Multi-Factor Key Derivation Function (MFKDF) for Fast, Flexible, Secure, & Practical Key Management

Vivek Nair and Dawn Song
University of California, Berkeley

Abstract

We present the first general construction of a Multi-Factor Key Derivation Function (MFKDF). Our function expands upon password-based key derivation functions (PBKDFs) with support for using other popular authentication factors like TOTP, HOTP, and hardware tokens in the key derivation process. In doing so, it provides an exponential security improvement over PBKDFs with less than 12 ms of additional computational overhead in a typical web browser. We further present a threshold MFKDF construction, allowing for client-side key recovery and reconstitution if a factor is lost. Finally, by "stacking" derived keys, we provide a means of cryptographically enforcing arbitrarily specific key derivation policies. The result is a paradigm shift toward direct cryptographic protection of user data using all available authentication factors, with no noticeable change to the user experience. We demonstrate the ability of our solution to not only significantly improve the security of existing systems implementing PBKDFs, but also to enable new applications where PBKDFs would not be considered a feasible approach.

1 Introduction

Since the introduction of PBKDF1 in 1991, password-based key derivation functions (PBKDFs) have enjoyed widespread use and deployment in a variety of settings, including in the WPA [25] and WPA2 [26] protocols, LastPass [35] and Dashlane [16] applications, and Windows [13] and iOS [5] operating systems. PBKDFs provide a convenient solution to the usable key management problem: by deterministically deriving keys based on a user's password, systems can achieve end-to-end encryption while providing a seamless user experience and subverting the need for secure key storage. However, the recent surge in password-based attacks like credential stuffing and password spraying has highlighted the critical weakness of passwords as a sole authentication factor.

Although a growing number of platforms have implemented multi-factor authentication in response to this threat, password-derived keys (and thus all secrets encrypted with them) remain only as secure as the passwords they are based on. We therefore suggest the use of, and provide a novel construction for, a key derivation function that incorporates all of a user's multiple authentication factors into the key derivation process. In doing so, user data is, for the first time, directly cryptographically protected by all authentication factors.

While incorporating all of a user's login factors into the key derivation process seems like a natural and long overdue extension of PBKDFs, achieving this with the most popular secondary authentication factors currently in use is difficult in practice. Current PBKDFs rely on the relatively unchanging nature of passwords to convert them, via deterministic one-way functions (OWFs), into fixed encryption keys. By contrast, most of the secondary factors in popular use today constitute one-time passwords (OTPs) that are expected to change upon each user login, which does not readily facilitate the derivation of a static key.

In this paper, we describe and evaluate the first known Multi-Factor Key Derivation Function (MFKDF) with support for common authentication factors like TOTP, HOTP, OOBA (e.g., Email/SMS), HMAC-SHA1 (e.g., YubiKey), and, of course, static factors like passwords and recovery codes. We do so using fast, standardized cryptographic primitives, resulting in a total computational overhead of $\leq$ 12 ms over PBKDFs in a typical web application (§11), with no noticeable changes to the user experience.

Our MFKDF construction represents a fast, flexible, secure, and practical key management solution that provides several advantages over existing PBKDFs: it supports self-service client-side key recovery without creating a central point of failure, can be used to implicitly authenticate without revealing authentication factors to a server, and enables cryptographic enforcement of arbitrarily complex authentication policies. We demonstrate these features by implementing and testing MFKDF in realistic, full-stack centralized and decentralized applications, including a trustless decentralized cryptocurrency wallet for which PBKDFs would not have been sufficiently secure (§10.2).

Contributions

1. We provide the first general construction of a multi-factor key derivation function (§4). Our function provides an exponential security improvement over PBKDFs (§6).
2. We provide KDF constructions for several popular authentication factors (§5), including the first known KDF constructions based on HOTP and TOTP (§5.2).
3. We provide a k-of-n threshold MFKDF construction (§8.1) that can be used to facilitate self-service account recovery without a central point of failure (§8.2).
4. We illustrate how MFKDF can cryptographically enforce arbitrarily specific key derivation policies (§9).

2 Background & Motivation

In this section, we aim to motivate the need for a multi-factor key derivation function by following the prototypical use case of a password management system. Password managers represent a common application for client-side key derivation, with password-based key derivation functions currently being a major architectural component of popular password management products like LastPass [35], Dashlane [16], and 1Password [1]. We first describe a typical architecture of a modern password management system, then describe the drawbacks of this current architecture, and finally illustrate how the multi-factor key derivation function presented herein can serve to remedy these flaws.

Password-Based Key Derivation

Password management systems are, by their nature, highly security-sensitive due to their role as a gateway to all of a user's online accounts. Systems that provide adequate security for this application are generally trustless, as users are often concerned not only with the threat of outside actors, but also with potential malicious activity of the service itself. The chief goal of a secure password management system architecture is therefore to preserve the confidentiality of stored secrets even in the event that all centralized system components are compromised by an adversary.[1] Today, password-based key derivation functions like PBKDF1 and PBKDF2 [29] are critical tools for enabling this level of security. In general, a password-based key derivation function F converts a password P, salt S, and optional configuration parameters cfg into a key DK of length $dkLen$: $DK = F(P, S, cfg, dkLen)$.

In practice, most password-based key derivation functions are built upon, and inherit the security properties of, cryptographic hash functions like SHA-256 [22]. When a user signs in to a typical password management application, PBKDF2 is used on the client side to derive a key from the user's password. A symmetric encryption function like AES-256 [38] is then used to encrypt all of a user's secrets on the client side prior to their storage in a centralized database. Thus, even an adversary with complete access to the database will not be able to derive the key necessary to decrypt the user's secrets without knowing their password.

An important property of password-based key derivation functions is a degree of intentional computational inefficiency that increases the relative difficulty of brute-force attacks. For example, the PBKDF2 configuration used by LastPass invokes 100,000 rounds of SHA-256 to increase its computational difficulty to a degree that remains relatively unnoticeable to users but is significantly burdensome to brute-force attackers. Advanced password-based key derivation functions like Argon2 [10] have been developed to further resist brute-force attacks, but operate in a fundamentally similar way to the functions like PBKDF2 described above.

[1] To further motivate this standard, we note that LastPass has experienced at least 6 security incidents [39], including two database breaches [45, 47].

Multi-Factor Authentication

While the password-based key derivation approach described above is effective at binding a user's secrets to their master password, it is not adequate on its own to protect a user's account due to the well-known insecurity of passwords as a sole authentication factor [20, 21] and their susceptibility to attacks such as credential stuffing [2]. Therefore, services typically use multi-factor authentication (MFA) in conjunction with password-based key derivation. Popular secondary authentication factors include "soft tokens" like HMAC-based One-Time Password (HOTP) [51] and Time-based One-Time Password (TOTP) [52], "hard tokens" like YubiKeys [53], and Out-of-Band Authentication (OOBA) factors like email and SMS [24]. These factors are inserted by password managers into the login process for obtaining an authentication token necessary to access encrypted secrets stored in a database.

The use of MFA during the login process may prevent attackers from accessing stored secrets under correct system operation, but fails to meet the previously-stated security goal of surviving a complete system breach. In the event of a compromise, passwords remain the only factor necessary to decrypt and access secrets. Thus, the current method of MFA only superficially addresses the threat of attacks like password spraying and credential stuffing in the context of password management. What is therefore needed is a mechanism for ensuring that a user's encryption key cannot be derived without utilizing all of their authentication factors.

Account Recovery

Absent additional considerations for account recovery, systems using password-based key derivation are liable to a complete loss of user data in the event of a lost password. In light of the fact that user passwords are, in fact, frequently forgotten by end users [23], this risk is generally considered untenable for users and service providers alike. This risk can be mitigated via the use of a master key as shown in Fig. 1.

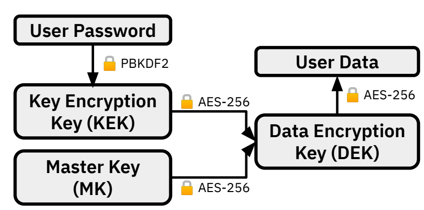


Figure 1: Typical architecture for account recovery via master key.

As illustrated above, when a user forgets their password, they are presented with one or more alternative security challenges (e.g., security questions). Upon successful completion of these challenges, the system uses the master key to recover the data encryption key and decrypt any stored data.

While the use of a central master key for data recovery may be deemed a necessary concession for the sake of usability, it once again fails to meet our security goal of surviving centralized system compromise; in fact, it largely defeats the initial purpose of using client-side user-derived keys.

Due to the obvious drawbacks of this approach, modern password managers rarely rely on central recovery keys for individual accounts. Instead, they largely utilize approaches that store recovery keys on trusted user devices. For instance, LastPass [35], 1Password [1], and Dashlane [16] all support account recovery via biometrics by storing a key on a trusted smartphone. LastPass additionally supports SMS recovery by storing a key on a trusted browser, while 1Password supports the use of Linux authentication for recovery.

In all instances, without a central key, users must maintain access to at least one trusted device where a recovery key has previously been stored if they hope to recover their account. Some services, like 1Password, go as far as to allow users to manually store and manage their secret key, which must be used together with their password to authenticate on new devices. Once again, no recourse is available if this key is not retained in at least one secure location. What is needed is a key derivation method that supports client-side recovery of secrets when one or more authentication factors is forgotten while requiring neither a trusted device nor a central point of failure.

Motivation

In summary, current system architectures based on password-based key derivation are flawed in a number of significant ways. They fail to incorporate secondary authentication factors into the key derivation process, leaving users susceptible to credential stuffing and password spraying, and introduce a central point of failure when using master keys for account recovery. These flaws are not problems of specific system architectural designs, but rather are fundamentally implicated by the use of password-based key derivation in a system.

In this paper, we aim to present a complete solution to the problems highlighted above by constructing a multi-factor key derivation function (MFKDF) that realizes the full benefits of multi-factor authentication within the key-derivation process (§4). Unlike current architectures, our suggested approach supports client-side key recovery without using a master key or trusted device (§8.2). In §10.1, we demonstrate a proof-of-concept password management system architecture based on MFKDF and illustrate its security advantages over current architectures. Importantly, we do so with no noticeable change to the user experience, and while supporting all of the same 2FA factors that users have already grown accustomed to, providing a more streamlined account login and recovery experience than systems using manual key management.

Of course, the current applications of password-based key derivation go far beyond password management, including use in the Windows [13] and iOS [5] operating systems and WPA [25] and WPA2 [26] wireless protocols. The MFKDF approach presented herein is naturally applicable to many of these systems, such as to bring the benefits of multi-factor authentication to operating systems and wireless networks. Further, in §10.2, we will illustrate how the security properties of MFKDF enable its use even in contexts where PBKDFs would never have been considered sufficiently secure.

3 Problem Statement

Krawczyk's analysis of HKDF [30] provides an excellent formalization of key derivation functions. We use a modified framework of Krawczyk here to give a formal statement of the security of our MFKDF constructions.

There are two components to our framework: *factor constructions*, which exist to derive static key material from a specific authentication factor, and *KDF constructions*, which use the results of one or more factor constructions to derive a key. We begin by discussing the factor constructions.

Factor Constructions

Definition 1. A factor F is a two-valued probability distribution (σ, α) generated by an efficient probabilistic algorithm.

Factors are said to have a private component (σ) and a public component (α). For example, if $F = (\sigma, \alpha)$ is a security question, then α might contain the publicly-known question and σ might contain the secret answer. On the other hand, if F is a password, then α might contain the publicly-known strength requirements while σ contains the password itself. In this paper, we refer to the public component α as the "factor parameters" and the private component σ as the "factor material." For some factors, the parameters α might change over time, but the material σ must remain constant.

Definition 2. We say that F is a statistical m-entropy factor if for all s and a in support of F, the conditional probability $Prob(\sigma = s \mid \alpha = a)$ is at most 2^{-m}.

Consider a factor F where α is always *"What's your favorite number between 1 and 64?"* and users chose σ uniformly in $[1, 64]$. Then F is a statistical 6-entropy factor.

Definition 3. We say that F is a computational m-entropy factor if there is a statistical m-entropy factor F' that is computationally indistinguishable from F.

Consider modifying the previous example such that α also contains a secure encryption of σ. Now F is a computational 6-entropy factor since σ is theoretically determinable within $[1, 64]$ given α, yet it is computationally infeasible to do so (compared to the difficulty of guessing $\sigma \in [1, 64]$).

Definition 4. A factor construction $\mathcal{F}$ uses a factor witness W_i and parameters α_i to output the factor material σ and new parameters α_{i+1}; $\mathcal{F} : (W_i, \alpha_i) \mapsto (\sigma, \alpha_{i+1})$.

