TMPS: Ticket-Mediated Password Strengthening

John Kelsey
National Institute of Standards and Technology,
Gaithersburg, MD, USA
Department of Electrical Engineering, ESAT/COSIC
KU Leuven, Belgium

Dana Dachman-Soled*
Department of Electrical and Computer Engineering
University of Maryland
College Park, MD, USA

Sweta Mishra
National Institute of Standards and Technology
Gaithersburg, MD, USA

Meltem Sönmez Turan
National Institute of Standards and Technology
Gaithersburg, MD, USA

ABSTRACT

We introduce the notion of Ticket-Mediated Password Strengthening (TMPS), a technique for allowing users to derive keys from passwords while imposing a strict limit on the number of guesses of their password any attacker can make, and strongly protecting the users' privacy. We describe the security requirements of TMPS, and then a set of efficient and practical protocols to implement a TMPS scheme, requiring only hash functions, CCA2-secure encryption, and blind signatures. We provide several variant protocols, including an offline symmetric-only protocol that uses a local trusted computing environment, and online variants that use group signatures or stronger trust assumptions instead of blind signatures. We formalize the security of our scheme by defining an ideal functionality in the Universal Composability (UC) framework, and by providing game-based definitions of security. We prove that our protocol realizes the ideal functionality in the random oracle model (ROM) under adaptive corruptions with erasures, and prove that security with respect to the ideal/real definition implies security with respect to the game-based definitions.

1 INTRODUCTION

In many real-world cryptosystems, the user's password is the greatest practical weakness. Commonly, the user enters a memorized password or passphrase, which is processed using a password-based key derivation function, to get a symmetric key. Knowledge of this key allows the attacker to completely violate the security of a system he has physically compromised–to read the user's files, sign arbitrary things with her private key, spend her bitcoins, etc.

Deriving a key from a password is an old, well-studied problem[19, 26, 31]. The practical difficulty comes from the ability of an attacker to run a parallel password search. After stealing the user's device, the attacker is able to run an offline attack on a large number of machines/processors, perhaps trying billions of different password guesses per second until the user's password is found. Few users can memorize a password capable of withstanding such an attack for long.

In this paper, we propose a novel method to approach this problem, called Ticket-Mediated Password Strengthening (TMPS). Informally, this works as follows: When the user wants to produce a new password-derived key, she runs a protocol with a server to produce a set of t *tickets*–bitstrings which she stores locally. Later, when she wants to unlock this key using her password, she runs another protocol with the server, providing (and discarding) one of these tickets. The password can only be used to unlock the user's key with the server's help, and the server will not provide this help without a ticket that has never before been used.

The result is that the user can establish a hard limit on the number of possible guesses of the password any attacker can make–if she has only 100 tickets on her device, then an attacker who compromises the device can never try more than 100 guesses of her password.

Our approach provides strong privacy guarantees for the user with respect to the server, and unlike many proposed password-hardening schemes in the literature, is focused on user-level key derivation problems, rather than on online authentication to some web service. We also provide a mechanism to allow the server to control which users it provides services to, without violating the users' privacy.

Among other advantages, our scheme gives users a security metric that is human-meaningful–the maximum number of guesses the attacker can ever have against their password. Hardness parameters of password hashes, or entropy estimates of a password, are meaningful only to security experts; the maximum number of attacker guesses that will be allowed is much easier to understand. On the other hand, our scheme imposes the need to be online in order to unlock a key secured by a password. (Though we provide variant schemes which can be used offline.)

1.1 Our Results

- We introduce the notion of Ticket-Mediated Password Strengthening (TMPS), a mechanism for allowing users to derive keys from passwords while imposing a strict limit on the number of guesses of their password any attacker can make, and strongly protecting the users' privacy.
- We formalize the security requirements of our new notion of TMPS, both by introducing game-based definitions (See Appendix B) as well as defining a corresponding ideal functionality in the Universal Composability (UC) framework (See Section 5).
- We present efficient protocols realizing our new notion. Our basic protocol requires only hash functions, CCA2-secure encryption, and blind signatures (See Section 4).

*Supported in part by NSF grants #CNS-1840893, #CNS-1453045 (CAREER), by a research partnership award from Cisco and by financial assistance award 70NANB15H328 from the U.S. Department of Commerce, National Institute of Standards and Technology.

