §E for a pseudocode implementation of this method.

4.2.1. HOTP Validation Window.
A common usability feature in HOTP-based systems is the use of a "validation window" to recover from counter desynchronization. This functionality can easily be added onto the above HOTP construction by storing multiple offset (diff) values corresponding to multiple acceptable counter values, and iteratively trying each of the stored offsets upon login. For example, if a validation window of size 2 is desired, the offsets corresponding to ctr and ctr + 1 are stored at all times. Because each offset must be used independently, including additional offsets does not reduce the security of the scheme, and instead simply increases the verification time if multiple attempts are required. For instance, even if one stored 10^6 offsets, trying each of them would be no faster than trying all 10^6 possible target values.
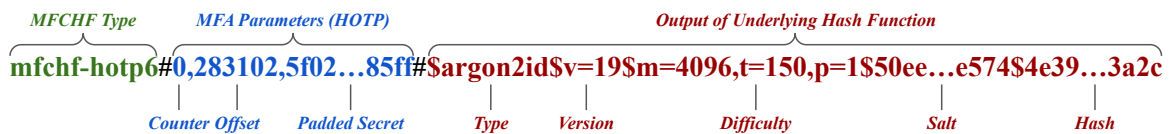

Figure 4: Example hash output generated by MFCHF with password and HOTP factors (mfchf-hotp6).

4.3. MFCHF with TOTP (mfchf-totp6)

Per RFC 6239, $\mathsf{TOTP}_K = \mathsf{HOTP}_K(\lfloor (T - T_0)/T_X \rfloor)$ where T is the current UNIX time, T_0 is the initial time, and T_X is the time interval [39]. Accordingly, the above MFCHF construction for HOTP can be modified to produce a suitable construction for TOTP by storing an array of offset (diff) values corresponding to the next w OTPs. Because the underlying HMAC key is available to the server in plaintext within the SETUP and VERIFY functions, this calculation can still be performed locally with no required modifications to the TOTP application. A full description of mfchf-totp6 is given in Alg. 3 of §E.

4.4. MFCHF with OOBA (mfchf-ooba6)

We next turn our attention to out-of-band authentication (OOBA) factors such as email and SMS. We assume the use of public key cryptography underlying each OOBA channel, which can be used to deliver an OTP only to an intended recipient. Let Enc represent encryption under a public-key encryption scheme (e.g., RSA) and pk represent the public key of the OOBA channel. As before, we assume that a salt has been randomly chosen ahead of time, but only a single password hash function (H) is required. The SETUP function is thus as follows:

1) Select a random first OTP and target value in $[0, 10^6)$
2) Encrypt the first OTP with pk: $ct = \mathsf{Enc}(\mathsf{first}, \mathsf{pk})$
3) Find the modular offset: $\mathsf{diff} = (\mathsf{target} - \mathsf{first}) \% 10^6$
4) Compute the hash: $\mathsf{hash} = \mathsf{H}(\mathsf{password} \odot \mathsf{target} \odot \mathsf{salt})$
5) Store $\{\mathsf{ct}, \mathsf{pk}, \mathsf{diff}, \mathsf{salt}, \mathsf{hash}\}$

Because our OOBA implementation is not limited by backward compatibility with existing HOTP/TOTP software, it is not restricted to using numeric OTPs. Instead, base 36 (or even 62) representation can be used to provide alphanumeric OTPs. Upon login, the server can forward ct to the OOBA channel. The user then provides the received otp to the server along with their password, which are used within the VERIFY function like so:

1) Recover target value: $\mathsf{target} = (\mathsf{diff} + \mathsf{otp}) \% 10^6$
2) Output $reject$ if $\mathsf{H}(\mathsf{password} \odot \mathsf{target} \odot \mathsf{salt}) \neq \mathsf{hash}$
3) Select a random next OTP value in $[0, 10^6)$
4) Encrypt the next OTP with pk: $ct = \mathsf{Enc}(\mathsf{next}, \mathsf{pk})$
5) Find the modular offset: $\mathsf{diff} = (\mathsf{target} - \mathsf{next}) \% 10^6$
6) Output $accept$, store $\{\mathsf{ct}, \mathsf{pk}, \mathsf{diff}, \mathsf{salt}, \mathsf{hash}\}$

Again, the server accepts the authentication request only if the user provides the correct otp, indicating that they had access to the OOBA channel. As in MFKDF, the recommended implementation of the OOBA factor for email authentication is to use the S/MIME key [33] of the recipient as pk. This can be extended to SMS authentication using the email-to-SMS gateway service [34] of each carrier. Alg. 4 of §E further describes mfchf-hotp6.

4.5. MFCHF with YubiKey (mfchf-hsha1)

Finally, we provide an MFCHF construction for hardware-based MFA devices such as smart cards and USB security keys. Amongst the most common protocols supported by these devices are FIDO U2F [2] and ISO 9798 2-Pass Unilateral Authentication over HMAC-SHA1 [11]. Unfortunately, the former is effectively impossible to incorporate into a multi-factor credential hash due to the inclusion of a client-side random nonce in all signatures. However, HMAC-SHA1 is both well supported (including all YubiKey devices [40]) and relatively straightforward to implement as an MFCHF factor. Our method requires a single password hash function (H) and HMAC-SHA1 (HS1). Assuming a key and salt have already been chosen, the SETUP function for mfchf-hsha1 is as follows:

1) Select a random challenge in $[0, 2^{160})$
2) Find the response: $\mathsf{response} = \mathsf{HS1}(\mathsf{key}, \mathsf{challenge})$
3) Compute the hash: $\mathsf{hash} = \mathsf{H}(\mathsf{password} \odot \mathsf{key} \odot \mathsf{salt})$
4) Blind key using response: $\mathsf{blind} = \mathsf{key} \oplus \mathsf{response}$
5) Store $\{\mathsf{blind}, \mathsf{challenge}, \mathsf{salt}, \mathsf{hash}\}$

Upon login, the challenge can be sent to the user's hardware device to generate a response, which is used along with their password within the VERIFY function like so:

1) Unblind key using response: $\mathsf{key} = \mathsf{blind} \oplus \mathsf{response}$
2) Output $reject$ if $\mathsf{H}(\mathsf{password} \odot \mathsf{key} \odot \mathsf{salt}) \neq \mathsf{hash}$
3) Select a random challenge in $[0, 2^{160})$
4) Find the response: $\mathsf{response} = \mathsf{HS1}(\mathsf{key}, \mathsf{challenge})$
5) Blind key using response: $\mathsf{blind} = \mathsf{key} \oplus \mathsf{response}$
6) Output $accept$, store $\{\mathsf{blind}, \mathsf{challenge}, \mathsf{salt}, \mathsf{hash}\}$

The authentication request is accepted only if the user is in possession of the hardware MFA device containing the shared key and is thus able to produce the correct response. The full specification is given in Alg. 1 of §E.

4.6. Use of SHA-1

Because SHA-1 has been deprecated since 2011 due to its lack of collision resistance [26], it may be seen as a red flag to recommend its use in new cryptographic deployments. However, the security of HMAC is not dependent upon collision resistance, and HMAC-SHA1 has been proven to remain secure without it [4].

Moreover, our hand is forced by the exclusive use of SHA-1 in popular MFA methods. For instance, HMAC-SHA1 remains the only deterministic challenge-response mechanism supported by YubiKeys. Furthermore, while HOTP and TOTP support the use of other hash functions, Google Authenticator still only supports HMAC-SHA1.

4.7. Post-Breach Security

Each of the four MFCHF constructions introduced in this section has the primary goal of increasing the brute-force resistance of stored password hashes by leveraging the entropy of MFA within stored hashes. For example, the HOTP, TOTP, and OOBA variants require the attackers to search the entire space of $\{\mathsf{password} \odot \mathsf{target}\}$ instead of just $\{\mathsf{password}\}$. This aspect of MFCHF's security is evaluated in detail in §7. However, the MFCHF techniques presented herein have the additional advantage of not storing factor-specific secrets in plaintext. As a result, the underlying authentication factors have the potential to remain operational in the event of a data breach. For example, a typical HOTP/TOTP-based system would store the HMAC key in plaintext on the server to verify OTPs, thus leaking the key in the event of a data breach and allowing the attacker to bypass the HOTP/TOTP factor entirely. On the other hand, MFCHF allows the server to verify OTPs without storing the HMAC key in plaintext, thus potentially leaving the factor operational in the period between the occurrence of a data breach and its detection.