A factor witness (W_i) refers to the one-time value used to demonstrate possession of a factor. For example, W_i might be a one-time 6-digit code in the case of HOTP. The role of the factor construction $\mathcal{F}$ is to output a constant σ despite the witness W_i potentially changing. The ability to update the parameters ($\alpha_i \mapsto \alpha_{i+1}$) each time $\mathcal{F}$ is invoked is critical to making this possible.

In this paper, we split $\mathcal{F}$ into the *FactorDerive* function, producing σ, and the *FactorUpdate*, producing α_{i+1}. Each factor also requires a corresponding *FactorSetup* function to produce the initial parameters α_0 given some configuration.

Goals. In this paper, we aim to provide factor constructions (*FactorSetup*, *FactorDerive*, and *FactorUpdate*) for computational m-entropy factors representing HOTP, TOTP, YubiKey, OOBA, and constant factors like passwords.

KDF Constructions

Definition 5. A KDF $\mathcal{D}$ outputs K (a string of ℓ bits) and α_{i+1} given ℓ and values (σ, α_i) sampled from a factor F.

Note that by definition, a KDF is defined with respect to a single factor. In practice, we will show that F can be a "composite factor" which combines multiple input factors, allowing KDF to be a multi-factor KDF (MFKDF). Thus, our MFKDF construction needs both an *MFKDFDerive* function ($\mathcal{D}$) and an *MFKDFSetup* function that establishes the initial parameters (α_0) of the composite factor F.

Definition 6. A *KDF* is said to be (t, q, ε)-secure with respect to a factor F if given a $(\sigma, \alpha) \in F$ no attacker $\mathcal{A}$ knowing α running in time t and making at most q arbitrary queries to *KDF* can distinguish between $KDF(\sigma, \ell)$ and $\{0,1\}^\ell$ with probability larger than $1/2 + \varepsilon$.

Def. 6 is our main security definition for MFKDF. It states that the output key K should be hard to distinguish from random noise $\{0,1\}^\ell$ without knowing the secret component (factor material) σ of the input factor F.

Definition 7. A KDF is called (t, q, ε) m-entropy secure if it is (t, q, ε)-secure with respect to all m-entropy factors.

Note that because our MFKDF constructions require special treatment of each factor type, we cannot assert their security on arbitrary m-entropy factors. However, it is reasonable to make this assumption of the underlying KDF (e.g., HKDF [30]) used to construct MFKDF.

Goals. In this paper, we aim to provide an MFKDF construction (*MFKDFSetup*, *MFKDFDerive*) which uses a set of factors $S = \{F_A, F_B, \ldots\}$ and parameters α_i to produce a key K and new parameters α_{i+1}; $\mathcal{D} : (S, \alpha_i) \mapsto (K, \alpha_{i+1})$.

We also aim to provide a k-of-n threshold multi-factor KDF construction, where given any C consisting of $\geq k$ factors in S ($\forall C \subseteq S$ s.t. $|C| \geq k$), $\mathcal{D} : (C, \alpha_i) \mapsto (K, \alpha_{i+1})$.

Problem Setting

Per the above definitions, each *KDF* and factor F has private components (W_i, σ, K) and public components (α). We assume a legitimate user is the only party able to generate factor witnesses W_i for their authentication factors, and thus is the only party able to derive factor material σ and keys K. We make no assumptions, however, of parties knowing α; the parameters α can even be stored on a public blockchain.

Definition 8. Factors F_A and F_B are independent if for all $(\sigma_a, \alpha_a) \in F_A$ and $(\sigma_b, \alpha_b) \in F_B$, $P(\sigma_a = s_a \mid \alpha_a = a_a) = P(\sigma_a = s_a \mid \alpha_a = a_a \wedge \alpha_b = a_b)$, $\wedge\, P(\sigma_b = s_b \mid \alpha_b = a_b) = P(\sigma_b = s_b \mid \alpha_a = a_a \wedge \alpha_b = a_b)$.

We add the assumption of independent factors to our problem setting and security definitions. Most factors in practice are independent because they rely on a random key. A counter-example would be a security question whose answer overlaps with the contents of a password.

Policy Constructions

Finally, we introduce the notion of a "policy-based MFKDF," a stronger construction than the ones introduced above which allows implementers to specify exactly which factor combinations are allowed to derive a key.

Definition 9. Given a set of factors $S = \{F_A, F_B, \ldots\}$, an allowable factor combination C is any subset of factors in S ($C \subseteq S, C \neq \varnothing$) that can be used to derive a key K.

Definition 10. A policy P is a set of all allowable factor combinations ($P = \{C_1, C_2, \ldots\}$) that can be used to derive a key K.

In the previously-described k-of-n threshold MFKDF, P was restricted to containing all subsets of S of size $\geq k$. However, per the above definition, P can now contain any nonempty subsets of S, regardless of size.

Goals. We aim to provide a policy-based multi-factor KDF construction w.r.t. P: $\forall C \in P$, $\mathcal{D}_P : (C, \alpha_i) \mapsto (K, \alpha_{i+1})$.

Security Goals. Standard and threshold MFKDF are subcases of policy-based MFKDF, so we define our security goals in the policy setting. Let $S = \{F_1, F_2, \ldots, F_n\}$ be n independent computational $\{m_1, m_2, \ldots, m_n\}$-entropy factors. Let *PKDF* be (t, q, ε)-secure. Now given a policy-based MFKDF *KDF* w.r.t. *PKDF* and any $P \in \mathcal{P}(\mathcal{P}(S) \setminus \varnothing) \setminus \varnothing$:

- **Correctness.** For any $C \in P$, $(K, \alpha') \leftarrow KDF_P(C, \alpha)$, and $C' \in P$, $(K', \alpha'') \leftarrow KDF_P(C', \alpha')$, $K = K'$.
(Providing any valid combinations of factors should always result in deriving the same key.)

- **Safety.** For any $C \in P$, $(K, \alpha') \leftarrow KDF_P(C, \alpha)$, and $C' \notin P$, $(K', \alpha'') \leftarrow KDF_P(C', \alpha')$, $K \neq K'$ except with negligible probability w.r.t. key size ℓ.
(Providing an invalid set of factors should be highly unlikely to derive the correct key.)

- **Entropy**. Let E denote $\sum\{m_1, m_2, \ldots, m_n\}$ for all $C \in P$. Then the MFKDF construction w.r.t. P shall be (t, q, ε)-secure w.r.t. a factor F_K, where F_K is a computational $(min(E))$-entropy factor.
(Attacking the derived key should be as hard as attacking the weakest set of allowed factors.)

4 Multi-Factor Key Derivation

We identified a few fundamental difficulties with implementing even a basic n-of-n multi-factor key derivation function. Firstly, there are a wide variety of authentication factors that must be supported, each requiring its own treatment of inputs and outputs to be used for key derivation. Secondly, many of these factors are constantly-changing OTPs, which nevertheless need to be used to derive the correct, static key if and only if the correct OTP (i.e., witness W_i) is provided at that instant. A third, self-imposed constraint is "plug-and-play" compatibility with existing systems using PBKDFs; systems should not need to be entirely rearchitected to use MFKDF, and the user experience should not be impacted whatsoever. This section outlines, at a high level, an MFKDF design that meets these goals along with the security goals of §3. A formal statement of this design is given in §A.3.

MFKDF Construction

In Fig. 2, we illustrate an MFKDF system consisting of general-purpose *MFKDFSetup* and *MFKDFDerive* functions, which present a standard interface to any number of factor-specific *FactorSetup*, *FactorDerive*, and *FactorUpdate* functions. This modular approach allows for individual treatment of various authentication factors, many of which are described in §5, as well as potential forward-compatibility with future factors. During a setup phase (e.g., upon account creation), initial key parameters ($\alpha_{K,0}$) are produced, encapsulating several factor parameters $\{\alpha_{F_A,0}, \alpha_{F_B,0}, \ldots\}$. The ith derive phase (upon ith login) then proceeds as follows:

1. The key parameters $\alpha_{K,i}$ are split into several factor parameters $\{\alpha_{F_A,i}, \alpha_{F_B,i}, \ldots\}$.
2. Each factor witness $\{W_{F_A,i}, W_{F_B,i}, \ldots\}$, along with its factor parameters, $\{\alpha_{F_A,i}, \alpha_{F_B,i}, \ldots\}$, is converted by its *FactorDerive* function into factor material σ_F.
3. The factor material $\{\sigma_{F_A}, \sigma_{F_B}, \ldots\}$ is combined by *MFKDFDerive* into key material σ_K.
4. Let *KDF* be a memory-hard PBKDF like Argon2 [10]. *MFKDFDerive* uses *KDF* on σ_K to produce key K.
5. The *FactorUpdate* functions use K and $\alpha_{F,i}$ to produce $\alpha_{F,i+1}$ for each factor.
6. The factor parameters $\{\alpha_{F_A,i+1}, \alpha_{F_B,i+1}, \ldots\}$ are combined with the output of *MFKDFDerive* to produce the updated key parameters, $\alpha_{K,i+1}$.

A major innovation in this approach is the feedback loop that allows the factor constructions to use K in producing $\alpha_{F,i+1}$, which has the effect of allowing factor constructions to hide secrets using K. These secrets can be used to "setup" the $i+1$th key derivation during the ith derivation, making possible the use of OTP factors.

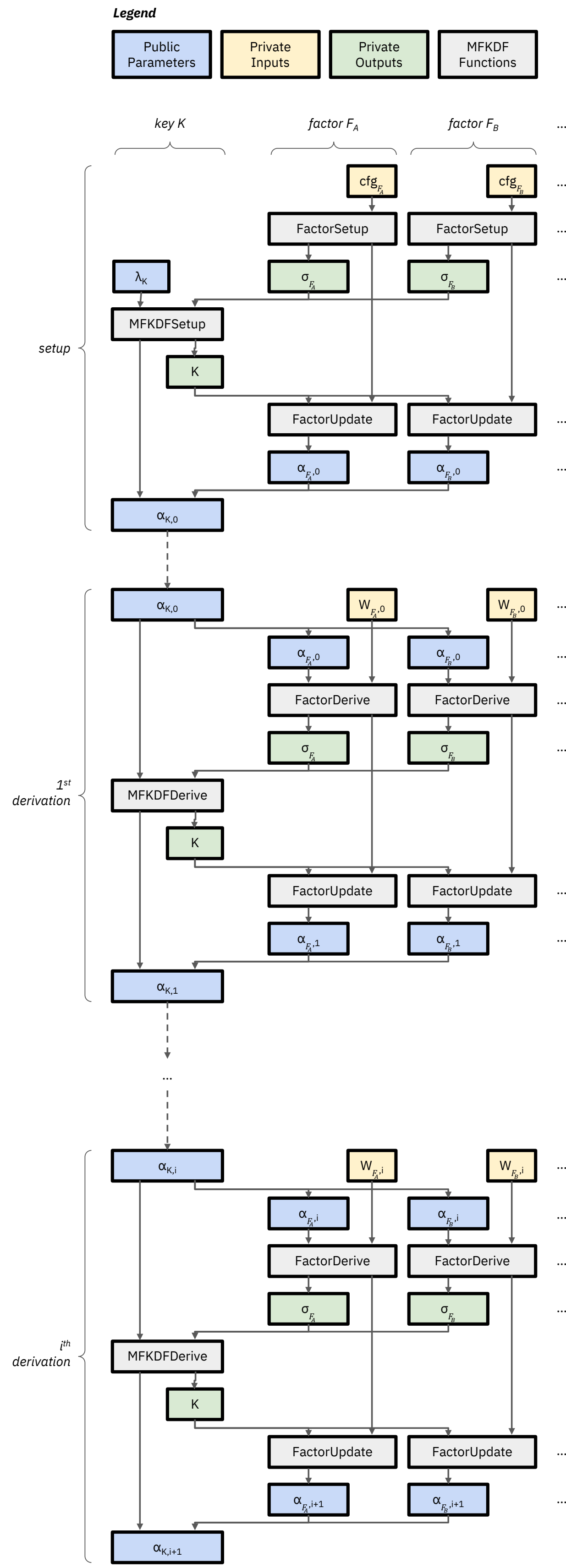


Figure 2: Multi-phase modular MFKDF construction.