- We prove that our protocol UC-realizes the aforementioned ideal functionality in the random oracle model (ROM) under adaptive corruptions with erasures (See Appendix C) and prove that security with respect to the Ideal/Real definition implies security with respect to the game-based definitions (See Appendix B).
- We present several variants of our protocol, including an offline version of our protocol using a local hardware security module (HSM) or trusted execution environment, and variants that make use of group signatures, proofs of work, or weaker security assumptions to ensure user privacy while still preventing overuse of server resources (See Section 6).
- Finally, we discuss efficient implementations and performance, in Section 7, and consider some questions left open by this research in Section 8.

1.2 Related Work

A long list of work (e.g., [5, 6, 32]) focuses on Password Authenticated Key Exchange (PAKE) protocols where the user and server share a password to securely establish a session key. These protocols have some similarities to our scheme (in particular the use of a trusted server), but their details (key generation) and usage are quite different. Typically, the PAKE protocols are vulnerable to offline dictionary attacks after server compromise, though this problem has been tackled in recent proposals such as the Opaque protocol [17]. Another related concept known as Password Protected Secret Sharing (PPSS) as described in [16], requires many servers to hold some share of a key; a secret sharing scheme is used to reconstruct them, secured by a password.

Many significant works on password security focus on password-based authentication systems. In [25], Mani describes a scheme that uses a server to assist in password hashing, but without any concern for user privacy–the goal in that scheme was to harden the password file by incorporating a pseudorandom function (PRF) computed on a single-purpose machine. Similarly, [2, 27, 28] describe a scheme with a separately-stored secret key in a crypto-server to strengthen password hashing, an informal description of the concept of password hardening, later formally defined in [13, 23, 29]. Our scheme differs in its goal;but is related to *password hardening* in using a separate server or device, in order to limit password guessing attacks.

A proposal by Lai et al. [22] defines a *password hardening encryption* scheme which provides substantial protection from password-cracking attacks by rate-limiting password cracking attempts assuming the crypto server is not compromised. Our scheme differs in many ways, most notably in the use of tickets for decryption, and in the assurance of user privacy. Also, our scheme focuses on the setting of password-based key derivation on the user's device, rather than on a server the user trusts with her data.

Still another line of work introduces the notion of password-based threshold authentication [1] for token-based authentication in single sign-on setting–in their scheme, any subset of ℓ of n servers participate in verifying the user's password and generating a token.

There is also a rich literature in server-assisted computation [3, 14, 20, 21] which preserves the user's privacy. Our scheme differs from most of this work in that we are not trying to offload computational work to the server, we are just using a server to limit the number of times some computation may be done. In our proposal, we use standard algorithms, and provide a great deal more flexibility to choose the underlying cryptographic functions than is available in these schemes.

2 PRELIMINARIES

2.1 Notation

Let $k \in \mathbb{N}$. The set of bitstrings of length k is denoted as $\{0,1\}^k$. The concatenation of two bitstrings x and y is denoted by $x \parallel y$. The exclusive-OR of two bitstrings x and y of same length is denoted as $x \oplus y$. We let b^k denote the string with k successive repetitions of bit b. If $\mathcal{X}$ is a set, we let $x \leftarrow\!\!\$\, \mathcal{X}$ denote sampling a uniformly random element x from $\mathcal{X}$. The security parameter is denoted by $n \in \mathbb{N}$. Unless otherwise specified, we assume all symmetric keys and hash outputs to be n bits in length.

2.2 Underlying Primitives and Functions

We use the following primitives in our protocols:

- $\mathsf{HASH}(X)$: The cryptographic hash of input X.
- $\mathsf{HMAC}(K, X)$: The HMAC of X under key K.
- $\mathsf{PH}(S, P)$: Hash of the password P using salt S.
- $\mathsf{KDF}(K, D, \ell)$: ℓ-bit key derived from the secret value K and public value D.
- $\Pi_{ENC} := (\mathsf{GEN}, \mathsf{ENC}, \mathsf{DEC})$: An encryption system where $\mathsf{ENC}(K, X)$ is encryption of plaintext X under the key K, and $\mathsf{DEC}(K, Y)$ is decryption of ciphertext Y under the key K.
- $\Pi_{BSIG} := (\mathsf{GEN}, \mathsf{BLIND}, \mathsf{UBLIND}, \mathsf{SIGN}, \mathsf{BVERIFY})$: A 2-move blind signature scheme where
  - $M^* \leftarrow \mathsf{BLIND}(M)$: The user blinds the message M to obtain M^* and sends to the signer.
  - $\sigma^* \leftarrow \mathsf{SIGN}_{SK}(M^*)$: The signer outputs a signature σ^* on input of message M^* and private key SK and sends to the user.
  - $F \leftarrow \mathsf{UBLIND}(\sigma^*)$: The user unblinds the signature σ^* to obtain F. Note that the user inputs additional private state to the UBLIND algorithm, which we leave implicit.
  - $\mathsf{BVERIFY}_{PK}(M, F)$: Verification of signature F on message M under public key PK as valid/invalid.