5. Authentication Features

A major focus of our discussion thus far has been on the backward compatibility of MFCHF with existing authentication hardware and software, so as to emphasize that MFCHF requires modifications to neither third-party applications nor learned user behaviors. Similarly, in §4.2.1, we demonstrated that HOTP validation windows, a commonly-implemented feature designed to improve usability, are fully compatible with the mfchf-hotp6 construction. In general, a significant goal of this paper is to reduce the friction of implementing MFCHF to minimize the drawbacks accompanying its significant security advantages and thereby satisfy the balance of considerations preceding its implementation.

To that end, there are two additional usability features that an authentication scheme must support to avoid a heavy usability penalty: factor persistence and factor recovery. The aim of this section is to illustrate that these features can easily be implemented in conjunction with the proposed MFCHF methods while not reducing the security or brute-force resistance of the underlying schemes. In the following section, these features are implemented together with the mfchf-hotp6 scheme to produce a proof-of-concept application that realizes the security benefits of MFCHF while retaining all popular usability features.

5.1. Factor Persistence

Factor persistence is a common usability feature that allows users to bypass multi-factor authentication when using a familiar trusted device. A standard method of implementing factor persistence is by storing a browser cookie containing a pseudorandom token value on devices the user indicates are trusted. Login requests containing the cookie with the correct value are only required to supply the primary authentication factor, bypassing any multi-factor authentication that may be in use. Because each factor is typically validated by an entirely independent mechanism, implementing factor persistence usually requires fairly limited additional server-side logic. However, when using MFCHF, implementing factor persistence in the usual manner would require separate storage of a password-only hash to facilitate primary authentication, thereby defeating the brute-force resistance of MFCHF.

Fortunately, each of the presented MFCHF constructions contains a built-in token that can be used to achieve factor persistence, namely the hmackey in the case of mfchf-hsha1, and the target value in the case of HOTP, TOTP, and OOBA. While these values are never stored in plaintext, they become temporarily available within each of the VERIFY functions upon login. Thus, when a user successfully authenticates using a new device, they can be prompted to bypass MFA on that device, storing their hmackey or target value as a cookie in the process. On subsequent logins, that value can be used together with their password to authenticate their request without necessitating the use of multi-factor authentication. Because only a multi-factor hash is stored, this solution allows the use of factor persistence with MFCHF with no loss in security or brute-force resistance as long as the trusted device is secure (which is the assumption fundamentally made by allowing a given device to bypass MFA).

5.2. Factor Recovery

Factor recovery is another usability feature that is vital to support, with nearly 80% of users requiring a password reset on a regular basis [20]. Typically, factor recovery is implemented by establishing a tertiary factor, such as a recovery code or email OOBA, specifically for the purpose of recovering a lost primary or secondary factor. For example, in a password plus HOTP scheme, the password and recovery code may be used together to recover a lost HOTP device. Once again, a naive implementation of this setup may involve storing a separate password hash, thereby defeating the security of MFCHF.

Here, we suggest an alternative solution that involves storing a separate MFCHF hash for each combination of factors an application wishes to support. We will specifically follow the example of password and HOTP authentication with a recovery code, but stress that any supported factor can be used for recovery via this method. Implementing HOTP device recovery in this scheme is easily achieved by storing a hash of the password and recovery code. Password recovery, however, is non-trivial to implement, and can be achieved as follows:

Setup. During the setup process, a password recovery hash is created by applying a password hash function to a recovery code concatenated with the target value of the primary mfchf-hotp6 authentication hash. The hotpsecret should be stored padded with the output of this recovery hash in addition to the primary hash. As with the primary hash, a standard cryptographic hash function like SHA256 should then be applied before storing the resulting value.

Verify. The counter and offset values from the primary mfchf-hotp6 hash should be synchronized with the recovery hashes upon each successful login to ensure the same HOTP factor used for primary authentication can be used to recover the account password if required.

Recovery. During an account recovery process, the HOTP code can be combined with the offset value to recover the target as usual, which in turn can be verified together with the supplied recovery code against the recovery hash. Because the hotpsecret is stored padded with the recovery hash, it can be found when the recovery hash is computed.

Reconstitution. Finally, the known hotpsecret and target values can be combined with a new password to create a new mfchf-hotp6 hash for primary authentication while leaving the HOTP factor and recovery hash intact.

The proposed method for factor recovery averts storing an independent hash for each authentication factor and instead only stores multi-factor hashes for allowable factor combinations. As such, it implements factor recovery via tertiary factors without reducing the security of the overall scheme. Specifically, it reduces the overall brute-force difficulty only to the weakest combination of factors allowed to authenticate, which is the strongest possible security a multi-factor credential hashing scheme can hope to achieve. This principle is further discussed in §6.3, which evaluates the overall brute-force entropy of an MFCHF demo application with three MFCHF hashes.

6. Proof-of-Concept Implementation

To demonstrate the immediate practical utility of MFCHF and provide a blueprint for its deployment, we implemented a fully-featured MFCHF JavaScript library, and produced a proof-of-concept application implementing MFCHF as a replacement for password hashing. The demo application is modeled as a template, built using a React.js frontend and serverless JavaScript backend, with full authentication functionality (registration, login, recovery, etc.) but no substantive application content.

6.1. MFCHF Deployment

In our proof-of-concept application, passwords are used as a primary authentication factor with HOTP as a secondary factor. Therefore, the primary hash used for authentication reflects the mfchf-hotp6 algorithm described in §4.2, with PBKDF2 used as the underlying password hash in this instance. We verified that the resulting application was fully backward-compatible with the latest versions of the Google Authenticator and Microsoft Authenticator applications from the Google Play store. As such, the use of MFCHF for hashing on the backend should have no discernible impact on users who already use HOTP, other than requiring all factors to be entered simultaneously rather than sequentially as seen in Fig. 7a.

6.2. Authentication Features

To emphasize the limited usability impact of MFCHF, we implemented a number of common usability and convenience features within the proof of concept app, demonstrating their practical compatibility with MFCHF. In particular, the application supports the use of a HOTP validation window (i.e., a look-ahead window) using the method of §4.2.1, allowing users to recover from the desynchronization of their HOTP device. Further, the application supports factor persistence per §5.1 as shown in Fig. 7b, allowing users to bypass MFA on trusted devices with no reduction in the brute-force resistance of stored MFCHF hashes if those trusted devices are secure.

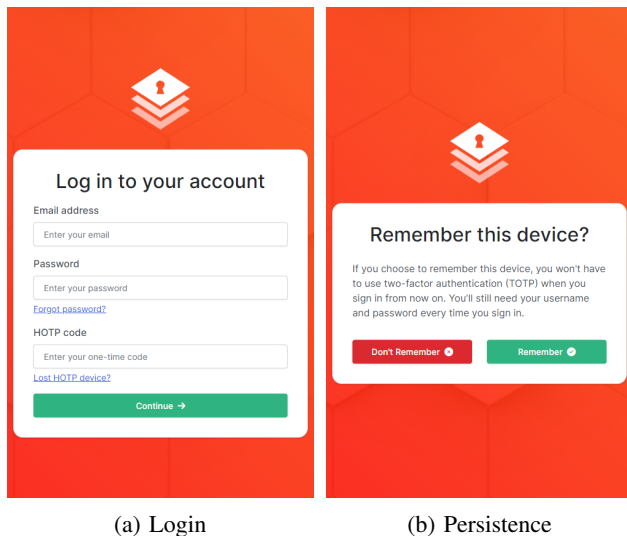(a) Login   (b) Persistence

Figure 7: Login screens from proof of concept application.

Finally, the demo application includes two secondary hashes for the purpose of factor recovery, as described in §5.2. Specifically, a HOTP + recovery code hash is stored for password recovery, and a password + recovery code hash is stored for HOTP recovery. Images depicting these features of the demo application are provided in §A.