5 Factor Constructions

As described in §1, the chief contribution of MFKDF is not necessarily the concept of a KDF with multiple inputs, but rather its support for existing, commonly used authentication factors, including dynamic factors like HOTP/TOTP. In this section, we describe MFKDF factor constructions for a wide variety of popular authentication factors. The goal of each factor derivation function is to convert factor parameters ($\alpha_{F,i}$) and a factor witness $W_{F,i}$ (e.g., a password, YubiKey response, or TOTP code) into fixed factor material σ_F that can be used by MFKDF to derive a key (K), along with updated factor parameters ($\alpha_{F,i+1}$). Due to the key-feedback mechanism of §4, the factor derivation function can use K when producing $\alpha_{F,i+1}$, but not when producing σ_F. A formal statement of each factor is given in §A.2.

5.1 Constant Factors

We begin by discussing constant factors such as passwords, security questions, and recovery codes, as these are amongst the easiest factors to construct. In general, PBKDFs are already well suited to handle such factors, so the witness $W_{F,i}$ can be directly returned as σ_F. In some cases, a *transform* function first is applied to $W_{F,i}$, such as to standardize the case of a security answer, or to extract useful data from a UUIDv4 recovery code. No parameters are required ($\alpha_F = \varepsilon$) due to the artifacts being static. Our MFKDF construction would therefore probably be overkill for simply combining multiple static factors, but their inclusion here is useful to facilitate classic factor combinations like password + TOTP.

5.2 Software Tokens

We next describe factor constructions for "soft tokens," specifically HOTP [51] and TOTP [52]. Our constructions rely on the information-theoretic security of modular arithmetic rings to convert a dynamic OTP ($W_{F,i}$) into a fixed integer ($target_F$) that can be used as σ_F. The resulting factors are perfectly backward-compatible with existing HOTP/TOTP implementations like "Google Authenticator." Both support a variable number of digits d; typically, $d = 6$.

5.2.1 HOTP

The HOTP factor construction is the first to take advantage of the key feedback loop described in §4. Although a user's HOTP code is expected to be different upon each derivation, one can predict what the user's next HOTP code will be if they know the HOTP key ($hotpkey_F$) and counter (ctr_F). On the other hand, $hotpkey_F$ cannot be stored openly, as it would allow an attacker to bypass the HOTP factor. The key feedback mechanism allows $hotpkey_F$ to be stored securely in α_F encrypted with the final derived key K while still being used during the ith HOTP factor derivation to set up the $i+1$th derivation. During the HOTP factor setup process for $hotpkey_F$, a fixed $target_F$ is set to a random integer in range $[0, 10^d)$. The first OTP ($otp_{F,0}$) is determined using $hotpkey_F$ with $ctr_{F,0} = 1$, and the modular difference is stored as $offset_{F,0} = (target_F - otp_{F,0}) \% 10^d$. The factor parameters $\alpha_{F,0}$ consist of $(d, ctr_{F,0}, offset_{F,0}, ct_F)$, where ct_F is $hotpkey_F$ encrypted under K. The factor derivation process (with $W_{F,i} = otp_{F,i}$) is as follows:

1. $(d, ctr_{F,i}, offset_{F,i}, ct_i)$ are obtained from $\alpha_{F,i}$.
2. $\sigma_F = target_F$ is found as $(offset_{F,i} + W_{F,i}) \% 10^d$.

At this point, the factor material (σ_F) has been successfully recovered (assuming $W_{F,i}$ is correct), and the MFKDF derivation of K can proceed. The HOTP factor can then use K for the remainder of the derivation process via the key-feedback loop described in §4:

3. Decrypt $hotpkey_F$ from ct_F using K.
4. Increment ctr_F ($ctr_{F,i+1} = ctr_{F,i} + 1$).
5. Determine $otp_{F,i+1}$ using $hotpkey_F$ and $ctr_{F,i+1}$.
6. Calculate $offset_{F,i+1}$ as $(target_F - otp_{F,i+1}) \% 10^d$.
7. Return $(d, ctr_{F,i+1}, offset_{F,i+1}, ct_F)$ as $\alpha_{F,i+1}$.

In summary, the above HOTP factor construction succeeds at converting a dynamic HOTP code into fixed factor material σ_F through use of a modular $offset_F$ that is updated each round using $hotpkey_F$ (stored as ct_F).

5.2.2 TOTP

Our TOTP factor construction uses a similar fundamental approach to the HOTP construction, given that by definition, $TOTP(K) = HOTP(K, \lfloor (T - T_0)/T_X \rfloor)$ where T is the current UNIX time, T_0 is the initial time, and T_X is the time interval (usually 30 seconds). However, given that we cannot predict exactly which time T the next derivation will occur, each possible $offset_F$ must be calculated within a fixed window w, corresponding to $\{ctr, ctr+1, ..., ctr+t\}$. Because $hotpkey_F$ is known to the client during the setup and update phases, these offsets can be calculated without involving the TOTP application. Thus, upon derivation at time T, all subsequent derivations between T and $w \cdot T_X$ will succeed at recovering σ_F (the same approach can also be used to facilitate counter desynchronization when using HOTP). This approach is quite practical, with a window of $w = 87600$ (30 days) requiring just 219 kb of storage, and less than 850 ms to setup and derive (see §11). When implemented, the TOTP derivation function need not re-calculate offset values corresponding to times in the future, and all offset calculation can be done in the background without blocking overall MFKDF key derivation or usage since σ_F is output at step 2 of the above process.

The proposed approaches for HOTP and TOTP-based key derivation require no modifications whatsoever to existing applications like Google Authenticator. Because $hotpkey_F$ is stored within α_F (encrypted as ct_F), the calculation of the next offset value(s) occurs entirely within the *FactorSetup* and *FactorUpdate* functions, and does not involve the authenticator application at all beyond the user obtaining $otp_{F,i}$ from the app once each login, as is always required.

Theorem. The HOTP and TOTP factor constructions yield computational $(d/log_{10}2)$-entropy factors.

Proof. Consider $F = (\sigma, \alpha)$ where $\sigma = target_F$ and $\alpha = (d, ctr_F, offset_F)$. Now σ is uniformly random in $[0, 10^d)$, and d, ctr_F, and $offset_F$ reveal no further information about σ, so for all s and a in support of F, the conditional probability $Prob(\sigma = s \mid \alpha = a)$ is at most $10^{-d} = 2^{-(d/log_{10}2)}$. Thus, F is a statistical $(d/log_{10}2)$-entropy factor. Now consider F' which adds ct_F to α; F is computationally indistinguishable from F' if *Enc* is secure. F' is the construction of §5.2. Thus the HOTP and TOTP factors are computational $(d/log_{10}2)$-entropy factors. ∎

Remark. Why should entropy be limited to $d/log_{10}2$ when HOTP/TOTP secrets are much larger than this? MFKDF cannot extract more entropy than the size of the *witness* used to authenticate, regardless of the size of the underlying secret. In particular, because the user inputs a d-digit code to authenticate, MFKDF cannot possibly extract more entropy from this factor than the brute-force attack search space of all 10^d possible codes. Thus, the entropy gain from HOTP/TOTP factors in MFKDF is theoretically optimal with respect to the amount of entropy that is possible to derive from these factors.

5.3 Hardware Tokens

We next turn our attention to hardware authentication devices ("hard tokens") like USB security keys and smart cards. While FIDO Universal 2nd Factor (U2F) and FIDO2 are the most commonly used standards for interacting with such devices, they are intentionally impossible to use for key derivation [18]. Specifically, their inclusion of a hardware-generated random nonce in all signatures makes device responses completely nondeterministic even if the secret is known. Fortunately, most hard tokens, including all YubiKeys [53], also support authenticating via HMAC-SHA1 challenge-response, which can be used to derive an MFKDF factor.

In the simplest construction of this factor, the HMAC key hk_F is itself used as σ_F. During a setup phase, a random challenge $c_{F,0}$ is generated, and the corresponding response $r_{F,0}$ is determined using hk_F. Both $c_{F,0}$ and $pad_{F,0} = r_{F,0} \oplus hk_F$ are stored in $\alpha_{F,0}$.[2] The factor derivation (with $W_{F.i} = r_{F,i}$) then proceeds as follows:

1. $c_{F,i}$ and $pad_{F,i}$ are extracted from $\alpha_{F,i}$.
2. The HMAC key is recovered as $hk_F = W_{F,i} \oplus pad_{F,i}$.
3. Generate a random 160-bit challenge $c_{F,i+1}$.
4. Get the corresponding response using hk_F with HMAC-SHA1 (*HS1*): $r_{F,i+1} = HS1(hk_F, c_{F,i+1})$.
5. Calculate the new pad, $pad_{F,i+1} = r_{F,i+1} \oplus hk_F$.
6. Let $\sigma_F = hk_F$, $\alpha_{F,i+1} = (c_{F,i+1}, pad_{F,i+1})$.

When a user wishes to sign in, they simply extract $c_{F,i}$ from $\alpha_{F,i}$, and generate $r_{F,i}$ with their YubiKey or equivalent device. This approach succeeds at generating static key material σ_F from devices such as YubiKeys supporting HMAC-SHA1, while maintaining the freshness of non-repeating challenges and responses. An alternative approach could be to use a fixed random value for σ_F and directly query the device with $c_{F,i+1}$ to get $r_{F,i+1}$, which has the benefit of not requiring hk_F to be known to the KDF, but does not provide the same freshness.

5.4 Out-of-band Authentication

Finally, we provide a general construction for out-of-band authentication (OOBA) factors such as SMS, email, and push notifications, which are currently amongst the most commonly used 2FA methods. Such factors, unlike all those previously described, are not completely trustless, instead requiring a degree of trust in the underlying channel (e.g., the cell carrier). The OOBA factor takes advantage of this by encrypting a dynamic OTP of d digits under the public key of the channel (pk_F). The modular addition of said OTP with a fixed $target_F$ provides the same information-theoretic security as in the HOTP and TOTP constructions.

In the setup phase, a fixed $target_F$ and initial $otp_{F,0}$ are both chosen randomly in range $[0, 10^d)$. As with HOTP/TOTP, the modular difference is stored as $offset_{F,0} = (target_F - otp_{F,0}) \ \% \ 10^d$. The parameters $\alpha_{F,0}$ consist of $(d, pk_F, offset_{F,0}, ct_F)$, where ct_F is now otp_F encrypted under pk_F. The factor derivation process (with $W_{F,i} = otp_{F,i}$) then proceeds as follows:

1. $(d, pk_F, offset_{F,i}, ct_F)$ are recovered from $\alpha_{F,i}$.
2. $\sigma_F = target_F$ is found as $(offset_{F,i} + W_{F,i}) \ \% \ 10^d$.
3. The next OTP ($otp_{F,i+1}$) is chosen in range $[0, 10^d)$.
4. Calculate $offset_{F,i+1}$ as $(target_F - otp_{F,i+1}) \ \% \ 10^d$.
5. Encrypt $otp_{F,i+1}$ under pk_F as $ct_{F,i+1}$.
6. Return $(d, pk_F, offset_{F,i+1}, ct_{F,i+1})$ as $\alpha_{F,i+1}$.

When a user wishes to authenticate via OOBA, they can obtain $ct_{F,i}$ from $\alpha_{F,i}$, submit $ct_{F,i}$ to the OOBA channel, and use $W_{F,i} = otp_{F,i}$ from the channel to derive σ_F. In practice, using the S/MIME key [42] of the recipient as pk_F is a good way to implement the OOBA factor for email, which can be extended to SMS using each cell carrier's email-to-SMS gateway service [48].

[2] In this paper, $\oplus$ denotes bitwise XOR and $\odot$ denotes concatenation.

6 Entropy & Brute-Force Resistance

Each of the factors presented in §5 is a computational m-entropy factor with the value of m shown in Tab. 1. For any given factor listed below, MFKDF extracts and fully utilizes the amount of entropy available in the witnesses used to authenticate. Thus, while it may be disappointing that any given factor does not provide more entropy, such limitations are inherently imposed by the factors themselves, not MFKDF.

Factor		**Entropy Bits (m or λ_F)**	
		General	**Typical**
Constant	Passwords	Varies	≈ 40 [20]
	Security Questions	Varies	≤ 10 [12]
	UUIDv4 (Recovery Code)	122	
Soft Token	HOTP	$d/\log_{10}2$	≈ 20 when $d=6$
	TOTP	$d/\log_{10}2$	≈ 20 when $d=6$
Hard Token	HMAC-SHA1 (e.g., YubiKey)	160	
OOBA	SMS, Email, & Push	$d/\log_{10}2$	≈ 20 when $d=6$

Table 1: Entropy of supported MFKDF factors.