Next, we define two internal functions: $\mathsf{VE}(D, K_P)$ provides verifiable encryption of K_P with D and $\mathsf{DV}(D, Z)$ decrypts K_P after checking the correctness of D. Both functions assume that D, K_P and hash outputs are n bits long.

We remark that we use the special-purpose verifiable encryption scheme define here, as opposed to using a generic authenticated encryption scheme, for two reasons: First, our UC security proof requires use of a random oracle call here to allow for programmability;

Algorithm 1 Verifiably encrypt K_P with D.

```
function VE(D, K_P)
    Z ← HASH(0 || D) || (HASH(1 || D) ⊕ K_P)
    return(Z)
```

Algorithm 2 Verifiably decrypt Z with D.

```
function DV(D, Z)
    X ← Z_{0...n-1}
    Y ← Z_{n...2n-1}
    X* = HASH(0 || D)
    if X == X* then
        return(HASH(1 || D) ⊕ Y)
    else
        return(⊥)
```

second, what is required here is not quite authenticated encryption–we only care about whether the *key* is correct, not about whether the decrypted plaintext is correct, and we only encrypt *once* under any key.

3 TICKET-MEDIATED PASSWORD STRENGTHENING

3.1 The Setting and the Problem

It is common for a user to need to enter a password in order to unlock some encrypted storage, such as encrypted flash memory, an encrypted hard drive, or an individual encrypted file. Usually, this is done by having the user enter a password, and then applying some computationally expensive function such as PBKDF#2 or scrypt to derive a cryptographic key from the password and a locally-stored salt.

Unfortunately, an attacker who steals the user's device can typically copy the salt and sufficient data to check a password guess onto a large number of machines with a large number of processors, and run a *password-guessing* attack, in which each processor generates plausible password guesses and checks them as quickly as possible. Very few users can remember a password that will withstand this kind of attack run over a few days. Thus, an attacker who steals the user's laptop may be able to unlock her hard drive, or decrypt the signing keys used in her Bitcoin wallet, despite her use of encryption.

Ticket-Mediated Password Strengthening (TMPS) is a scheme for using a password to unlock a cryptographic key. It's a natural replacement for password-based key derivation–e.g., feeding a locally-stored salt and a user-entered password into scrypt to derive an encryption key. The natural use case is using a password to access encrypted storage. However, TMPS can be used anytime a user enters a password to unlock a secret value, assuming the device has internet access.

The goal of TMPS is to provide a better solution to the problem of deriving or unlocking keys from passwords–one that massively reduces the power of password-guessing attacks.

3.2 TMPS Overview

In *Ticket-Mediated Password Strengthening* (TMPS, for short), the user[1] first interacts with a server to get a set of *tickets*. Each ticket entitles the user to assistance from the server with one attempt to unlock a master secret (called the *payload key*) using a password. Later, users (or anyone else with access to the tickets) may use the tickets to attempt to unlock the payload key using the password.

TMPS requires a setup phase, and two protocols: Request and Unlock. During setup, the server establishes public encryption and signing keys and makes them available to its users.

In order to get tickets, the user chooses a payload key and a password, and runs the Request protocol with the server, requesting t tickets. If the protocol terminates successfully, the user ends up with t tickets, each of which entitles her to one run of the Unlock protocol.

In order to use a password to unlock the payload key, the user must consume a ticket–she runs the Unlock protocol with the server, passing the server some information from the ticket and some information derived from her ticket and her password. When the protocol runs successfully, the user recovers the payload key.

The security requirements of a TMPS scheme are:

(1) Request
  - (a) The server learns nothing about the password or payload key from the Request protocol.
  - (b) There is no way to get a ticket the server will accept, except by running the Request protocol.
  - (c) Each ticket is generated for a specific password and payload key; tickets generated for one password and payload key give no help in unlocking or learning any other password or payload key.