6.3. Entropy & Security

To summarize, the demo application uses three distinct multi-factor credential hashes to achieve the desired functionality: a password + HOTP hash for standard logins, a HOTP + recovery code hash for password recovery, and a password + recovery code hash for HOTP recovery. All three of these MFCHF hashes greatly exceed the entropy of a standard password hash (≈ 40 bits [10]), as shown in Fig. 8. Therefore, the weakest link in this application is the password-plus-HOTP hash used for normal authentication, which is still 2^{20} (or about one million) times harder to brute-force attack than a standard password hash.
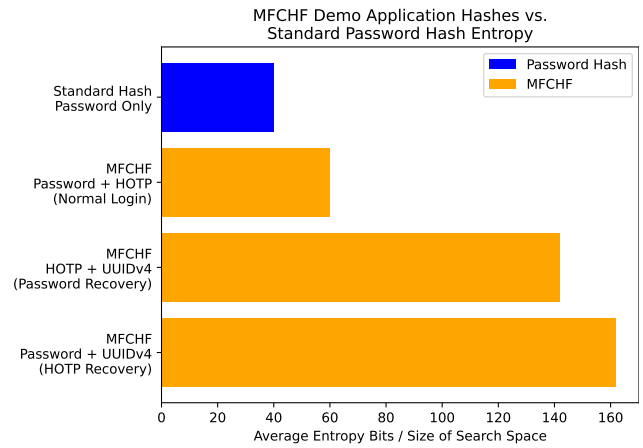

Figure 8: Average brute-force entropy of hashes used in MFCHF proof-of-concept application.

In addition to providing greater asymmetric resistance to brute-force attacks, the absence of HOTP secrets (or recovery codes) stored in plaintext anywhere in the application greatly increases the probability of these secondary factors remaining operational and secure in the interim period between the occurrence of a breach and its detection.

6.4. Summary

We present the fully-featured web application demo of this section to illustrate that MFCHF is concretely practical and suitable for real-world deployment. As described above, the scheme used in this proof of concept provides a 1,000,000x increase in asymmetric brute-force attack difficulty and increased post-breach security. Furthermore, these security advantages implicate very little impact on usability if MFA was already in use, with full backward compatibility with existing HOTP applications. As we have demonstrated, common usability features such as factor persistence, factor recovery, and HOTP validation windows are compatible with MFCHF without reducing its added security. What follows is a general evaluation of the performance and security of the proposed schemes.

7. Evaluation

Although the demo application of §6 abstractly addresses the practicality of MFCHF in a realistic setting, we further performed experiments to evaluate the performance of MFCHF in three important aspects. First, we benchmarked the computational and storage overhead of MFCHF over existing password hash functions to highlight its efficiency. Next, we evaluated the theoretical increase in brute-force search space provided by each MFCHF construction. Finally, we performed real brute-force attacks against a variety of schemes to demonstrate the security of MFCHF against a realistic adversary.

7.1. Performance

To benchmark the performance of MFCHF, we ran the JavaScript library used for the proof-of-concept application of §6 using Chrome v106.0.5249.103 on the same benchmarking machine used for all other experiments (see §B). The SETUP and VERIFY functions for mfchf-hotp6, mfchf-totp6, mfchf-ooba6, and mfchf-hsha1 were run 100 times each to measure the mean overhead of each function. To isolate the computational overhead of MFCHF from that of the underlying hashing mechanism, SHA256 was used as the hash function in all cases. For TOTP, a window of 2,920 was used. The results are shown in Fig. 9.
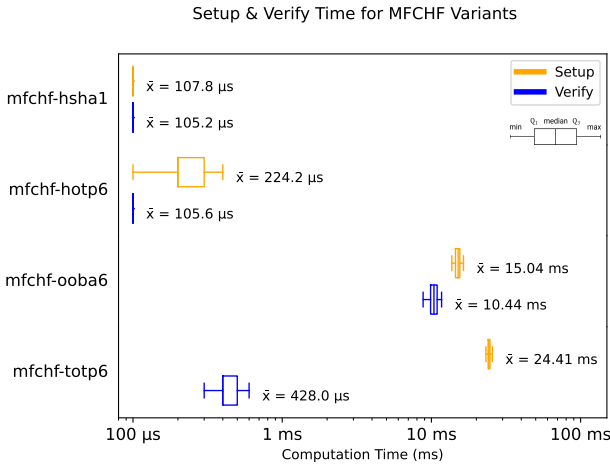

Figure 9: Overhead of MFCHF functions (box plot).

Another aspect worth evaluating is the space required to store each hash. A standard Argon2 hash requires about 77 bytes. By comparison, the space required increases to 131 bytes for mfchf-hotp6 and mfchf-hsha1 and 353 bytes for mfchf-ooba6. The absolute size of these hashes remains small enough to likely not pose a barrier to adoption, particularly in consideration of the fact that they replace the need to separately store an HMAC secret. However, the space required for mfchf-totp6 could vary from 7 to 219 kb depending on the size of window used.

7.2. Entropy

Next, we briefly evaluate the theoretical increase in input entropy (i.e., brute-force search space) of MFCHF over standard password hashes. For the HOTP and TOTP variants of MFCHF, adversaries must evaluate all 10^6 possible target values for each attempted password. Because

OOBA can use an alphanumeric OTP, adversaries must evaluate all 36^6 possible target values for each attempted password. Finally, for the HMAC-SHA1 (i.e., YubiKey) variant, the HMAC key is included in the hash, which can have 2^{160} possible values. Overall, the theoretical entropy gained by mfchf-hotp6, mfchf-totp6, mfchf-ooba6, and mfchf-hsha1 is 20, 20, 31, and 160 bits, respectively.

7.3. Brute-Force Resistance

While the theoretical increase in search space of MFCHF strongly indicates an improvement in brute-force difficulty, we also performed an experiment to validate the increased brute-force resistance of MFCHF. We first created 100 salted password hashes for each of several standard (MD5, SHA1, SHA256, and SHA512) and adaptive (PBKDF2, bcrypt, scrypt, and Argon2) hashing schemes. The hashes were chosen from a dictionary of the 10,000 most common passwords, and cost parameters for adaptive hashes were configured to take 200 ms to compute. Next, we used John the Ripper (for Argon2) or Hashcat (for all others) to crack each password via an exhaustive search of the dictionary. The hash and crack times for each hash type are shown in Fig. 10, with the full results in §D.
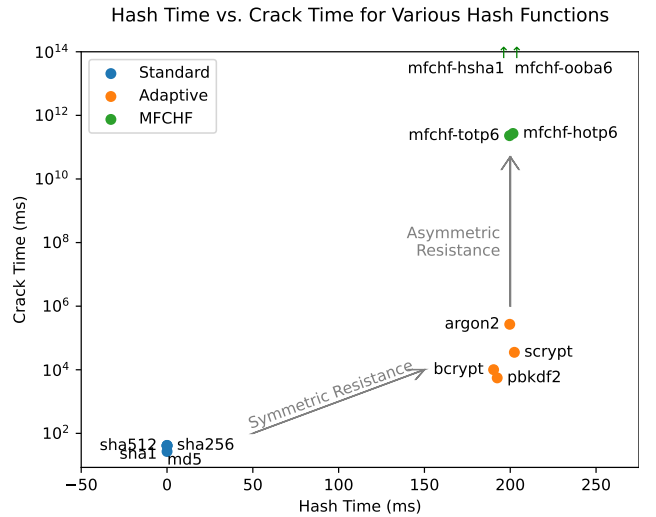

Figure 10: Hash and crack time for various hash types.

Next, we evaluated the brute force time of four types of MFCHF hashes (mfchf-hotp6, mfchf-totp6, mfchf-ooba6, and mfchf-hsha1) with Argon2id as the underlying hash function. Given that each of these hashes would take well over a year to crack using the above method, we instead let John the Ripper run for exactly 24 hours for each of the four hashes and examined the percentage of the search space exhausted in that time. We found that 0.016% and 0.019% of the search space were exhausted after 24 hours for mfchf-hotp6 and mfchf-totp6, respectively. Given that cracking a hash would require attempting 50% of the search space on average, we expect the average crack times for mfchf-hotp6 and mfchf-totp6 to be about 8.6 years and 7.2 years, respectively. For mfchf-ooba6 and mfchf-hsha1, John the Ripper reported 0.000% progress after 24 hours. Based on the relative entropy, we expect these hashes would take thousands of years to crack. These MFCHF results are also included in Fig. 10 and in §D.

7.4. Discussion

In §2.1, we introduced the notions of *symmetric resistance*, where brute-force attack difficulty increases proportionally with an increase in verification time, and *asymmetric resistance*, where brute-force attack difficulty is disproportionately improved relative to verification time for legitimate users. As demonstrated by our evaluation, MFCHF provides a robust improvement in asymmetric brute-force attack resistance: with the same 200 ms target as bcrypt, scrypt, PBKDF2, and Argon2, MFCHF provides a 10^6 to 10^{48} times increase in the difficulty of cracking hashed credentials. It does so by leveraging the entropy of passwords and multi-factor authentication together within a single multi-factor credential hash, rather than simply increasing the computational difficulty of the hash function, as is seen in most adaptive schemes.