A vitally important feature of the MFKDF construction in §4 is the application of a computationally-difficult KDF only after all factor material as been combined into mat_K. As a result, attackers cannot separately guess individual MFKDF factors, and must simultaneously correctly guess all factors to derive a key. Due to the "avalanche effect" (correlation freeness) [3] of the underlying KDF, if the derived key is incorrect, an attacker cannot easily determine which factor(s) were wrongly guessed. Therefore, given n available factors with a mean factor entropy of $\bar{m}$ bits, the crack time t will be $t \propto 2^{\bar{m}}$ for PBKDFs and $t \propto 2^{\bar{m}n}$ for MFKDF. We thus claim that n-factor MFKDF provides an exponential security improvement over PBKDFs; in other words, an MFKDF defined with respect to n m-entropy factors $(F_1, F_2, ..., F_n)$ is nm-entropy secure (see extended paper [37] for proofs).

To illustrate the practical effect of this property, consider a typical PBKDF-derived key with 40 bits of entropy and a 2-factor password-plus-HOTP MFKDF-derived key with 60 bits of entropy. Assuming PBKDF2-SHA256-100,000 as the underlying PBKDF (the configuration used by LastPass [39]), a 1 TH/s attacker[3] should require ≈ 1.27 days to crack the PBKDF key and $\approx 3,653$ years to crack the MFKDF key, while the derivation time for the real user remains constant.

7 Authenticating with Derived Keys

So far, we have presented an MFKDF construction and a series of factor constructions that succeed at converting all of a user's authentication factors into a static key that can be used to encrypt user secrets. Doing so marks a paradigm shift in multi-factor authentication from software-based assurance to direct cryptographic protection of sensitive user data using all available authentication factors. The addition of secondary factors such as TOTP into the key derivation process allows user data to potentially remain secure under total system compromise, even in light of threats like credential stuffing. However, we pause for a moment to step back and consider the original purpose of these secondary factors: securely authenticating users.

A typical way of validating secondary factors like TOTP would be to store a user's TOTP key at a central authentication server. During authentication, the server generates the current TOTP code and compares it to the user-submitted value. Doing so, however, would largely defeat the purpose of using a TOTP factor in MFKDF, as an attacker who compromises the authentication server could steal the user's TOTP key and derive at least that portion of the user's MFKDF key. The same challenge exists with many of the factors presented in §5. Therefore, when using MFKDF, we require a means of verifying a user's factors that does not require server-side storage of factor-related secrets.

Thankfully, the security properties of MFKDF allow the derived key to itself be used to authenticate end users and implicitly verify all of their authentication factors. Because a properly-configured MFKDF key (K) cannot feasibly be derived without the presentation of all constituent factors, verifying a user's derivation of K effectively constitutes verification of all factors. A standard key-based authentication algorithm like ISO/IEC 9798-2 Unilateral Authentication [27] can therefore now be used to authenticate users.[4]

One final obstacle is that an MFKDF-derived key K should not directly be shared with an authentication server if it is also to be relied on for data confidentiality. Thus, after K is derived, separate sub-keys $datakey_K$ and $authkey_K$ should be derived using HKDF [32]; $datakey_K$ can be used to encrypt user data (e.g., using AES [38]) and $authkey_K$ can be used for authentication as described above. This approach, illustrated in Fig. 3, results in trustless cryptographic assurance of both data confidentiality and user authentication when using MFKDF. It is significantly advantageous over password-based authentication because a breach of the authentication server does not compromise confidentiality due to the use of separate client-side keys for encryption and authentication.

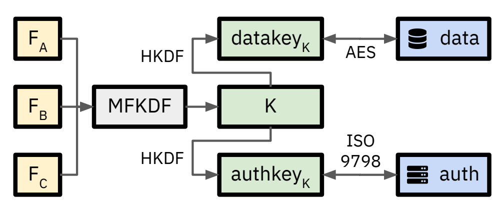


Figure 3: MFKDF key establishment for authentication and confidentiality.

[3] A single AntMiner S19 provides over 100 TH/s of SHA-256 [11].

[4] The asymmetric variants of ISO 9798-2 can also be deployed here by using a symmetric MFKDF-derived key as a fixed seed to deterministically generate an asymmetric key pair via HMAC-DRBG [8].

8 Threshold MFKDF Construction

In §2, we introduced the challenge of account recovery when using password-derived keys. This challenge exists, perhaps to an even greater extent, when using multi-factor derived keys, as the loss of any authentication factor could now imply total data loss if no further provisions are made. While said provisions typically entail using a master key with password-based key derivation, we describe in this section an equivalent recovery mechanism for MFKDF that does not implicate a central point of failure.

In a highly typical 2FA-enabled system, a user establishes a password, a secondary factor (e.g., HOTP), and a recovery factor (e.g., a recovery code). The password and secondary factor are used during normal logins; if the password is forgotten, the secondary factor and recovery factor can be used together to recover it, and similarly, if the secondary factor is forgotten, the password and recovery factor can be used together to recover it.

We note that the above policy is effectively equivalent to saying any 2 of the 3 established factors $\{F_A, F_B, F_C\}$ can be used to access an account, where in that case, F_A is a password, F_B is a HOTP code, and F_C is a recovery code. We make explicit this concept of 2-of-3 factors being required by introducing a general t-of-n threshold MFKDF construction. In doing so, we allow MFKDF keys to be recovered on the client side as normal even if $n-t$ factors are lost, providing resilience against partial factor loss while codifying what is anyway the de facto standard for recovery.

8.1 Threshold Multi-Factor Key Derivation

As with the basic n-of-n MFKDF construction in §4, our threshold construction consists of separate setup and derive functions. The setup function uses n factors to output a key of ℓ bits ($K \in \{0,1\}^\ell$) and t-of-n parameters ($\alpha_{K,0}$), and the derive function uses at least t of the n factors and $\alpha_{K,i}$ to output K and $\alpha_{K,i+1}$.

Let $HKDF$ be HKDF per RFC 5869 [32], $(Share, Comb, Rec)$ be Shamir's secret sharing [44], and KDF be a hard PBKDF like Argon2 [10]. Given n factors ($\{F_1, F_2, ..., F_n\}$), a threshold t, and security parameter ℓ, a k-of-n MFKDF key can be setup like so:

1. Sample σ_K uniformly randomly in $\{0,1\}^\ell$.
2. Split σ_K into n shares using $Share(\sigma_K, t, n)$.
3. For each factor F_j, get factor material σ_{F_j}.
4. Expand each σ_{F_j} to size ℓ using HKDF.
5. Combine each expanded factor with its corresponding share $share_{F_j}$ ($pad_{F_j} = Enc(share_{F_j}, \sigma_{F_j})$) to get $pads_K$.
6. Use KDF on σ_K to produce K.
7. Get $params_{K,0}$ as $\alpha_{F,0}$ for each factor using K.
8. Return $(t, \ell, params_{K,0}, pads_K)$ as $\alpha_{K,0}$.

Using t of the n original factors ($\{F_1, F_2, ..., F_t\}$) and $\alpha_{K,i}$, the derivation function is as follows:

1. Get $(t, \ell, params_{K,i}, pads_K)$ from $\alpha_{K,i}$.
2. For each factor F_j, get factor material σ_{F_j}.
3. Expand each σ_{F_j} to size ℓ using HKDF.
4. Decrypt each expanded factor with its corresponding pad in $pads_K$ to recover a share $share_{F_i}$.
5. Combine shares using $Comb(\{share_{F_i}, ..., share_{F_k}\})$ to recover σ_K.
6. Use KDF on σ_K to produce K.
7. Get $\alpha_{F,i+1}$ using K for each provided factor and update $params_{K,i}$ accordingly to produce $params_{K,i+1}$.
8. Return $(t, \ell, params_{K,i+1}, pads_K)$ as $\alpha_{K,i+1}$.

This threshold MFKDF construction takes advantage of the same feedback mechanism as §4 to facilitate dynamic OTP factors, and the factor constructions from §5 can remain unchanged due to the overall modular design. Some aspects of the description are simplified for clarity; e.g., it omits the use of a salt. As before, a more complete formal statement of the scheme is given in §A.4.

8.2 Recovery & Reconstitution

Consider the 2-of-3 key described above based on a password (F_A), HOTP code (F_B), and recovery code (F_C). Using the above t-of-n threshold MFKDF construction with $t = 2$, a user who has forgotten their password will still be able to derive their key using F_B and F_C and recover their account. This client-side recovery process constitutes a major security improvement by eliminating central master keys. However, the user may now want to replace F_A with a new password F_{A_2} moving forward, ideally without having to establish a brand new key and re-encrypt all secrets. The above t-of-n threshold MFKDF construction is specifically designed to support this replacement operation like so:

1. Extract $(t, \ell, params_{K,in}, pads_{K,in})$ from $\alpha_{K,in}$.
2. Derive σ_K from t known factors as in §8.1.
3. Get $\sigma_{F_{A_2}}$ and $\alpha_{F_{A_2}}$ of the new factor. Update $params_{K,in}$ to reflect $\alpha_{F_{A_2}}$, yielding $params_{K,out}$.
4. Recover $share_{F_A}$ (jth) as $Rec(\sigma_K, t, n, j)$.
5. Update pad_{F_A}, e.g., $pad_{F_{A_2}} = share_{F_A} \oplus \sigma_{F_{A_2}}$.
6. Update $pads_{K,in}$ to reflect pad_{FA_2}, yielding $pads_{K,out}$.
7. Store $\alpha_{K,out} = (t, \ell, params_{K,out}, pads_{K,out})$.

The $\alpha_{K,out}$ produced by this operation is effectively identical to $\alpha_{K,in}$, other than the forgotten factor F_A being replaced by F_{A_2}; the resulting key K is the same. This process, termed "key reconstitution," allows the factors constituting a threshold MFKDF-derived key to be updated while keeping the resulting K static so that encrypted secrets remain accessible. The process can be extended to add or remove factors, or update the threshold t, without changing K as long as σ_K is known.

9 Cryptographic Policy Enforcement

The t-of-n threshold MFKDF construction of §8 is sufficient for enabling many useful authentication schemes, but falls short of enforcing every combination of factors we might reasonably hope to enforce. For example, in a four factor setup with a password, HOTP code, recovery code, and security questions, a 2-of-4 threshold MFKDF setup would enable recovering either the password or HOTP code using either the recovery code or security questions, but would also allow the key to be derived with just the recovery code and security questions. Furthermore, one may wish to assert that the password only be recovered using security questions, and that the HOTP factor only be recovered using a recovery code. To achieve this level of specificity in enforcement of exactly which factor combinations should be allowed to derive a key requires a more granular key policy framework.

9.1 Policy Framework

Consider a set of available authentication factors ($S = \{F_A, F_B, ...\}$). An allowable factor combination C_i is defined as a non-empty subset of factors in S ($C_i \subseteq S, C_i \neq \varnothing$) that can be used to derive a key (key_K). A policy P is a set of all unique allowable factor combinations ($P = \{C_1, C_2, ...\}$) to derive a key. A policy P is said to be cryptographically enforced by a KDF if a user presenting a factor combination ($C = \{F_A, F_B, ...\}$) will be able to derive key_K if and only if $C \in P$. Given a set of factors S, a fully expressive policy framework will be able to enforce any policy $P \in \mathcal{P}(\mathcal{P}(S) \setminus \varnothing) \setminus \varnothing$. For example, a fully-expressive policy framework for three authentication factors (F_A, F_B, F_C) will be able to enforce a policy P defined by any non-empty subset of $\{\{F_A\}, \{F_B\}, \{F_C\}, \{F_A, F_B\}, \{F_B, F_C\}, \{F_A, F_C\}, \{F_A, F_B, F_C\}\}$.

To note explicitly a few non-requirements, asserting the absence of a factor (e.g., "user does not know password F_A") is not meaningful in this context. Allowing derivation based on an empty set of factors ($P_K = \{\varnothing\}$) is trivial but not particularly useful, and an empty policy ($P_K = \varnothing$), implying a key is impossible to derive, is also not useful.

Using this definition of fully-expressive policy enforcement, we present in this section a framework for enforcing arbitrarily complex authentication policies using MFKDF.

9.2 Key Stacking

We introduce the notion of "key stacking" as a building block towards cryptographic policy enforcement. Key stacking entails using one multi-factor derived key as factor material for another multi-factor derived key. Therefore, when one wishes to derive an MFKDF key with stacked key factors, they may first need to derive one or more intermediate MFKDF keys, which are used as inputs to the final MFKDF key derivation. The many intentional symmetries in the *Setup* and *Derive* functions of MFKDF and those of MFKDF factors, namely the use of a α_F or α_K, and output of fixed σ_F or σ_K, may provide sufficient intuition into the key stacking method, but a formal construction is still given in §A.2 for completeness.