(2) Unlock
  - (a) An Unlock run will be successful (it will return the correct K_P) if and only if:
    - (i) This ticket came from a successful run of the Request protocol.
    - (ii) This ticket has never been used in another Unlock call.
    - (iii) The same password used to request the ticket is used to Unlock it.
  - (b) From an unsuccessful run of the Unlock protocol, the user gains no information about the payload key.
  - (c) From an unsuccessful run of the Unlock protocol, the user learns (at most) that the password used to run the protocol was incorrect.
  - (d) The server learns nothing about the payload key or password from the Unlock protocol.
  - (e) The server learns nothing about which user ran the Unlock protocol with it at any given time.

Note that these requirements don't describe the generation of the payload key or the selection of the password. If the payload key is known or easily guessed, then TMPS can do nothing to improve the situation. In any real-world use, the payload key should

[1] For convenience, we refer to "the user" generating random values and running protocols in the rest of this paper when we really mean "software on the user's device." The user herself should only need to remember the password, and perhaps provide credentials to identify herself to the server when she requests new tickets.

be generated using a high-quality cryptographic random number generator.

The strength of the password matters for the security of ticket-mediated password strengthening, but in a very limited way–each run of UNLOCK consumes one ticket and allows the user to check one guess of the password. An attacker given N equally-likely passwords and t tickets thus has at most a t/N probability of successfully learning the password.

3.3 Discussion

The usual way password-based key derivation fails is that an offline attacker tries a huge number of candidate passwords, until he finally happens upon the user's password. He then derives the same key as the user derived, and may decrypt her files. A TMPS scheme avoids this attack by requiring the involvement of the server in each password guess, and (more importantly) by limiting the number of guesses that will ever be allowed. If the user of a TMPS scheme requests only 100 tickets from the server, then an attacker who compromises her machine and learns the tickets will never get more than 100 guesses of her password. If he cannot guess the password in his first 100 guesses, then he will never learn either the password or the payload key. Even if he is given the correct password after he has used up all the tickets, he cannot use that password to learn anything about the payload key.

The security of a TMPS scheme relies on the server being unwilling to allow anyone to reuse a ticket, and the inability of anyone to unlock a payload key with a password *without* running the UNLOCK protocol with a server, and consuming a fresh ticket in the process.

A corrupt server can weaken the security of TMPS, but only in limited ways. It cannot learn anything about the password or payload key. It cannot determine which user is associated with which ticket, or link REQUEST and UNLOCK runs. But it *can* enable an attacker who has already compromised a user's tickets to reuse those tickets as many times as he likes.

4 THE BASIC PROTOCOL

In this section, we describe a set of protocols that implement Ticket-Mediated Password Strengthening in a concrete way. Our protocols require a secure cryptographic hash function, a public key encryption scheme providing CCA2 security[2], and a blind signature scheme[3]. Our scheme has some similarities to an online anonymous e-cash scheme–notably in the need to reject attempts to "double-spend." However, the application and many of the details are quite different–each "coin" in our scheme is bound to a specific password hashing computation.

A *ticket* gives a user enough information to enlist the server in helping carry out one password-based key derivation. Each ticket contains an *inside* part (which the user retains and does not share with the server) and an *outside* part (which the user sends to the server). The different parts of a ticket are bound together with each other and with the specific password and key derivation being carried out, and can't be used for a different key derivation.

[2] An attacker who can alter a ciphertext to get a new valid ciphertext for the same plaintext can attack our scheme.
[3] Variants which do not require a blind signature scheme appear in Section 6.

We make two assumptions about this protocol: First, all messages in this protocol take place over an encrypted and authenticated channel. Second, the user somehow demonstrates that he is entitled to be given tickets by the server; we assume the user has already done this before the REQUEST protocol is run. There are many plausible ways this might be done, such as:

(1) The user may pay per ticket.
(2) The user may demonstrate his membership in some group to whom the server provides this service.
(3) The server may simply provide this service for all comers.

The specific method used is outside our scope. However the user demonstrates her authorization to receive tickets, it is very likely to involve revealing her identity. In order to protect the user's privacy from the server, the REQUEST protocol must thus prevent the server linking tickets with this identifying information, or linking tickets issued together.

4.1 Server Setup

The steps given below are done once by the server[4].

- The server establishes an encryption keypair PK_S, SK_S for some algorithm that supports CCA2 security. Server distributes its public key to all users.
- The server establishes a signature keypair PK'_S, SK'_S for some algorithm that supports blind signatures.
- The server establishes a list to store previously-seen tickets.