Indeed, the practical effect of even the weakest MFCHF variants (HOTP/TOTP) far exceeds what one might intuitively expect given the seemingly small amount of added entropy (20 bits). While the average crack time for this setup without MFCHF was just 4.5 minutes, using MFCHF would increase the time to crack such a hash to over 7 years with no impact on the verification time for a legitimate user. This indicates that the current landscape of adaptive password hashing, with about 40 bits of entropy in the average password and 200 ms of latency tolerance, exists precisely at an optimality point for MFCHF to have a dramatic impact on the feasibility of a brute-force attack.

Interestingly, our empirical results do not represent a simple linear improvement in brute-force resistance relative to the added entropy. For example, most adaptive hash functions were susceptible to highly-parallel GPU-based attacks, significantly reducing their security in practice. Indeed, our chosen dictionary size of 10,000 was sufficiently large that almost all 10,496 GPU cores could be deployed in parallel to attack adaptive hashes (while still being small enough to feasibly evaluate within a short period of time). Increasing the dictionary size further could increase the attack difficulty across all hash types, but should not affect the relative advantage of MFCHF.

Finally, the computational overhead of some MFCHF variants required a reduction in underlying cost parameters to maintain a 200 ms latency target, which could reduce the added resistance in practice. For instance, while the overhead of all MFCHF functions was relatively small in relation to the 200 ms target, the higher overhead of mfchf-totp6 compared to mfchf-hotp6 resulted in mfchf-hotp6 providing more brute-force resistance than mfchf-totp6, despite both functions theoretically providing the same amount (20 bits) of additional entropy.

8. Security Analysis

Having now empirically demonstrated the asymmetric resistance of MFCHF, we next provide brief arguments for why the schemes of §4 satisfy the security properties of *correctness*, *safety*, and *asymmetric resistance*, as defined in §3.5. While a formal proof framework for multi-factor key derivation exists [18], [25], there is no suitable equivalent for multi-factor credential hashing. Our arguments are instead based on a semi-formal reduction to the security properties of the underlying hash functions.

8.1. Primitives

The MFCHF constructions of §4 use an adaptive password hash function such as Argon2 (H_1), and a standard cryptographic hash function such as SHA-256 (H_2). For both hash functions, the following properties are required:

- *Determinism* – For a given input value m, $H(m)$ must always generate the same hash value h with $p = 1$.
- *Total Pre-Image Resistance* – Given only an arbitrary hash value h, it is hard for an adversary to find any bit of input m such that $H(m) = h$ except with $p = 0.5 + negl$.
- *Second Pre-Image Resistance* – Given an input m_1, it is hard for an adversary to find another input m_2 such that $H(m_1) = H(m_2)$ and $m_1 \neq m_2$ except with $p = negl$.
- *Uniformly Pseudorandom* – Given an arbitrary m and $H(m) = h$, each bit of h is 1 with $p = 0.5$.
- *Strict Avalanche Criterion* – Given $H(m) = h$, changing one bit of m should change each bit of h with $p = 0.5$.

8.2. Correctness

The correctness property of MFCHF requires that when provided with valid witnesses corresponding to a user's established authentication factors and hash, the server outputs *accept* with $p = 1$. For a given login request, there is exactly one valid password and witness. In each MFCHF construction, the server outputs *accept* if $H(\text{password} \odot \text{target} \odot \text{salt})$ matches a stored hash of the correct values. It directly follows from the *determinism* property of H that if the correct password and target are provided, H will generate the same value h (and therefore, the server will output *accept*) with $p = 1$.

What remains to be shown is that target will match the expected value if the correct OTP is provided. In the case of the HMAC-SHA1 construction, this is true because bitwise XOR with one parameter fixed is an involution ($A \oplus B \oplus B = A$): key $\oplus$ response $\oplus$ response = key. For all other constructions, this instead holds because of the congruence relation $((A - B) \% N + B) \% N \equiv A$, which suggests $((A - B) \% N + B) \% N = A$ when $A \in [0, N)$. Thus, $((\text{target} - \text{otp}) \% 10^6 + \text{otp}) \% 10^6 = \text{target}$.

8.3. Safety

The safety property of MFCHF posits that when at least one of a user's factor witnesses is invalid, the server outputs *reject* except with $p = negl$. In each MFCHF construction, the server outputs *reject* if $H(\text{password} \odot \text{target} \odot \text{salt})$ does not match a stored hash of the correct values. If an adversary can violate the safety of MFCHF by finding an invalid password or target value that causes H to generate the same value h as the correct credentials (and therefore, the server to output *accept*), then they have also violated the *second pre-image resistance* of H by finding $m_1 \neq m_2$ such that $H(m_1) = H(m_2)$.

It remains to be shown that target will not match the expected value if the incorrect OTP is provided. To illustrate this, we simply invert the proofs given in §8.2: $A \oplus B \oplus B' \neq A$ if $B \neq B'$ (thus key is wrong if response is wrong), and $((A - B) \% N + B') \% N \neq A$ if $B \neq B'$ (thus target is wrong if otp is wrong).

8.4. Asymmetric Resistance

Finally, the asymmetric resistance property of MFCHF suggests that an MFCHF hash should be significantly harder to crack than an adaptive password hash with the same fixed verification time. Specifically, the brute-force difficulty is exponential with respect to the total entropy of the input space, so an MFCHF hash with a password and a 6-digit OTP should be 10^6 times harder to crack than a comparable password hash.

Given $\mathsf{H}(m) = h$, the *avalanche effect* of H suggests that if m is changed slightly (e.g, flipping a single bit), h changes significantly (e.g., half the bits flip). Now given $\mathsf{H}(\text{password} \odot \text{target} \odot \text{salt}) = h$, the asymmetric resistance of the MFCHF hash h derives from the *avalanche effect* of H, as any incorrect bit of $\{\text{password} \odot \text{target}\}$ will cause h to be completely incorrect. Furthermore, the *total pre-image resistance* of H precludes an adversary from reversing any part of h to determine which bits are incorrect. Given λ_1 bits of entropy in password and λ_2 bits of entropy in target, all $2^{\lambda_1 + \lambda_2}$ possible values must be exhaustively searched. The effect of adding a 6-digit ($\approx$ 20-bit) OTP is thus to increase the brute-force difficulty of h by a factor $\approx 2^{20}$ or 10^6 for the same verification time, resulting in asymmetric resistance.

Lastly, it remains to be argued that the values stored alongside the hash and salt in MFCHF do not weaken or reveal secret information about the underlying factors:

- For the blind value, this is the case because assuming the output of H is *uniformly pseudorandom*, it can be used as a one-time pad (Vernam cipher) to hide the value of key with information-theoretic security.
- For the diff value, this holds because of the security of modular arithmetic (rings); a pseudorandom otp can be used to hide target with information-theoretic security.
- For HMAC-SHA1, the challenge value is chosen uniformly randomly in $[0, 2^{160})$. If HMAC-SHA1 is secure, challenge does not reveal key or response.
- For OOBA, the values of ct and pk do not reveal otp if the public-key encryption method is IND-CCA secure.

9. Limitations

The first and most obvious limitation of MFCHF is that it requires the use of one of the supported multi-factor authentication methods. While MFA is increasingly widely supported by a variety of online services, the rate of voluntary adoption by end users has remained slow. Thus, it must be emphasized that our solution is chiefly targeted at services that already implement one of the supported authentication factors, and will still only benefit the subset of users who choose to enable MFA.

The above limitation is further evident in the recovery setup of §5.2, which requires three independent factors. It is already often the case that an online service supporting one MFA factor (e.g., TOTP) will also allow for a lost password to be recovered via another factor (e.g., Email OOBA). Still, the need to establish multiple factors for recovery may pose a barrier to the adoption of this method.

Another limitation is the need for a PKI in the OOBA variant. For email, S/MIME [33] is a widely accepted protocol that provides a PKI that can be used for this purpose and is supported by the majority of modern email

software. This can be extended to SMS via the email-to-SMS gateway services offered by most major phone carriers [34]. However, not all email providers support S/MIME, and not all phone carriers provide an email-to-SMS gateway. Therefore, OOBA is the only MFCHF variant that relies on features without universal support, and some users, such as those using legacy mail clients, will not be able to benefit from the OOBA construction.

With respect to the TOTP construction, one limitation is the need to choose a large enough window of offsets that a user does not lose access to their account if inactive for long periods. The need to calculate a large number of offset values is the reason TOTP has a higher setup overhead than all other MFCHF variants in §7.1. As a mitigating factor, we note that the calculation of offsets in the TOTP construction occurs after the *accept* or *reject* determination has already been made (see Alg. 3 in §E). Thus, a service could opt to update offset values asynchronously in the background after authenticating a user.