To illustrate how key stacking enables enforcement of previously impossible factor combinations, consider again the four-factor setup described above with a password (F_A), HOTP code (F_B), recovery code (F_C), and security questions (F_D), whereby the password should only be recovered using security questions, and the HOTP should factor only be recovered using a recovery code. With key stacking, we can now enforce this pattern like so:

1. Let K_P be a 1-of-2 MFKDF key using $\{F_A, F_B\}$.
2. Let K_Q be a 1-of-2 MFKDF key using $\{F_A, F_C\}$.
3. Let K_R be a 1-of-2 MFKDF key using $\{F_D, F_B\}$.
4. Via key stacking, $F_P = K_P$, $F_Q = K_Q$, $F_R = K_R$.
5. Let K be a 1-of-3 MFKDF key using $\{F_P, F_Q, F_R\}$.

Per the above construction, K can be derived using a password and HOTP code, password and recovery code, or security questions and HOTP code, but not using security questions and recovery code, password and security questions, or HOTP and recovery code. Therefore, our desired authentication policy has been achieved. In §9.3, we will show that this technique is sufficient to enforce arbitrarily complex authentication policies.

When using key stacking, only the final MFKDF derivation needs a hard underlying PBKDF, and all intermediate MFKDF derivations can use a fast KDF like HKDF or even return σ_K directly. We thus expect about 10 ms of additional overhead per stacked key (see §11). For security reasons, only the final derived key should be fed back to the factor update functions for all constituent factors.

9.3 Policy Enforcement

Per the completeness definition of §9.1, threshold MFKDF and key stacking are sufficient to cryptographically enforce any desired authentication policy $P = \{C_1, C_2, ...\}$ given a set of factors $S = \{F_A, F_B, ...\}$ as follows:

1. For all $C_i \in P$, let K_i be an n-of-n key with $\forall F \in C_i$.
2. Let $\{F_{K_1}, F_{K_2}, ...\}$ be $\{K_1, K_2, ...\}$ via key stacking.
3. Let K be a 1-of-n key with $\{F_{K_1}, F_{K_2}, ...\}$.

Per the above construction, a user presenting a factor combination C will be able to derive K via a stacked sub-key K_i if and only if $C \in P$; this is true for any $K_i = P \in \mathcal{P}(\mathcal{P}(S) \setminus \varnothing) \setminus \varnothing$. Note while the above construction proves MFKDF can achieve arbitrary authentication policy enforcement, it is not necessarily the most efficient way to implement any given authentication policy. The example of §9.5 illustrates a more direct and concretely efficient implementation of a particular policy.

9.4 Security Statement

Theorem. Let *HKDF* be (t,q,ε)-secure. Given a set of independent computational m_i-entropy factors S and any policy $P \in \mathcal{P}(\mathcal{P}(S) \setminus \varnothing) \setminus \varnothing$, let E_i denote $\sum\{m_1, m_2, \dots, m_n\}$ for all $C_i \in P$. Let $j = argmin_i E_i$, and F_P be a combination of C_j. Then the policy-based MFKDF construction of §A.5 w.r.t *HKDF* and P shall be (t,q,ε)-secure w.r.t. F_P.

Proof Sketch. Full proofs are given in the extended paper [37]. We begin by showing that n factors can be combined to produce a factor with entropy equal to the sum of input factor entropy. We use this to show that the basic MFKDF (§4) is secure w.r.t. the combination of its factors. Next, we show that in the 1-of-n case, the threshold MFKDF (§8.1) is secure w.r.t. its lowest-entropy factor. Finally, because policy MFKDF (§9.3) is a composition of basic n-of-n MFKDF and threshold 1-of-n MFKDF, it is secure w.r.t. the sum of factor entropy in the weakest allowable combination $C \in P$.

9.5 Practical Example

Consider the following nuanced but fairly realistic authentication policy:

1. Users MUST authenticate using a password and TOTP.
2. Users MAY recover either factor using security questions or a UUIDv4 recovery code.
3. Users MUST also always use email OTP for recovery.
4. Users MAY bypass TOTP when using a known device.[5]

The above policy can be enforced using threshold MFKDF with key stacking, as shown in Fig. 4. In fact, we implement (and cryptographically enforce) this exact policy in our centralized proof-of-concept system in §10.1.

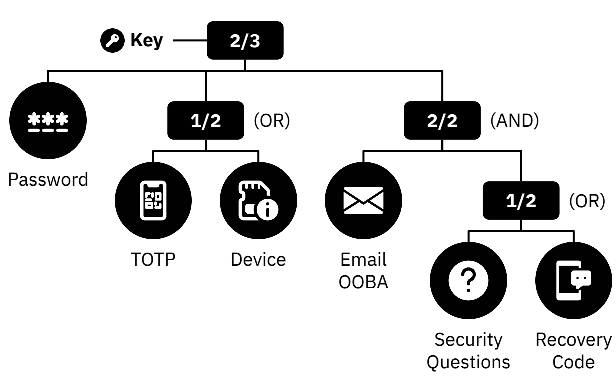


Figure 4: MFKDF factor tree enforcing sample authentication policy.

[5]A device factor was not discussed in §5, but can be constructed quite easily by persisting factor material on a trusted device, e.g., as a cookie.

10 Applications

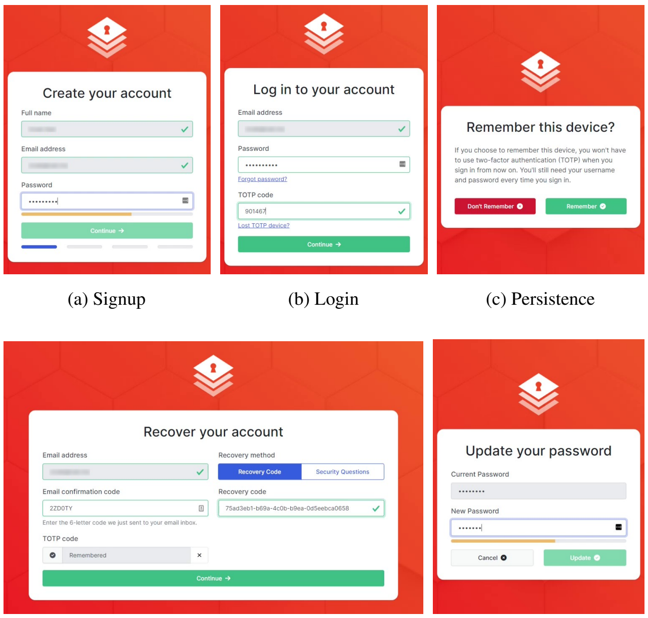
To illustrate the immediate practical utility of MFKDF, we implemented and evaluated a fully-featured MFKDF library in JavaScript, which we used to produce two proof-of-concept applications. The first proof-of-concept (§10.1) is a centralized full-stack web application used to demonstrate the use of MFKDF in current PBKDF applications. The second proof-of-concept (§10.2) is a fully decentralized application intended to illustrate the use of MFKDF in environments where PBKDFs would not be viable. We also used this library to perform a performance evaluation in §11.

10.1 Centralized Proof-of-Concept

In §2, we motivated the need for MFKDF by using password managers as an archetypal example of PBKDF-based systems. We now revisit the example of password management by discussing our implementation of a secure MFKDF-based password management application. To this end, we implemented and evaluated a full-stack password management web application using MFKDF for both authentication and encryption of stored passwords. All MFKDF operations take place on the client side; the host can be fully untrusted. The integrity of the client-side JavaScript code, including the MFKDF library, must be ensured, such as through the use of Subresource Integrity (SRI) tags or a trusted CDN. This is exactly the deployment model currently used by major PBKDF2-based web applications such as LastPass and Dashlane, with PBKDF2 simply being replaced by MFKDF in our application.

Using the method of §9, an MFKDF-based password management application can implement and cryptographically enforce an authentication policy of its choosing; in our proof-of-concept application, we chose to use the policy of §9.5. Per this policy, a user can log in normally using a password and TOTP code, and can recover either factor if lost by using email OOBA along with either a recovery code or security questions. When enforced with MFKDF, this policy represents a 10^6-times increase in brute-force difficulty over PBKDFs, and is a major improvement over existing password management systems which use a central master key to facilitate recovery per SP 800-57 [7]. We also implemented the recovery and reconstitution method of §8.2, allowing users to change a forgotten factor after recovery without re-encrypting secrets.

Aside from the various security advantages of using MFKDF in this application, our major takeaway from this endeavor is with respect to the usability of MFKDF. MFKDF does not demand the use of new authentication factors purpose-built for key derivation, instead supporting the same (unmodified) factors like HOTP, TOTP, YubiKey, or OOBA that users are familiar with and likely already using (e.g., we verified that our application was fully compatible with the latest version of Google Authenticator.) They can therefore use the same signup, login, and recovery processes, with MFKDF operating transparently in the background to provide enhanced security with no tangible impact on the UI or UX (see Fig. 5). We also showed that the implementation of common convenience features like skipping MFA on trusted devices and recovering and reconstituting lost factors are not hindered by the use of MFKDF. The use of MFKDF during the login and setup processes introduced less than 100 ms of overhead to each.



(a) Signup (b) Login (c) Persistence

(d) Recovery (e) Reconstitution

Figure 5: Authentication screens for centralized demo application.

10.2 Decentralized Proof-of-Concept

To demonstrate the potential use of MFKDF in fully decentralized applications, we implemented a distributed cryptocurrency wallet that supports "logging in" with traditional authentication factors (username, password, and MFA), but relies on no committees or trust assumptions. In essence, the wallet key is derived directly from the user's authentication factors with MFKDF whenever it is needed, rather than storing the key (or shares thereof) on any device. While the idea of such a wallet has already been proposed [17], its implementation hasn't been possible without MFKDF.

Fig. 6 illustrates the architecture we used to implement the proof-of-concept decentralized MFKDF wallet. Users create their wallet by establishing a 2-of-3 threshold MFKDF key. Upon creation, the policy document is uploaded to the InterPlanetary File System (IPFS) [9] and a corresponding InterPlanetary Name System (IPNS) [34] record is created, the address of which becomes the "username."[6] A user "logs in" to their wallet using said username along with at least two of their three authentication factors, allowing them to retain access to their wallet even if any one factor is lost or forgotten. If the MFKDF policy document is updated, such as to reconstitute a lost factor or upon each login for factors like HOTP, the new policy is uploaded to IPFS and the IPNS record is updated accordingly.

[6]In the future, Ethereum Name Service (ENS) [33] could be used instead of IPNS to facilitate human-readable usernames.

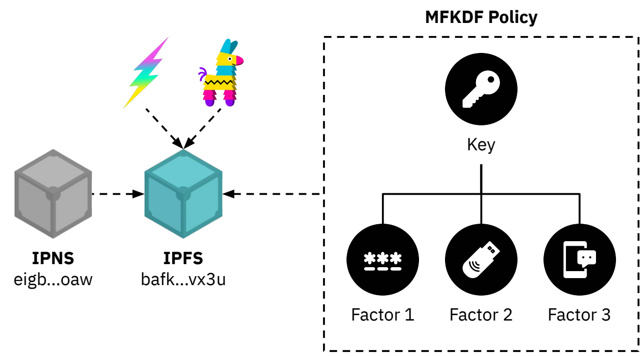


Figure 6: Architecture of decentralized MFKDF wallet demo application

Our MFKDF-based wallet has many of the traditional benefits of custodial wallets (namely, portability, recoverability, and multi-factor authentication with familiar authentication factors) while in fact being decentralized, trustless, and non-custodial. We present this proof-of-concept mainly to emphasize that the potential applications for MFKDF reach far beyond the situations where PBKDFs would typically be deployed. PBKDFs would not be considered feasible for such an application due to the lower key entropy, potential for credential stuffing, password spraying, and brute-force, and the inability to recover from a lost factor.

11 Performance Evaluation

To evaluate the performance of MFKDF in a practical setting, we benchmarked a fully-featured JavaScript implementation of MFKDF in Chrome Browser v103.0.5060.114 on Windows 10 v21H2. Our test device used an AMD Ryzen 9 5950X (16-core, 3.4 GHz), although only single-thread performance is relevant in this browser setting.

11.1 MFKDF Performance

Fig. 7 shows the setup and derivation time for a 3-of-3 MFKDF key and 2-of-3 threshold MFKDF key based on the mean of $N = 100$ setup and derive iterations with password, HOTP, and recovery code (UUIDv4) factors.

The 3-of-3 MFKDF setup and derivation had a mean computation time of $\bar{x} = 3.84$ ms and $\bar{x} = 10.52$ ms respectively. For the 2-of-3 threshold MFKDF, these were $\bar{x} = 8.83$ ms and $\bar{x} = 11.90$ ms. In both cases, HKDF was used as the KDF to isolate the overhead of MFKDF from the computational difficulty of the underlying KDF.