4.2 REQUEST: Protocol for Requesting Tickets

The basic ticket requesting protocol is illustrated below. The user starts out with a password P and a payload key K_P, and generates t tickets with the assistance of the server.

In order to create a ticket without revealing any identifying information to the server, the user will do the following steps:

(1) Randomly generate an n-bit salt S and an n-bit secret value B.
(2) Encrypt B using the public encryption key PK_S of the server, producing E.
(3) Run a protocol to get a blind signature on E from the server–this is F.
(4) Derive a one-time key from the password and the secret B:
$$D \longleftarrow \mathsf{HMAC}(B, \mathsf{PH}(S, P))$$
(5) Encrypt the payload key under the one-time key:
$$Z \longleftarrow \mathsf{VE}(D, K_P)$$

The ticket will consist of (S, E, F, Z); the user must irretrievably delete all the intermediate values above. The user repeats the steps t times to get t tickets. Below, we show the REQUEST protocol:

[4] Rolling over to new keys periodically can be done, but requires some extra administrative steps to ensure that old tickets continue to be usable.

Protocol: REQUEST(P, K_P, t):

User **Server**

for $i = 1 \dots t$
$S \leftarrow_\$ \{0,1\}^n$
$B \leftarrow_\$ \{0,1\}^n$
$E \leftarrow \mathtt{ENC}(PK_S, B)$
$E^* \leftarrow \mathtt{BLIND}(E)$

$\xrightarrow{E^*}$

$\sigma^* \leftarrow \mathtt{SIGN}_{SK'_S}(E^*)$

$\xleftarrow{\sigma^*}$

$F \leftarrow \mathtt{UBLIND}(\sigma^*)$
$C \leftarrow \mathtt{PH}(S, P)$
$D \leftarrow \mathtt{HMAC}(B, C)$
$Z \leftarrow \mathtt{VE}(D, K_P)$
Forget B, C, D, E^*, σ^*
$T_i \leftarrow (S, E, F, Z)$
endfor
return$(T_{1,2,\dots,t})$

At the end of this protocol, the user constructs t tickets she can use to run the UNLOCK protocol. The server, on the other hand, knows only that it has issued t tickets–it knows nothing else about them!

4.3 UNLOCK: Protocol for Unlocking a Ticket

In order to use a ticket along with a password $\hat{P}$ to unlock K_P, the user does the following steps:

(1) Hash the password: $\hat{C} \leftarrow \mathtt{PH}(S, \hat{P})$.
(2) Send $(E, F, \hat{C})$ to the server.
(3) If the signature is invalid or E is being reused, then the server returns $\perp$.
(4) Otherwise:
 (a) The server stores E, F as a used ticket.
 (b) $B \leftarrow \mathtt{DEC}(SK_S, E)$
 (c) $D \leftarrow \mathtt{HMAC}(B, \hat{C})$
 (d) The server sends back D.
(5) The user tries to decrypt Z with D. If this succeeds, she learns K_P. Otherwise, she learns that $\hat{P}$ was not the right password.

Note that in these two protocols, the server never learns anything about K_P, P, or $\hat{P}$, and has no way of linking a ticket between REQUEST and UNLOCK calls.

We also note that the UNLOCK protocol could be easily modified to enable creation of new tickets when the submitted password to UNLOCK is correct. This would ensure that a user who knows the password always has at least one valid ticket, which would improve usability of our scheme in real-world applications.

Protocol: UNLOCK($S, E, F, Z, \hat{P}$):

User **Server**

$\hat{C} \leftarrow \mathtt{PH}(S, \hat{P})$

$\xrightarrow{E, F, \hat{C}}$

IF
E fresh AND
$\mathtt{VERIFY}_{SK'_S}(E, F)$
THEN
$B \leftarrow \mathtt{DEC}(SK_S, E)$
$D \leftarrow \mathtt{HMAC}(B, \hat{C})$
ELSE
$D \leftarrow \perp$

$\xleftarrow{D}$

$K_P \leftarrow \mathtt{DV}(D, Z)$
return(K_P)

5 SECURITY ANALYSIS

In this section, we provide a security analysis and some security proofs for our basic protocol. Our approach comes in three separate parts: First, we define an *ideal functionality* for the system. Second, we prove that our basic protocol is indistinguishable from the ideal functionality in the UC framework. Finally, we provide several game-based security definitions, and prove bounds on an attacker's probability of winning the games when they are interacting with the ideal functionality. These game-based definitions show that the ideal functionality we've defined actually provides the practical security we need from this scheme.