Finally, MFCHF fundamentally requires that all authentication factors be verified simultaneously, rather than being verified sequentially, as is often currently the case. While this provides a marked security improvement by preventing each factor from being individually attacked, it could prove frustrating for a user who fails to authenticate and is unsure which factor is incorrect. In the future, a method that statistically determines which factor is wrong could be used without losing much entropy in the process.

10. Failure Modes

Per our threat model (§3.4), the failure of an MFCHF hash occurs if an adversary is able to reverse the hash to obtain any of the underlying authentication factors. Our security analysis argues that the only ways to defeat an MFCHF hash are (1) to compromise the underlying factors or (2) to perform a brute-force attack. Still, there are several scenarios in which either may occur, even if MFCHF is correctly implemented with secure primitives.

Clearly, if a combination of factors with insufficient total entropy (such as a 6-digit OOBA factor and a 6-digit HOTP factor) is chosen, brute-force attacks against the entire input space may still be feasible. What is perhaps less obvious is that even if a single factor is weak, the entire hash may be attacked. For example, though an MFCHF hash combining a password and HOTP factor may itself be infeasible to crack, the compromise of the password factor may allow the remaining factor to be defeated by brute force. However, in such a scenario, the marginal value of compromising the second factor is limited, as HMAC secrets, unlike passwords, are rarely shared across accounts. Still, the use of password strength requirements and compromised credential checking remains essential.

Additionally, the "total data breach" threat model described in §3.4 does not consider the risk of an ongoing threat like undetected malware on the authentication server. In the scenario where the server remains actively compromised for long periods of time, authentication secrets can be stolen during the ephemeral period in which they are decrypted upon user login. While still an improvement over the typical method of storing HMAC, HOTP, and TOTP keys in plaintext, this constitutes another potential failure mode in which factors become compromised.

Finally, it is expected that a system implementing MFCHF is still only as secure as its underlying factors, and these factors must be properly managed and protected. For example, if an SMS device is vulnerable to a SIM-swapping attack, then an mfchf-ooba6 hash utilizing that device would be susceptible to compromise.

11. Related Work

Within the field of password hashing, there are no known works describing hash functions that incorporate common authentication factors like HOTP, TOTP, or YubiKey (HMAC-SHA1) into the hashing process so as to form a multi-factor credential hash. Instead, the vast majority of research into attack-resistant password hashing has focused on symmetric brute-force resistance, including works such as PBKDF2 [17], bcrypt [32], scrypt [28], yescrypt [29], and Argon2 [5] that provide various degrees of hardware resistance. Other works, such as Catena [12] and Lyra2 [35], emphasize resistance to side-channel attacks. Finally, delegable password hashing functions like Makwa [31] are designed to allow clients to outsource computation power for password hashing to untrusted third parties, thereby potentially increasing the computational power available to the client.

The relative resistance of password hashing functions to various hardware-enabled adversaries (e.g., CPU, GPU, or ASIC-based threats) has been the subject of benchmarking experiments [14]. However, we did not find any other studies surveying the effect of these functions on the actual rate of password disclosure across real data breaches.

With respect to multi-factor credential hashing, the "factor constructions" of MFKDF [25] were the main inspiration for the techniques of this paper. In MFKDF, these techniques are applied on the client side to derive a key for the purpose of end-to-end encryption, rather than being used on the server side for credential hashing. In particular, the authentication technique suggested by MFKDF requires the establishment of a shared key, which, unlike the techniques of this paper, would immediately fail to provide secure authentication after a data breach (although stored secrets would remain secure). Furthermore, the three-part scheme of MFKDF (SETUP, DERIVE, and UPDATE) are simplified into a more efficient two-part scheme (SETUP and VERIFY) in this work.

Prior to MFKDF, several works have proposed two-factor key derivation based specifically on a password and a YubiKey hardware token [8], [27]. Once again, these works focus on client-side deployment for an end-to-end encryption use case. Other than MFKDF, there are no known prior works utilizing HOTP or TOTP for key derivation, let alone credential hashing. While other works have addressed using these factors for the related problem of Multi-Factor Authenticated Key Exchange (MFAKE) [21], [23], [30], this approach aims to establish a shared key in the presence of man-in-the-middle adversaries and does not provide brute-force resistance or protection against a compromised server.

12. Future Work

12.1. Factors

Our focus in this paper was on those factors for which constructions were provided in MFKDF, namely static factors (e.g., passwords and recovery codes), HOTP, TOTP, HMAC-SHA1 (YubiKey), and out-of-band authentication (OOBA) factors such as SMS and email. While support for arbitrary factors based on MPC and trusted hardware are possible, they were not included in this paper due complicating the security model, but can be included in future work. With respect to TOTP, future work should emphasize reducing the size of data stored from the 219 kb currently required. Finally, future research should investigate the incorporation of factors not currently supported by MFKDF, such as biometrics, geolocation, device fingerprinting, behavioral authentication, and OIDC.

12.2. Features

Although the account recovery feature described in this paper provides sufficient functionality for the vast majority of systems, future work could take advantage of the threshold and policy-based MFKDF variants for use in systems with more advanced authentication policies. Furthermore, future iterations of MFCHF should ideally facilitate progressive deployment on systems that currently use password hashing without requiring all passwords to be reset. Lastly, future research could analyze the usability impact of MFCHF, specifically with regard to the requirement to provide and validate all authentication factors simultaneously. Mechanisms for mitigating this impact are also worth investigating, though this requirement may, in fact, be an inevitable trade-off of the MFCHF approach. Specifically, as mentioned in §9, we would love to see a feature that relinquishes a small amount of entropy to probabilistically "hint" the user which factor is wrong in the event that a login attempt fails to validate.

12.3. Applications

While we focus on the server-side use of MFCHF in the client-server setting, future work should explore the use of MFCHF in other applications, such as for user authentication within operating systems. Crypt, the predecessor to Bcrypt, was originally developed for password hashing in Unix, so as to enable password-based authentication without storing passwords in plaintext. While using authentication factors like HOTP within operating systems would typically not have been feasible due to the need to store an HMAC key, MFCHF can enable the secure use of such factors for operating system authentication. The potential use of MFCHF for authentication in networks and decentralized systems should also be investigated.

13. Conclusion

With both the volume and severity of major information security incidents continuing to experience exponential growth, password hashing has played an important role in reducing the rate of credential disclosure (and thus potential liability) resulting from a data breach. Indeed, our analysis of over 4,000 actual data breaches containing hashed credentials clearly demonstrates that the use of salted and adaptive password hashing has a significant effect on the rate of password disclosure not just in theory, but also in practice. Still, while the majority of research in this area has emphasized the use of adaptive hash functions for symmetric brute-force attack resistance, the low tolerance of users for added latency places an upper limit on the applicability of this technique in practice.

Coinciding with the rise in large-scale data breaches is the increased use of multi-factor authentication to combat the threat of credential stuffing. We thus set out to create a multi-factor credential hashing function that utilizes the additional entropy provided by multi-factor authentication to provide asymmetric resistance to brute-force attacks. Our scheme emphasizes usability by supporting common features like account recovery, desynchronization windows, and factor persistence, while maintaining compatibility with popular, unmodified authentication factors like HOTP, TOTP, and YubiKey that are already in use.

To demonstrate the practicality of MFCHF as a drop-in replacement for password hashing, we produced a full implementation of MFCHF using HOTP and evaluated its deployment within a typical full-stack web application. Our evaluation shows MFCHF to be dramatically harder to brute-force attack, while having a low computational overhead with negligible impact on setup and verification time for a legitimate user. Overall, in systems where MFA is already in use, MFCHF provides a compelling value proposition over traditional password hashing by improving brute-force resistance and post-breach security with limited impact on usability or performance.

Availability

Our GitHub repository, which contains the source code and data used to produce the results presented in this paper, is available here: https://github.com/mfchf/paper

Acknowledgments

References

[1] Accenture. Ninth Annual Cost Of Cybercrime Study. URL: https://www.accenture.com/_acnmedia/pdf-96/accenture-2019-cost-of-cybercrime-study-final.pdf.

[2] FIDO Alliance. Universal 2nd Factor (U2F) Overview.