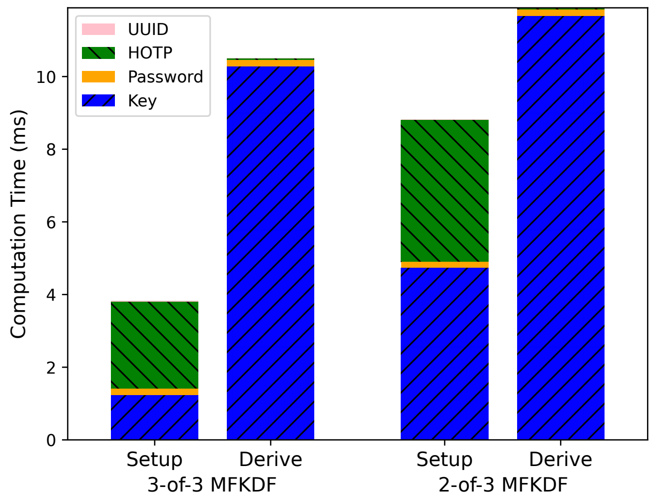


Figure 7: Setup and derivation time of 2-of-3 and 3-of-3 MFKDF.

11.2 Factor Performance

Fig. 8 shows the range of computation times for the setup and derive functions of each supported factor across 100 iterations. No individual factor has a setup or derive time of more than 2 ms, other than OOBA ($\bar{x} = 21.2$ ms, $\bar{x} = 19.62$ ms) and TOTP ($\bar{x} = 33.16$ ms, $\bar{x} = 0.49$ ms).

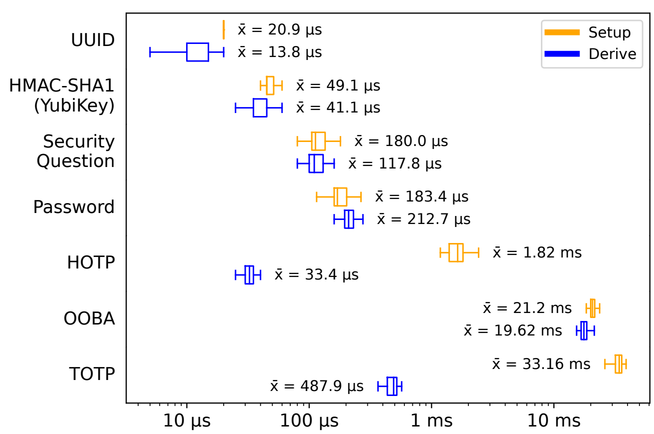


Figure 8: Setup and derivation time of all supported MFKDF factors.

The overhead of the TOTP factor is highly dependent on the time window parameter used. The above results reflect a window of 2920 cycles (24 hours). If the time window is increased to 87600 cycles (30 days), the setup time is $\bar{x} = 841$ ms and derivation time is $\bar{x} = 9.31$ ms.

Discussion. We noted in §2 that PBKDFs are intentionally computationally difficult so as to slow the rate of brute-force attacks. Most sources target a key derivation time of 250 ms. While we used HKDF instead of a PBKDF in the above tests to isolate the overhead of the MFKDF construction from the intentional difficulty of the underlying KDF, the results should be understood in the context of this 250 ms target for the underlying KDF. Thus, even the 20 ms overhead of the slowest factors (other than TOTP with a large w) is negligible.

12 Related Work

There are no known works that describe a general-purpose multi-factor key derivation function or key derivation using OTP authentication factors like TOTP and HOTP, based on a thorough search of terms such as “multi-factor key derivation,” “two-factor key derivation,” and “hotp/totp key derivation.”

Several prior works have proposed two-factor key derivation approaches based specifically on a YubiKey hardware token and a password [14, 40]. Entirely biometric-based key derivation is also a widely studied problem [6, 43, 46, 50].

Many works propose alternatives to key derivation for solving the key management problem, in particular by secret sharing a key amongst several user devices or [15] a trusted committee [17, 28]. Such solutions require a majority of the committee or devices to be honest and uncompromised, an assumption we do not make in MFKDF.

Finally, several works address the related but distinct problem of multi-factor authenticated key exchange [19, 36, 41, 49]. Unlike key derivation, authenticated key exchange focuses on establishing a secure communication channel between two trusted parties in the presence of man-in-the-middle adversaries. Thus, multi-factor authenticated key exchange (MFAKE) is an effective replacement for password-authenticated key exchange (PAKE), but not for password-based key derivation functions (PBKDFs) as MFKDF is.

13 Discussion

While the concept of “multi-factor key derivation” may be apparent from the name itself, the combination of PBKDFs with MFA is non-trivial due to the dynamic nature of popular factors. In this paper, we overcome this challenge by converting dynamic factors into static keying material using data encrypted with the output key, essentially allowing secrets to be safely stored within the key derivation function itself. The key material for several factors can then be concatenated to generate a key via a standard KDF, or via secret sharing to produce flexible threshold and policy variants.

Despite this relatively simple structure, MFKDF has the potential to improve new and existing applications alike. Returning to our motivating example of password managers, these applications in particular stand to benefit, allowing secret keys to be derived from multiple authentication factors rather than just from passwords (or, in some cases, directly managed by users). Further, secure account recovery can now be facilitated even if a recovery key was not previously stored on a trusted user device. Moreover, the threat of credential stuffing is significantly reduced; while attackers require just a single attempt to check a compromised credential against a PBKDF-encrypted ciphertext, MFKDF leaves one or more entire factors intact which must be separately cracked.

13.1 Limitations

There are several distinct scenarios in which an MFKDF-derived key may be compromised, even if correctly implemented with secure primitives. First, it is expected that an MFKDF-derived key is only as secure as its underlying factors, and that these factors must themselves be properly managed and protected. For example, if an SMS device is vulnerable to a SIM-swapping attack, then the corresponding MFKDF OOBA factor may also be at risk.

Further, if a combination of factors with insufficient entropy is chosen, brute-force attacks may still be feasible. As noted in §9, MFKDF-derived keys inherit the entropy of the weakest combination of factors allowed to derive them.

What is perhaps less obvious is that a combination of these two attacks may also be feasible. For example, while an MFKDF key combining a password and OTP factor may itself be infeasible to crack, the compromise of the password factor may allow the remaining factor to be defeated by brute-force.

Finally, the key could be compromised if it is not properly managed after derivation (e.g., via spyware or a browser vulnerability), as is the case with PBKDF-derived keys.

13.2 Future Work

Applications. In §2, we described the wide variety of systems currently depending on PBKDFs. Many of these systems can be enhanced by the use of MFKDF. We hope to perhaps see corporate networking protocols encrypting traffic using all authentication factors, or enterprise operating systems supporting disk encryption based on MFKDF. Of course, many password managers or cloud storage services would benefit greatly from MFKDF. As illustrated in §10.2, MFKDF also has potential utility in decentralized applications. Overall, we look forward to seeing many systems, new and existing, either enhanced or made possible by MFKDF.

Factors. While the constructions given in §5 account for the vast majority of current MFA usage [4], there remain important authentication methods for which factor constructions are not currently known. Chief among these are intrinsic factors like biometrics, geolocation, device fingerprinting, and behavioral authentication. Deriving key material based on single sign-on, in particular via OIDC, would also be very helpful. We hope to see specific constructions for these, as well as "general" factor constructions, perhaps based on MPC or trusted hardware, for arbitrary factors.

Extensions. Further, we suggest a number of theoretical extensions to MFKDF that would further extend its utility. First, while it was not our focus here, we suggest investigating the use of MFKDF as a simple replacement to password hashing. Next, we suggest extensions improving the forward-security of MFKDF. We also suggest integrating MFKDF with encrypted databases to facilitate useful applications on MFKDF-encrypted data, including the sharing of encrypted objects between multiple users. Lastly, we hope to see works illustrating the backwards-compatibility of MFKDF with unmodified PBKDF-based systems, such as via browser extensions.

Usability. Finally, while many aspects of our design were motivated by usability, no user study has yet been performed. Future work should therefore focus on empirically comparing the usability of MFKDF with that of PBKDFs.

14 Conclusion

PBKDFs continue to play an outsized role in widely-used applied cryptographic systems compared to the research attention they receive. MFKDF offers a multi-axis improvement over PBKDFs, providing better security and additional functionality with no changes to the authentication factors nor noticeable impact on the UI or UX of the end user. It represents a fast, flexible, secure, and practical solution to the key management problem, with an emphasis on solving issues like factor recovery, authentication policy enforcement, plug-and-play compatibility with existing PBKDF-based systems, and support for existing widely-used authentication factors. In doing so, it changes the status quo of MFA from software verification to direct cryptographic assurance.

Acknowledgements

We thank Guru Vamsi Policharla, Deevashwer Rathee, Xiaoyuan Liu, Gonzalo Munilla Garrido, Julien Piet, Sriram Sridhar, Jens Ernstberger, and Sanjam Garg for their advice and feedback. We also thank our anonymous shepherd for their guidance. This work was supported in part by the National Science Foundation, by the National Physical Science Consortium, and by the Fannie and John Hertz Foundation. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the supporting entities.

Availability

The source code for a JavaScript library implementing all of the MFKDF features described in this paper is available at:

- github.com/multifactor/mfkdf

We invite interested parties to visit mfkdf.com for detailed documentation, tutorials, unit tests, and code coverage reports. The centralized demo, decentralized demo, and benchmarking scripts used in this paper are available at these repositories:

- github.com/multifactor/mfkdf-application-demo
- github.com/multifactor/mfkdf-wallet-demo
- github.com/multifactor/mfkdf-benchmark

References

[1] 1Password. 1Password Security Design, 2021.

[2] Akamai. 2020 state of the internet.

[3] Saif Al-Kuwari, James H Davenport, and Russell J Bradford. Cryptographic Hash Functions: Recent Design Trends and Security Notions.

[4] FIDO Alliance. State of authentication report, October 2017.

[5] Apple. iOS Security, May 2012.

[6] Lucas Ballard, Seny Kamara, and Michael K. Reiter. The Practical Subtleties of Biometric Key Generation.

[7] Elaine Barker. Recommendation for Key Management Part 1: General. Technical Report NIST SP 800-57pt1r4, National Institute of Standards and Technology, January 2016.

[8] Elaine Barker and John Kelsey. Recommendation for Random Number Generation Using Deterministic Random Bit Generators. Technical Report NIST Special Publication (SP) 800-90A Rev. 1, National Institute of Standards and Technology, June 2015.

[9] Juan Benet. IPFS - Content Addressed, Versioned, P2P File System, July 2014. arXiv:1407.3561 [cs].

[10] 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.

[11] BITMAIN. ANTMINER Products.

[12] Joseph Bonneau, Elie Bursztein, Ilan Caron, Rob Jackson, and Mike Williamson. Secrets, Lies, and Account Recovery: Lessons from the Use of Personal Knowledge Questions at Google. In *Proceedings of the 24th International Conference on World Wide Web*, pages 141–150, Florence Italy, May 2015. International World Wide Web Conferences Steering Committee.

[13] Elie Burzstein and Jean Michel Picod. Recovering windows secrets and efs certificates offline. In *Proceedings of the 4th USENIX Conference on Offensive Technologies*, WOOT'10, page 1–8, USA, 2010. USENIX.

[14] 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, 2015.

[15] Anders Dalskov, Daniele Lain, Enis Ulqinaku, Kari Kostiainen, and Srdjan Capkun. 2FE: Two-Factor Encryption for Cloud Storage, October 2020. arXiv:2010.14417 [cs].

[16] Dashlane. Security white paper, March 2021.

[17] Ittay Eyal. On cryptocurrency wallet design. Cryptology ePrint Archive, Paper 2021/1522, 2021.

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

[19] Nils Fleischhacker, Mark Manulis, and Amir Azodi. A Modular Framework for Multi-Factor Authentication and Key Exchange. In Liqun Chen and Chris Mitchell, editors, *Security Standardisation Research*, volume 8893, pages 190–214. Springer International Publishing, Cham, 2014. Series Title: Lecture Notes in Computer Science.

[20] Dinei Florencio and Cormac Herley. A Large Scale Study of Web Password Habits. Technical Report MSR-TR-2006-166, Microsoft, November 2006.

[21] Ameya Hanamsagar, Simon S Woo, Christopher Kanich, and Jelena Mirkovic. How Users Choose and Reuse Passwords.