[3] Ioannis Arapakis, Souneil Park, and Martin Pielot. Impact of Response Latency on User Behaviour in Mobile Web Search. In *Proceedings of the 2021 Conference on Human Information Interaction and Retrieval*, pages 279–283, March 2021. arXiv:2101.09086 [cs]. URL: http://arxiv.org/abs/2101.09086, doi:10.1145/3406522.3446038.

[4] Mihir Bellare. New Proofs for NMAC and HMAC: Security Without Collision-Resistance. URL: https://eprint.iacr.org/2006/043.

[5] Alex Biryukov, Daniel Dinu, and Dmitry Khovratovich. Argon2: New generation of memory-hard functions for password hashing and other applications. In *2016 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 292–302, 2016. doi:10.1109/EuroSP.2016.31.

[6] BITMAIN. ANTMINER Shop. URL: https://shop.bitmain.com/.

[7] Privacy Rights Clearinghouse. Data Breaches. URL: https://privacyrights.org/data-breaches.

[8] Paul Crocker and Pedro Querido. Two Factor Encryption in Cloud Storage Providers Using Hardware Tokens. In *2015 IEEE Globecom Workshops (GC Wkshps)*, pages 1–6, December 2015. doi:10.1109/GLOCOMW.2015.7414154.

[9] Donald E. Eastlake 3rd and Paul Jones. US Secure Hash Algorithm 1 (SHA1). Request for Comments RFC 3174, Internet Engineering Task Force, September 2001. Num Pages: 22. URL: https://datatracker.ietf.org/doc/rfc3174, doi:10.17487/RFC3174.

[10] Dinei Florencio and Cormac Herley. A Large Scale Study of Web Password Habits. Technical Report MSR-TR-2006-166, Microsoft, November 2006. URL: https://www.microsoft.com/en-us/research/publication/a-large-scale-study-of-web-password-habits/.

[11] International Organization for Standardization. IT Security techniques – Entity authentication – Part 2: Mechanisms using authenticated encryption. Standard, International Organization for Standardization, Geneva, CH, June 2019. URL: https://www.iso.org/standard/67114.html.

[12] Christian Forler, Stefan Lucks, and Jakob Wenzel. Catena: A memory-consuming password-scrambling framework. Cryptology ePrint Archive, Paper 2013/525, 2013. https://eprint.iacr.org/2013/525. URL: https://eprint.iacr.org/2013/525.

[13] HashMob. Hashmob: Password recovery community. URL: https://hashmob.net/.

[14] George Hatzivasilis, Ioannis Papaefstathiou, and Charalampos Manifavas. Password Hashing Competition- Survey and Benchmark. URL: https://eprint.iacr.org/2015/265.

[15] Troy Hunt. Have I Been Pwned: Check if your email has been compromised in a data breach. URL: https://haveibeenpwned.com/.

[16] Ponemon Institute. Cost of a Data Breach Report 2022, August 2022. URL: https://www.ibm.com/resources/cost-data-breach-report-2022.

[17] Burt Kaliski. PKCS #5: Password-Based Cryptography Specification Version 2.0. Request for Comments RFC 2898, Internet Engineering Task Force, September 2000. Num Pages: 34. URL: https://datatracker.ietf.org/doc/rfc2898, doi:10.17487/RFC2898.

[18] Hugo Krawczyk. Cryptographic Extraction and Key Derivation: The HKDF Scheme. URL: https://eprint.iacr.org/2010/264.

[19] Hugo Krawczyk, Mihir Bellare, and Ran Canetti. HMAC: Keyed-Hashing for Message Authentication. Request for Comments RFC 2104, Internet Engineering Task Force, February 1997. Num Pages: 11. URL: https://datatracker.ietf.org/doc/rfc2104, doi:10.17487/RFC2104.

[20] Lani Leuthvilay. 78% of People Reset a Password They Forgot in Past 90 Days | HYPR, 2019. URL: https://blog.hypr.com/hypr-password-study-findings.

[21] Ying Liu, Fushan Wei, and Chuangui Ma. Multi-Factor Authenticated Key Exchange Protocol in the Three-Party Setting. In Xuejia Lai, Moti Yung, and Dongdai Lin, editors, *Information Security and Cryptology*, Lecture Notes in Computer Science, pages 255–267, Berlin, Heidelberg, 2011. Springer. doi:10.1007/978-3-642-21518-6_18.

[22] Security Magazine. Over 22 billion records exposed in 2021. URL: https://www.securitymagazine.com/articles/97046-over-22-billion-records-exposed-in-2021.

[23] Nils Fleischhacker Manulis, Mark and Amir Azodi. A Modular Framework for Multi-Factor Authentication and Key Exchange. URL: https://eprint.iacr.org/2012/181.

[24] Susan Moore. Gartner Forecasts Worldwide Security and Risk Management Spending to Exceed $150 Billion in 2021. URL: https://www.gartner.com/en/newsroom/press-releases/2021-05-17-gartner-forecasts-worldwide-security-and-risk-managem.

[25] Vivek Nair and Dawn Song. Multi-factor key derivation function (mfkdf), 2022. URL: https://arxiv.org/abs/2208.05586, doi:10.48550/ARXIV.2208.05586.

[26] NIST. NIST Retires SHA-1 Cryptographic Algorithm, December 2022. Last Modified: 2022-12-15T09:34-05:00. URL: https://www.nist.gov/news-events/news/2022/12/nist-retires-sha-1-cryptographic-algorithm.

[27] Nick Parker. Enhancing Password Based Key Derivation Techniques. URL: https://www.zetetic.net/storage/files/enhancing-password-based-key-derivation-techniques.pdf.

[28] Colin Percival. Stronger Key Derivation via Sequential Memory-Hard Functions. URL: https://www.researchgate.net/publication/252853607_Stronger_key_derivation_via_sequential_memory-hard_functions.

[29] Alexander Peslyak. yescrypt - a Password Hashing Competition submission. URL: https://www.password-hashing.net/submissions/specs/yescrypt-v2.pdf.

[30] David Pointcheval and Sébastien Zimmer. Multi-factor Authenticated Key Exchange. In Steven M. Bellovin, Rosario Gennaro, Angelos Keromytis, and Moti Yung, editors, *Applied Cryptography and Network Security*, Lecture Notes in Computer Science, pages 277–295, Berlin, Heidelberg, 2008. Springer. doi:10.1007/978-3-540-68914-0_17.

[31] Thomas Pornin. The MAKWA Password Hashing Function. URL: https://www.bolet.org/makwa/makwa-spec-20150422.pdf.

[32] Niels Provos and David Mazières. A Future-Adaptable password scheme. In *1999 USENIX Annual Technical Conference (USENIX ATC 99)*, Monterey, CA, June 1999. USENIX Association. URL: https://www.usenix.org/conference/1999-usenix-annual-technical-conference/future-adaptable-password-scheme.

[33] Blake C. Ramsdell. Secure/Multipurpose Internet Mail Extensions (S/MIME) Version 3.1 Certificate Handling. Request for Comments RFC 3850, Internet Engineering Task Force, July 2004. Num Pages: 16. URL: https://datatracker.ietf.org/doc/rfc3850, doi:10.17487/RFC3850.

[34] Nathalia Velez Ryan. How to Send an Email to Text Message | Twilio, 2022. URL: https://www.twilio.com/blog/how-to-send-email-to-text.

[35] Leonardo C. Santos, Ewerton R. Almeida, Paulo C. F. Andrade, Marcos A. Simplicio Jr Dos, and Paulo S. L. M. Barreto. Lyra2: Efficient Password Hashing with High Security against Time-Memory Trade-Offs. URL: https://eprint.iacr.org/2015/136.

[36] Zero Trust Engineering Team. Zero Trust Reference Architecture, August 2022. URL: https://dodcio.defense.gov/Portals/0/Documents/Library/(U)ZT_RA_v1.1(U)_Mar21.pdf.

[37] Verizon. DBIR Report 2022 - Master's Guide. URL: https://www.verizon.com/business/resources/reports/dbir/2022/master-guide/.

[38] Mountain View, David M'Raihi, Frank Hoornaert, David Naccache, Mihir Bellare, and Ohad Ranen. HOTP: An HMAC-Based One-Time Password Algorithm. Request for Comments RFC 4226, Internet Engineering Task Force, December 2005. Num Pages: 37. URL: https://datatracker.ietf.org/doc/rfc4226, doi:10.17487/RFC4226.

[39] Mountain View, Johan Rydell, Mingliang Pei, and Salah Machani. TOTP: Time-Based One-Time Password Algorithm. Request for Comments RFC 6238, Internet Engineering Task Force, May 2011. Num Pages: 16. URL: https://datatracker.ietf.org/doc/rfc6238, doi:10.17487/RFC6238.

[40] Yubico. Yubikey: Strong two-factor authentication. URL: https://www.yubico.com/.
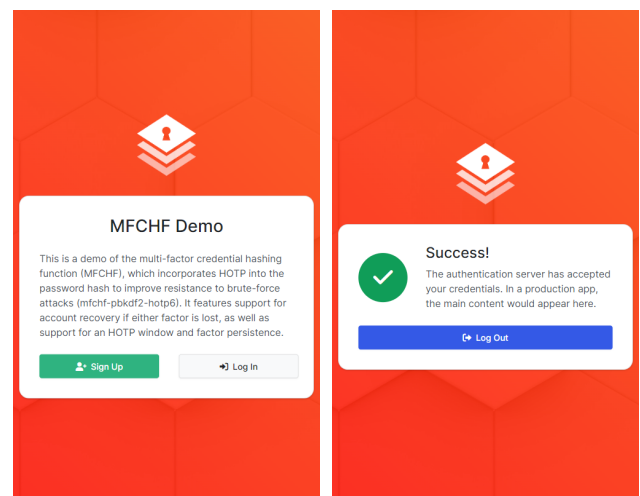
A. Demo Application
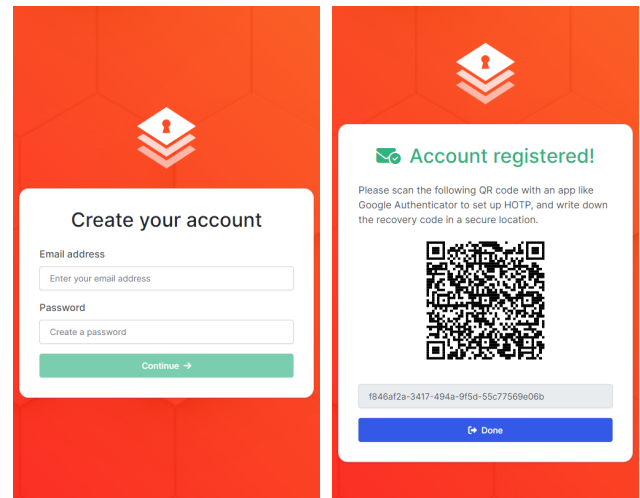
(a) Registration

(b) HOTP Setup
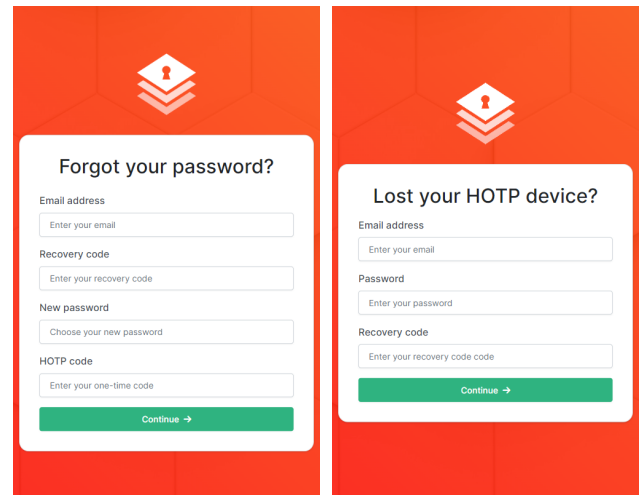
(c) Password Recovery

(d) HOTP Recovery

(e) Home Screen

(f) Internal Application

Figure 11: Authentication screens for demo application.

B. Specifications

For reproducibility, we include here the exact specifications of the device used for all benchmarking and evaluation throughout this paper. We expect the general trends observed in the results to hold regardless of the hardware used.

- **CPU**: AMD Ryzen 9 5950X (16-core, 3.40 GHz)
- **GPU**: NVIDIA GeForce RTX 3090 (10496-core, 24.0 GB)
- **RAM**: 64.0 GB (2x32, 3200 MHz)
- **SSD**: 2.0 TB NVMe M.2 (PCIe Gen3x4, 3400 MB/s)

C. Hashmob Results

Hash Type	Hashcat Mode	# of Data Breaches	% of Hashes Cracked	Mean Crack Time
MD5	0	1553	71.13% (σ=35.57%)	7.6d (σ=36.1d)
vBulletin <v3.8.5	2611	406	63.54% (σ=27.65%)	8.0d (σ=38.5d)
vBulletin >= v3.8.5	2711	379	57.49% (σ=27.67%)	0.1d (σ=48.9d)
bcrypt	3200	283	12.37% (σ=19.86%)	71.2d (σ=89.3d)
phpass	400	214	32.52% (σ=30.02%)	1205.2d (σ=15189.7d)
SHA1	100	198	74.01% (σ=36.47%)	7.5d (σ=37.7d)
MyBB 1.2+ IPB2+	2811	162	63.96% (σ=23.72%)	5.3d (σ=38.5d)
AuthMe sha256	20711	158	83.35% (σ=16.20%)	8.1d (σ=17.1d)
Django (PBKDF2-SHA256)	10000	85	15.84% (σ=18.67%)	54.5d (σ=74.0d)
md5(md5($pass))	2600	81	71.24% (σ=30.94%)	4.2d (σ=20.4d)
osCommerce xt:Commerce	21	74	73.21% (σ=32.59%)	13.1d (σ=44.5d)
MySQL4.1/MySQL5	300	73	76.82% (σ=37.97%)	2.3d (σ=17.9d)
bcrypt(md5($pass))	25600	54	24.46% (σ=20.51%)	351.9d (σ=72.7d)
md5($salt.pass)	20	48	67.73% (σ=29.34%)	10.9d (σ=80.3d)
WPA-PBKDF2-PMKID+EAPOL	22000	45	5.48% (σ=13.96%)	75.7d (σ=85.6d)
SHA2-256	1400	40	56.37% (σ=39.57%)	16.5d (σ=34.3d)
md5crypt	500	38	33.09% (σ=29.45%)	14.0d (σ=37.1d)
md5($pass.salt)	10	36	61.83% (σ=28.35%)	25.6d (σ=53.6d)
Joomla <2.5.18	11	32	60.54% (σ=36.14%)	36.5d (σ=47.6d)
NTLM	1000	31	36.94% (σ=36.45%)	23.2d (σ=51.1d)
SMF >v1.1	121	27	57.49% (σ=30.67%)	59.4d (σ=82.0d)
MySQL323	200	26	82.18% (σ=30.85%)	13.8d (σ=39.5d)
OpenCart	13900	25	37.61% (σ=21.86%)	36.9d (σ=56.7d)
sha1($salt.pass)	120	24	63.74% (σ=25.05%)	42.0d (σ=70.3d)
nsldaps SSHA-1(Base64)	111	22	16.32% (σ=7.30%)	143.0d (σ=38.4d)
Drupal7	7900	20	15.43% (σ=29.53%)	144.3d (σ=162.3d)
Django (SHA-1)	124	19	76.37% (σ=20.46%)	17.6d (σ=10.9d)
sha256($salt.pass)	1420	18	58.60% (σ=27.83%)	33.9d (σ=32.2d)
SHA2-512	1700	18	51.91% (σ=35.59%)	28.2d (σ=68.4d)
sha512($salt.pass)	1720	17	31.82% (σ=26.60%)	82.9d (σ=15.5d)
descrypt	1500	15	56.91% (σ=40.26%)	20.9d (σ=56.7d)
Ruby on Rails Restful Auth	27200	14	67.07% (σ=23.94%)	57.6d (σ=71.1d)
sha1($pass.salt)	110	12	40.88% (σ=39.77%)	40.8d (σ=106.7d)
sha256($pass.salt)	1410	12	40.00% (σ=31.05%)	38.8d (σ=58.0d)

TABLE 1: Summary of 4,259 data breaches on HashMob, excluding formats with $\leq$ 10 data breaches or 0 cracks.