[22] Tony Hansen and Donald E. Eastlake 3rd. US Secure Hash Algorithms (SHA and SHA-based HMAC and HKDF). Request for Comments RFC 6234, Internet Engineering Task Force, May 2011.

[23] HYPR. 78% of People Reset a Password They Forgot in Past 90 Days.

[24] Ping Identity. What is Out-of-Band Authentication (OOBA)?

[25] IEEE. Ieee standard for wireless lan medium access control (mac) and physical layer (phy) specifications. *IEEE Std 802.11-1997*, 1997.

[26] IEEE. Ieee standard for information technology-telecommunications and information exchange between systems-local and metropolitan area networks-specific requirements-part 11: Wireless lan mac and phy specifications: Amendment 6: Mac security enhancements. *IEEE Std 802.11i-2004*, pages 1–190, 2004.

[27] ISO/IEC. IT Security techniques – Entity authentication – Part 2: Mechanisms using authenticated encryption. Standard, International Organization for Standardization, Geneva, CH, June 2019.

[28] Stanislaw Jarecki, Aggelos Kiayias, Hugo Krawczyk, and Jiayu Xu. Highly-efficient and composable password-protected secret sharing (or: How to protect your bitcoin wallet online). Cryptology ePrint Archive, Paper 2016/144, 2016.

[29] 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.

[30] Hugo Krawczyk. Cryptographic Extraction and Key Derivation: The HKDF Scheme. In *Advances in Cryptology – CRYPTO 2010*, volume 6223, pages 631–648. Springer Berlin Heidelberg, Berlin, Heidelberg, 2010. Series Title: Lecture Notes in Computer Science.

[31] Hugo Krawczyk, Mihir Bellare, and Ran Canetti. HMAC: Keyed-Hashing for Message Authentication. Request for Comments RFC 2104, Internet Engineering Task Force, February 1997.

[32] Hugo Krawczyk and Pasi Eronen. HMAC-based Extract-and-Expand Key Derivation Function (HKDF). Request for Comments RFC 5869, Internet Engineering Task Force, May 2010. Num Pages: 14.

[33] ENS Labs. Ethereum Name Service (ENS).

[34] Protocol Labs. InterPlanetary Name System (IPNS).

[35] LastPass. Our Zero-Knowledge Security Model.

[36] 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*, pages 255–267, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

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

[38] NIST. FIPS 197, Advanced Encryption Standard (AES).

[39] Kaiti Norton. LastPass: Is it a Safe Password Manager?, July 2021.

[40] Nick Parker and Stephen Lombardo. Enhancing Password Based Key Derivation Techniques. *PasswordsCon 2014*, page 26, 2014.

[41] 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*, pages 277–295, Berlin, Heidelberg, 2008. Springer.

[42] 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.

[43] Minhye Seo, Jong Hwan Park, Youngsam Kim, Sangrae Cho, Dong Hoon Lee, and Jung Yeon Hwang. Construction of a New Biometric-Based Key Derivation Function and Its Application. *Security and Communication Networks*, 2018:1–14, December 2018.

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

[45] Joe Siegrist. LastPass Hacked - Identified Early & Resolved, 2015.

[46] Colin Soutar, Danny Roberge, Alex Stoianov, Rene M. Gilroy, and B. V. K. Vijaya Kumar. Biometric encryption using image processing. In *Electronic Imaging*, 1998.

[47] Karim Toubba. Notice of Recent Security Incident, December 2022.

[48] Twilio. How to Send an Email to Text Message.

[49] Mariano Luis T. Uymatiao and William Emmanuel S. Yu. Time-based otp authentication via secure tunnel (toast): A mobile totp scheme using tls seed exchange and encrypted offline keystore. In *2014 4th IEEE International Conference on Information Science and Technology*, pages 225–229, 2014.

[50] Erkam Uzun, Carter Yagemann, Simon Chung, Vladimir Kolesnikov, and Wenke Lee. Cryptographic Key Derivation from Biometric Inferences for Remote Authentication. In *Proceedings of the 2021 ACM Asia Conference on Computer and Communications Security*, ASIA CCS '21, pages 629–643, New York, NY, USA, May 2021. Association for Computing Machinery.

[51] 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.

[52] 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.

[53] Yubico. Yubikey: Strong two-factor authentication.

A Algorithms

A.1 Formalization

Let an MFKDF *FactorInstance* F_i be a tuple $(\sigma_F \in \{0,1\}^{\ell_F}, (key \mapsto \alpha_{F,i}))$ representing the ith derivation of a given factor, where σ_F is the factor-specific key material (static), ℓ_F is the number of bits of entropy in F, and $(key \mapsto \alpha_{F,i})$ is a function that takes the final derived *key* and returns the factor policy $(\alpha_{F,i})$, a tuple whose contents vary by factor.

An MFKDF factor construction consists of a *FactorSetup* function, which instantiates a factor given a configuration cfg_F (which varies by factor), and a *FactorDerive* function, which derives a factor from an instance-specific $W_{F,i} \in \{0,1\}^{\ell_F}$ and a $\alpha_{F,i}$ output by *FactorSetup* or a previous derivation:

$$\begin{aligned}
&FactorSetup: \\
&\quad cfg_F \mapsto FactorInstance\ F_0 \\
&FactorDerive: \\
&\quad W_{F,i}, \alpha_{F,i} \mapsto FactorInstance\ F_{i+1}
\end{aligned}$$

Similarly, let an MFKDF *KeyInstance* K_i be a tuple $(\alpha_{K,i}, key \in \{0,1\}^{\ell_K})$ representing the ith derivation of a key, where *key* is the derived key (static), ℓ_K is the size of the derived key in bits, and $\alpha_{K,i}$ varies.

An MFKDF construction consists of a *KeySetup* function, which instantiates a key of size ℓ_K given an array of *FactorInstance*s, and a *KeyDerive* function, which derives an established key from an array of *FactorInstance*s and a $\alpha_{K,i}$:

$$\begin{aligned}
&KeySetup: \\
&\quad \ell_K, FactorInstance[]\ FS_0 \mapsto KeyInstance\ K_0 \\
&KeyDerive: \\
&\quad FactorInstance[]\ FS_i, \alpha_{K,i} \mapsto KeyInstance\ K_{i+1}
\end{aligned}$$

In §A.2, we will provide MFKDF factor constructions for several popular authentication factors. In §A.3, we will provide a basic MFKDF construction, and in §A.4, we will provide a threshold MFKDF construction. We use $\odot$ for bitstring and array concatenation, and $\oplus$ for bitwise XOR.

A.2 Factor Constructions

Algorithm 1 Factor Construction for Constant Factors

function SETUP($cfg_F : (W_F \in \{0,1\}^{\ell_F})$)
    $\sigma_F \leftarrow transform(W_F)$
    return $(key \to (), \sigma_F)$
end function
function DERIVE($W_F \in \{0,1\}^{\ell_F}, \alpha_{F,i}$)
    $\sigma_F \leftarrow transform(W_F)$
    return $(key \to (), \sigma_F)$
end function

We begin with the factor construction for constant factors like passwords, security questions, and recovery codes, shown in algorithm 1. This is by far the simplest factor construction, with just an optional *transform* step between the input W and output σ_F. The *transform* step can be used, for example, to standardize the case of a security answer, or to extract useful data from a UUIDv4 recovery code.

Algorithm 2 Factor Construction for HMAC-SHA1

Require: Let *HS1* be HMAC-SHA1 per RFC 2014 [31].
function SETUP($cfg_F : (hmackey_F \in \{0,1\}^{160})$)
    $challenge_{F,0} \leftarrow \{0,1\}^{160}$
    $W_{F,0} \leftarrow HS1(hmackey_F, challenge_{F,0})$
    $pad_{F,0} \leftarrow W_{F,0} \oplus hmackey_F$
    $\alpha_{F,0} \leftarrow (challenge_{F,0}, pad_{F,0})$
    return $(key \to \alpha_{F,0}, hmackey_F)$
end function
function DERIVE($W_{F,i} \in \{0,1\}^{160}, \alpha_{F,i}$)
    $(challenge_{F,i}, pad_{F,i}) \leftarrow \alpha_{F,i}$
    $hmackey_F \leftarrow W_{F,i} \oplus pad_{F,i}$
    $challenge_{F,i+1} \leftarrow \{0,1\}^{160}$
    $response_{F,i+1} \leftarrow HS1(hmackey_F, challenge_{F,i+1})$
    $pad_{F,i+1} \leftarrow challenge_{F,i+1} \oplus hmackey_F$
    $\alpha_{F,i+1} \leftarrow (challenge_{F,i+1}, pad_{F,i+1})$
    return $(key \to \alpha_{F,i+1}, hmackey_F)$
end function

Algorithm 2 shows the MFKDF factor construction for HMAC-SHA1 challenge-response, an authentication method implemented by many hardware tokens such as YubiKeys. Since the $challenge_{F_i}$ and the corresponding response ($response_{F,i+1}$) are in effect uniformly random and non-repeating, they can be used as a one-time pad for the $hmackey_F$ (which is itself the σ_F) with information theoretic security. The HMAC-SHA1 factor fixes $\ell_F = 160$ due to the output of SHA1 being exactly 20 bytes.

Algorithm 3 Factor Construction for OOBA

Require: Let (Enc, Dec) be public-key encryption.
function SETUP($cfg_F : (d, pk_F)$)
    $target_F \leftarrow \mathbb{N} \cup [0, 10^d)$
    $otp_{F,0} \leftarrow \mathbb{N} \cup [0, 10^d)$
    $offset_{F,0} \leftarrow (target_F - otp_{F,0})\ \%\ 10^d$
    $ct_{F,0} \leftarrow Enc(otp_{F,0}, pk_F)$
    return $(key \to (d, pk_F, ct_{F,0}, offset_{F,0}), target_F)$
end function
function DERIVE($W_{F,i} \in \mathbb{N} \cup [0, 10^d), \alpha_{F,i}$)
    $(d, pk_F, ct_{F,i}, offset_{F,i}) \leftarrow \alpha_{F,i}$
    $target_F \leftarrow (offset_{F,i} + W_{F,i})\ \%\ 10^d$
    $otp_{F,i+1} \leftarrow \mathbb{N} \cup [0, 10^d)$
    $offset_{F,i+1} \leftarrow (target_F - otp_{F,i+1})\ \%\ 10^d$
    $ct_{F,i+1} \leftarrow Enc(otp_{F,i+1}, pk_F)$
    $\alpha_{F,i+1} \leftarrow (d, pk_F, ct_{F,i+1}, offset_{F,i+1})$
    return $(key \to \alpha_{F,i+1}, target_F)$
end function

Algorithm 3 shows the MFKDF factor construction for out-of-band authentication (OOBA) factors such as email or SMS. Unlike all other factors presented herein, OOBA factors are not perfectly trustless; they inherently depend on the honesty of the underlying channel (e.g., email provider, phone carrier). The OOBA factor takes advantage of this by pre-encrypting a numeric OTP of d digits using the public key of the channel. The modular addition of the OTP with a fixed $target_F$ provides the same information theoretic security as the one-time pad for the HMAC-SHA1 factor.

Algorithm 4 Factor Construction for HOTP

Require: Let *HOTP* be HOTP per RFC 4226 [51].
Require: Let (Enc, Dec) be symmetric-key encryption.

function SETUP($cfg_F : (d, hotpkey_F \in \{0,1\}^{\ell_F})$)
  $target_F \leftarrow \mathbb{N} \cup [0, 10^d)$
  $otp_{F,0} \leftarrow HOTP(hotpkey_F, 1) \ \% \ 10^d$
  $offset_{F,0} \leftarrow (target_F - otp_{F,0}) \ \% \ 10^d$
  function POLICY(key)
    $ct_F \leftarrow Enc(hotpkey_F, key)$
    return $(d, 1, offset_{F,0}, ct_F)$
  end function
  return (POLICY, $target_F$)
end function
function DERIVE($W_{F,i} \in \mathbb{N} \cup [0, 10^d), \alpha_{F,i}$)
  $(d, ctr_{F,i}, offset_{F,i}, ct_F) \leftarrow \alpha_{F,i}$
  $target_F \leftarrow (offset_{F,i} + W_{F,i}) \ \% \ 10^d$
  $ctr_{F,i+1} \leftarrow ctr_{F,i} + 1$
  function POLICY(key)
    $hotpkey_F \leftarrow Dec(ct_F, key)$
    $otp_{F,i+1} \leftarrow HOTP(hotpkey_F, ctr_{F,i+1}) \ \% \ 10^d$
    $offset_{F,i+1} \leftarrow (target_F - otp_{F,i+1}) \ \% \ 10^d$
    return $(d, ctr_{F,i+1}, offset_{F,i+1}, ct_F)$
  end function
  return (POLICY, $target_F$)
end function

Algorithms 4 and 5 show MFKDF factor constructions for HOTP and TOTP respectively, each supporting a variable number of OTP digits d (typically, $d = 6$). HOTP is the first factor to take advantage of the key feedback mechanism described in §4, allowing the $hotpkey_F$ to be securely embedded within α_F and used to set up F_{i+1} during the derivation of F_i by incrementing $ctr_{F,i}$. The TOTP and HOTP constructions takes advantage of the same information theoretic blinding via modular arithmetic as is used in the OOBA factor. The TOTP construction is similar to HOTP, with the addition of a window parameter w. By default, we suggest $w = 87600$ cycles (30 days). Note that by definition, $TOTP(K) = HOTP(K, \lfloor (T - T_0)/T_X \rfloor)$. Per our construction, the *offset* value is calculated for every possible time between T and $T + wT_X$; this does not reveal anything useful to attackers due to the forward security of TOTP. For the OOBA, HOTP, and TOTP constructions presented above, $\ell_F = d/\log_{10} 2$ ($\ell_F \approx 20$ when $d = 6$).