D. Evaluation Results

Hash Type	Attack Mode	Cost Parameters	Mean Verification Time	Mean Crack Time
MD5	Hashcat 10	n/a	16.9 μs	26.91 ms
SHA1	Hashcat 110	n/a	21.1 μs	41.07 ms
SHA256	Hashcat 1410	n/a	10.6 μs	41.85 ms
SHA512	Hashcat 1710	n/a	10.2 μs	40.93 ms
PBKDF2-SHA2	Hashcat 20300	n=1000000	192.4 ms	5552.75 ms ($\approx$ 5.5 s)
Bcrypt	Hashcat 3200	c=12	190.3 ms	10071.5 ms ($\approx$ 10 s)
Scrypt	Hashcat 8900	n=32768, r=24, p=1	202.4 ms	35378.8 ms ($\approx$ 35 s)
Argon2id	John (argon2)	m=4096, t=150, p=1	199.7 ms	2.69×10^5 ms ($\approx$ 4.5 min)
MFCHF-HOTP6	John (argon2)	m=4096, t=150, p=1	201.6 ms	2.7×10^{11} ms ($\approx$ 8.6 yr)
MFCHF-TOTP6	John (argon2)	m=4096, t=125, p=1	199.6 ms	2.3×10^{11} ms ($\approx$ 7.2 yr)
MFCHF-OOBA6	John (argon2)	m=4096, t=135, p=1	200.9 ms	$\approx \infty$ (>10,000 yr)
MFCHF-HSHA1	John (argon2)	m=4096, t=150, p=1	199.8 ms	$\approx \infty$ (>10,000 yr)

TABLE 2: Average verification and crack time for a variety of hash functions including MFCHF.

E. Algorithms

Algorithm 1 MFCHF with YubiKey (mfchf-hmacsha1)

Require: HS1 is HMAC-SHA1 per RFC 2014 [19]
Require: H is a password hash function (e.g., argon2)
1: **function** SETUP(password)
2:     hmackey $\leftarrow$ Random$(0, 2^{160})$
3:     challenge $\leftarrow$ Random$(0, 2^{160})$
4:     salt $\leftarrow$ Random$(0, 2^{256})$
5:     response $\leftarrow$ HS1(hmackey, challenge)
6:     digest $\leftarrow$ H(password $\odot$ hmackey $\odot$ salt)
7:     paddedkey $\leftarrow$ hmackey $\oplus$ response
8:     hash $\leftarrow$ {paddedkey, challenge, salt, digest}
9:     **return** hash, hmackey
10: **end function**
11: **function** VERIFY(password, response, hash)
12:     {paddedkey, salt, digest} $\leftarrow$ hash
13:     hmackey $\leftarrow$ paddedkey $\oplus$ response
14:     expected $\leftarrow$ H(password $\odot$ hmackey $\odot$ salt)
15:     **if** digest $\neq$ expected **then**
16:         **return** *reject*
17:     **end if**
18:     challenge $\leftarrow$ Random$(0, 2^{160})$
19:     response $\leftarrow$ HS1(hmackey, challenge)
20:     paddedkey $\leftarrow$ hmackey $\oplus$ response
21:     hash $\leftarrow$ {paddedkey, challenge, salt, digest}
22:     **return** *accept*, hash
23: **end function**

Algorithm 2 MFCHF with HOTP (mfchf-hotp6)

Require: HOTP is HOTP per RFC 4226 [38]
Require: H_1 is a password hash function (e.g., argon2)
Require: H_2 is a standard hash function (e.g., sha256)
1: **function** SETUP(password)
2:     target $\leftarrow$ Random$(0, 10^6)$
3:     hotpkey $\leftarrow$ Random$(0, 2^{256})$
4:     salt $\leftarrow$ Random$(0, 2^{256})$
5:     counter $\leftarrow$ 1
6:     firstotp $\leftarrow$ HOTP(hotpkey, counter) $\% \ 10^6$
7:     offset $\leftarrow$ (target $-$ firstotp) $\% \ 10^6$
8:     pad $\leftarrow$ H_1(password $\odot$ target $\odot$ salt)
9:     paddedkey $\leftarrow$ hotpkey $\oplus$ pad
10:     digest $\leftarrow$ H_2(pad)
11:     hash $\leftarrow$ {counter, offset, paddedkey, salt, digest}
12:     **return** hash, hotpkey
13: **end function**
14: **function** VERIFY(password, otp, hash)
15:     {counter, offset, paddedkey, salt, digest} $\leftarrow$ hash
16:     target $\leftarrow$ (offset $+$ otp) $\% \ 10^6$
17:     pad $\leftarrow$ H_1(password $\odot$ target $\odot$ salt)
18:     **if** H_2(pad) $\neq$ digest **then**
19:         **return** *reject*
20:     **end if**
21:     counter $\leftarrow$ counter $+$ 1
22:     hotpkey $\leftarrow$ paddedkey $\oplus$ pad
23:     nextotp $\leftarrow$ HOTP(hotpkey, counter) $\% \ 10^6$
24:     offset $\leftarrow$ (target $-$ nextotp) $\% \ 10^6$
25:     hash $\leftarrow$ {counter, offset, paddedkey, salt, digest}
26:     **return** *accept*, hash
27: **end function**

Algorithm 3 MFCHF with TOTP (mfchf-totp6)

Require: HOTP is HOTP per RFC 4226 [38]
Require: T, T_0, T_X are TOTP times per RFC 6238 [39]
Require: H_1 is a password hash function (e.g., argon2)
Require: H_2 is a standard hash function (e.g., sha256)
1: **function** SETUP(password, w)
2:     target $\leftarrow$ Random$(0, 10^6)$
3:     totpkey $\leftarrow$ Random$(0, 2^{256})$
4:     salt $\leftarrow$ Random$(0, 2^{256})$
5:     counter $\leftarrow$ $\lfloor (T - T_0)/T_X \rfloor$
6:     **for** j **in** $[0 \ldots w]$ **do**
7:         otp $\leftarrow$ HOTP(totpkey, counter $+$ j) $\% \ 10^d$
8:         offsets[j] $\leftarrow$ (target $-$ otp) $\% \ 10^d$
9:     **end for**
10:     pad $\leftarrow$ H_1(password $\odot$ target $\odot$ salt)
11:     paddedkey $\leftarrow$ totpkey $\oplus$ pad
12:     digest $\leftarrow$ H_2(pad)
13:     hash $\leftarrow$ {counter, offsets, paddedkey, salt, digest}
14:     **return** hash, totpkey
15: **end function**
16: **function** VERIFY(password, otp, hash)
17:     {counter, offsets, paddedkey, salt, digest} $\leftarrow$ hash
18:     index $\leftarrow$ $\lfloor (T - T_0)/T_X \rfloor - $ counter
19:     target $\leftarrow$ (offsets[index] $+$ otp) $\% \ 10^6$
20:     pad $\leftarrow$ H_1(password $\odot$ target $\odot$ salt)
21:     **if** H_2(pad) $\neq$ digest **then return** *reject*
22:     **end if**
23:     counter $\leftarrow$ $\lfloor (T - T_0)/T_X \rfloor$
24:     totpkey $\leftarrow$ paddedkey $\oplus$ pad
25:     **for** j **in** $[0 \ldots w]$ **do**
26:         otp $\leftarrow$ HOTP(totpkey, counter $+$ j) $\% \ 10^d$
27:         offsets[j] $\leftarrow$ (target $-$ otp) $\% \ 10^d$
28:     **end for**
29:     hash $\leftarrow$ {counter, offsets, paddedkey, salt, digest}
30:     **return** *accept*, hash
31: **end function**

Algorithm 4 MFCHF with OOBA (mfchf-ooba6)

Require: (Enc, Dec) is public-key encryption
Require: H is a password hash function (e.g., argon2)
1: **function** SETUP(password, pk)
2:     target $\leftarrow$ Random$(0, 36^6)$
3:     firstotp $\leftarrow$ Random$(0, 36^6)$
4:     offset $\leftarrow$ (target $-$ firstotp) $\% \ 36^6$
5:     salt $\leftarrow$ Random$(0, 2^{256})$
6:     digest $\leftarrow$ H(password $\odot$ target $\odot$ salt)
7:     ct $\leftarrow$ Enc(firstotp, pk)
8:     hash $\leftarrow$ {ct, pk, offset, salt, digest}
9:     **return** hash
10: **end function**
11: **function** VERIFY(password, otp, hash)
12:     {pk, offset, salt, digest} $\leftarrow$ hash
13:     target $\leftarrow$ (offset $+$ otp) $\% \ 10^6$
14:     expected $\leftarrow$ H(password $\odot$ target $\odot$ salt)
15:     **if** digest $\neq$ expected **then return** *reject*
16:     **end if**
17:     nextotp $\leftarrow$ Random$(0, 36^6)$
18:     offset $\leftarrow$ (target $-$ nextotp) $\% \ 36^6$
19:     ct $\leftarrow$ Enc(nextotp, pk)
20:     hash $\leftarrow$ {ct, pk, offset, salt, digest}
21:     **return** *accept*, hash
22: **end function**