Algorithm 5 Factor Construction for TOTP

Require: Let *HOTP* be HOTP per RFC 4226 [51].
Require: Let T, T_0, T_X be TOTP times per RFC 6238 [52].
Require: Let (Enc, Dec) be symmetric-key encryption.

function SETUP($cfg_F : (d, w, totpkey_F \in \{0,1\}^{\ell_F})$)
  $target_F \leftarrow \mathbb{N} \cup [0, 10^d)$
  $ctr_{F,0} \leftarrow \lfloor (T - T_0)/T_X \rfloor$
  for $j \leftarrow 0, w$ **do**
    $otp \leftarrow HOTP(totpkey_F, ctr_{F,0} + j) \ \% \ 10^d$
    $offsets_{F,0}[j] \leftarrow (target_F - otp) \ \% \ 10^d$
  end for
  function POLICY(key)
    $ct_F \leftarrow Enc(totpkey_F, key)$
    return $(d, w, ctr_{F,0}, offsets_{F,0}, ct_F)$
  end function
  return (POLICY, $target_F$)
end function
function DERIVE($W_{F,i} \in \mathbb{N} \cup [0, 10^d), \alpha_{F,i}$)
  $(d, w, ctr_{F,i}, offsets_{F,i}, ct_F) \leftarrow \alpha_{F,i}$
  $ctr_{F,i+1} \leftarrow \lfloor (T - T_0)/T_X \rfloor$
  $idx_{F,i} \leftarrow ctr_{F,i+1} - ctr_{F,i}$
  $offset_{F,i} \leftarrow offsets_{F,i}[idx_{F,i}]$
  $target_F \leftarrow (offset_{F,i} + W_{F,i}) \ \% \ 10^d$
  function POLICY(key)
    $hotpkey_F \leftarrow Dec(ct_F, key)$
    for $j \leftarrow 0, w$ **do**
      $otp \leftarrow HOTP(totpkey_F, ctr_{F,i+1} + j) \ \% \ 10^d$
      $offsets_{F,i+1}[j] \leftarrow (target_F - otp) \ \% \ 10^d$
    end for
    return $(d, w, ctr_{F,i+1}, offsets_{F,i+1}, ct_F)$
  end function
  return (POLICY, $target_F$)
end function

We also include the factor construction for the "stacked key" factor in algorithm 6, although its construction may already be evident from the symmetric definitions of an MFKDF construction and an MFKDF factor construction. The MFKDF input, *FactorInstance*[] *FS*, is provided to the SETUP and DERIVE functions as W_F; the *KeySetup* or *KeyDerive* function is invoked, and the derived key key_{K_F} is returned as the factor key material σ_F. Thus, $\ell_F = \ell_{K_F}$ here.

Algorithm 6 Factor Construction for Key Stacking

function SETUP($cfg_F : (\ell_K, FactorInstance[]\ FS)$)
  $KeyInstance\ K_{F,0} \leftarrow KeySetup(\ell_K, FS)$
  $(\alpha_{K_F,0}, key_{K_F}) \leftarrow K_{F,0}$
  return $(key \rightarrow \alpha_{K_F,0}, key_{K_F})$
end function
function DERIVE($FactorInstance[]\ FS_F, \alpha_{F,i}$)
  $KeyInstance\ K_{F,i+1} \leftarrow KeyDerive(FS_F, \alpha_{F,i})$
  $(\alpha_{K_F,i+1}, key_{K_F}) \leftarrow K_{F,i+1}$
  return $(key \rightarrow \alpha_{K_F,i+1}, key_{K_F})$
end function

A.3 MFKDF Construction

Algorithm 7 is a simple n-of-n MFKDF construction. Each authentication factor is converted into factor material σ_F via its corresponding factor construction. The σ_F for all factors are concatenated to form σ_K, which in turn is used to derive the *key* via a PBKDF. The key feedback mechanism of §4 is then used to produce $\alpha_{K,i+1}$, making possible the use of dynamic factors like HOTP.

Algorithm 7 MFKDF Construction

Require: Let *KDF* be a hard PBKDF like Argon2 [10].

function SETUP($\ell_K, \mathit{FactorInstance}[]\ FS_0$)
  $\mathit{salt}_K \leftarrow \{0,1\}^{\ell_K}$
  $\sigma_K \leftarrow \varepsilon$
  $\mathit{fns}_{K,0} \leftarrow []$
  $\mathit{fps}_{K,0} \leftarrow []$
  for all $\mathit{FactorInstance}\ F \in FS_0$ **do**
    $(\sigma_F, \mathit{fn}_F) \leftarrow F$
    $\sigma_K \leftarrow \sigma_K \odot \sigma_F$
    $\mathit{fns}_{K,0} \leftarrow \mathit{fns}_{K,0} \odot \mathit{fn}_F$
  end for
  $\mathit{key} \leftarrow \mathit{KDF}(\ell_K, \sigma_K, \mathit{salt}_K)$
  for all $\mathit{fn}_{K,0} \in \mathit{fns}_{K,0}$ **do**
    $\alpha_{F,0} \leftarrow \mathit{fn}_{K,0}(\mathit{key})$
    $\mathit{fps}_{K,0} \leftarrow \mathit{fps}_{K,0} \odot \alpha_{F,0}$
  end for
  return $((\ell_K, \mathit{salt}_K, \mathit{fps}_{K,0}), \mathit{key})$
end function
function DERIVE($\mathit{FactorInstance}[]\ FS_i, \alpha_{K,i}$)
  $(\ell_K, \mathit{salt}_K, \mathit{fps}_{K,i}) \leftarrow \alpha_{K,i}$
  $\sigma_K \leftarrow \varepsilon$
  $\mathit{fns}_{K,i} \leftarrow []$
  $\mathit{fps}_{K,i+1} \leftarrow []$
  for all $\mathit{FactorInstance}\ F \in FS_i$ **do**
    $(\sigma_F, \mathit{fn}_F) \leftarrow F$
    $\sigma_K \leftarrow \sigma_K \odot \sigma_F$
    $\mathit{fns}_{K,i} \leftarrow \mathit{fns}_{K,i} \odot \mathit{fn}_F$
  end for
  $\mathit{key} \leftarrow \mathit{KDF}(\ell_K, \sigma_K, \mathit{salt}_K)$
  for all $\mathit{fn}_{K,i} \in \mathit{fns}_{K,i}$ **do**
    $\alpha_{F,i+1} \leftarrow \mathit{fn}_{K,i}(\mathit{key})$
    $\mathit{fps}_{K,i+1} \leftarrow \mathit{fps}_{K,i+1} \odot \alpha_{F,i+1}$
  end for
  return $((\ell_K, \mathit{salt}_K, \mathit{fps}_{K,i+1}), \mathit{key})$
end function

A.4 Threshold MFKDF Construction

Algorithm 8 implements a t-of-n threshold MFKDF. It expands upon the simple MFKDF construction by adding a threshold parameter t. The key material (σ_K) is split into n shares via Shamir's secret sharing [44], which are then padded by factor keys derived using HKDF [32]. The one-time-pad can also be replaced by symmetric-key encryption as long as no checksums or integrity mechanisms are included in the scheme. Thus, if at least t factors are provided, t shares can be derived from their pads in α_K and thus σ_K can be recovered.

Algorithm 8 Threshold MFKDF Construction

Require: Let *KDF* be a hard PBKDF like Argon2 [10].
Require: Let *HKDF* be HKDF per RFC 5869 [32].
Require: Let $(\mathit{Share}, \mathit{Comb})$ be secret sharing (SSS) [44].

function SETUP($t, \ell_K, \mathit{FactorInstance}[]\ FS_0$)
  $\sigma_K \leftarrow \{0,1\}^{\ell_K}$
  $\mathit{salt}_K \leftarrow \{0,1\}^{\ell_K}$
  $\mathit{key} \leftarrow \mathit{KDF}(\ell_K, \sigma_K, \mathit{salt}_K)$
  $\mathit{shares}_{K,0} \leftarrow \mathit{Share}(\sigma_K, t, \mathit{len}(FS_0))$
  $\mathit{fps}_{K,0} \leftarrow []$
  $\mathit{fs}_K \leftarrow []$
  for all $\mathit{FactorInstance}\ F \in FS_0$ **do**
    $(\sigma_F, \mathit{fn}_F) \leftarrow F$
    $\mathit{pad}_F \leftarrow \mathit{HKDF}(\ell_K, \sigma_F, \varepsilon, \varepsilon)$
    $\mathit{share}_F \leftarrow \mathit{pad}_F \oplus \mathit{shares}_{K,0}[i]$
    $\mathit{fs}_K \leftarrow \mathit{fs}_K \odot \mathit{share}_F$
    $\alpha_{F,0} \leftarrow \mathit{fn}_F(\mathit{key})$
    $\mathit{fps}_{K,0} \leftarrow \mathit{fps}_{K,0} \odot \alpha_{F,0}$
  end for
  return $((t, \ell_K, \mathit{salt}_K, \mathit{fps}_{K,0}, \mathit{fs}_K), \mathit{key})$
end function
function DERIVE($\mathit{FactorInstance}[]\ FS_i, \alpha_{K,i}$)
  $(t, \ell_K, \mathit{salt}_K, \mathit{fps}_{K,i}, \mathit{fs}_K) \leftarrow \alpha_{K,i}$
  $\mathit{shares}_{K,i} \leftarrow []$
  $\mathit{fns}_{K,i} \leftarrow []$
  $\mathit{fps}_{K,i+1} \leftarrow \mathit{fps}_{K,i}$
  for all $\mathit{FactorInstance}\ F \in FS_i$ **do**
    $(\sigma_F, \mathit{fn}_F) \leftarrow F$
    $\mathit{pad}_F \leftarrow \mathit{HKDF}(\ell_K, \sigma_F, \varepsilon, \varepsilon)$
    $\mathit{share}_F \leftarrow \mathit{pad}_F \oplus \mathit{fs}_K[j]$
    $\mathit{shares}_{K,i} \leftarrow \mathit{shares}_{K,i} \odot \mathit{share}_F$
    $\mathit{fns}_{K,i} \leftarrow \mathit{fns}_{K,i} \odot \mathit{fn}_F$
  end for
  $\sigma_K \leftarrow \mathit{Comb}(\mathit{shares}_{K,i}, t, \mathit{len}(\mathit{fs}_{K,i}))$
  $\mathit{key} \leftarrow \mathit{KDF}(\ell_K, \sigma_K, \mathit{salt}_K)$
  for all $\mathit{fn}_{K,i} \in \mathit{fns}_{K,i}$ **do**
    $\alpha_{F,i+1} \leftarrow \mathit{fn}_{K,i}(\mathit{key})$
    $\mathit{fps}_{K,i+1}[j] \leftarrow \alpha_{F,i+1}$
  end for
  return $((t, \ell_K, \mathit{salt}_K, \mathit{fps}_{K,i+1}, \mathit{fs}_K), \mathit{key})$
end function

A.5 Policy MFKDF Construction

Algorithm 9 Policy MFKDF Construction

function SETUP($t, \ell_K, \mathit{FactorInstance}[][]\ P$)
  $\mathit{FactorInstance}[]\ R \leftarrow []$
  for all $\mathit{FactorInstance}[]\ C \in P$ **do**
    $SK \leftarrow$ ALG6SETUP$((\ell_K, C))$
    $R \leftarrow R \odot SK$
  end for
  return ALG8SETUP$(1, \ell_K, R)$
end function