The ideal functionality makes use of a table τ–a key-value database indexed by a ticket T. T can be any n-bit string, or the special values $\perp$ and *.

A user calls REQUEST to get a new ticket[5]. We assume a two-sided authenticated and secure channel for REQUEST–the ideal functionality knows the user's identity, and the user knows she is talking with the ideal functionality. Also, REQUEST requires an interaction with the server, in which the server also learns the user's identity. At the end of the REQUEST call, the user either has a valid ticket, or knows she did not get a valid ticket. Note that in the case of a corrupted server, we allow the server to "override" the honest behavior of the ideal functionality by outputting a value R. If $R = 1$, the ideal functionality proceeds as normal. If $R = 0$, it indicates that the server does not wish to cooperate. In this case, the output to the user is $\perp$. Note that in the real world, we cannot prevent the corrupt server from issuing an invalid ticket. However, in this case, we require that the user can detect that the ticket is invalid. The strongest guarantees we can hope for in the real world are therefore captured by our ideal functionality.

[5]The ideal functionality is defined for one ticket, but in our protocol, we define REQUEST to return t tickets at a time. This is equivalent to just rerunning the REQUEST ideal functionality t times.

Algorithm 3 Ideal Functionality: Initialize and Request

Initialize the table that will store passwords,
payload keys and aliases.
function Initialize(sid)
  sid.$\tau \leftarrow \{\}$
function Request(U, sid, P, K_P)
  # *T corresponds to the "ideal" ticket.*
  $T \leftarrow_{\$} \{0,1\}^n$
  # *Insert $(P, K_P, \perp)$ into table τ with key T. The $\perp$*
  # *value indicates that the ticket T is fresh.*
  sid.$\tau[T] \leftarrow (P, K_P, \perp)$
  Send to server sid: (sid, Request, U)
  if server sid compromised **then**
    Wait for response (sid, Request, U, R).
  else
    $R \leftarrow 1$
  if $R = 1$ **then**
    Send to source U: (sid, Request, T)
  else
    Send to source U: (sid, Request, $\perp$)

The user makes use of a ticket and a password to recover her payload key with an Unlock call. We assume the Unlock call is made over a secure channel which is authenticated on one side–the user knows she is talking with the ideal functionality, but the ideal functionality doesn't know who is talking to it. Unlock also requires an interaction with the server, in which the server is not told the identity of the user. At the end of the Unlock call, the user either learns the payload key associated with the ticket she has used, or receives an error message ($\perp$) and knows the Unlock call has failed. Note that in the case of a corrupted server, we allow the server to "override" the honest behavior of the ideal functionality by outputting a value R. If $R = 1$, the ideal functionality responds with the payload key, in the case that the password is correct, *even if the ticket is not fresh.* If $R = 0$, it indicates that the server does not wish to cooperate. In this case, the output to the user is $\perp$. Note that in the real world, we cannot prevent the corrupt server from responding to unlock requests with tickets that are not fresh (this corresponds to the corrupt ideal server flipping R from 0 to 1). Moreover, in the real world, we cannot prevent a corrupt server from deviating from the protocol and computing the wrong payload key (this corresponds to the corrupt ideal server flipping R from 1 to 0). However, in this case, we require that the user can detect that the returned payload key is invalid. The strongest guarantees we can hope for in the real world are therefore captured by our ideal functionality.

Before stating our theorem, we note that we assume that the protocols for Request and Unlock given in Sections 4.2 and 4.3 are executed in a hybrid model, where an ideal functionality for secure, two (resp. one)-sided authenticated channels, $\mathcal{F}_{\text{ac}}$ (resp. $\mathcal{F}_{\text{osac}}$), (see e.g. [8]) is invoked each time a message is sent. We require that the VE scheme used is the one given in Algorithms 1 and 2. We assume three independent random oracles: $H_{\text{pw}}, H_{\text{KD}}, H_{\text{VE}}$. H_{pw} is the password hash. H_{KD} is used to model the HMAC key derivation as a random oracle[6] and H_{VE} is the random oracle for the verifiable encryption scheme given in Algorithms 1, 2.

Algorithm 4 Ideal Functionality: Unlock

If ticket and password good, return K_P.
Otherwise, return $\perp$.
function Unlock(sid, T, $\hat{P}$)
  if $T \in$ sid.τ **then**
    $(P, K_P, \alpha) \leftarrow$ sid.$\tau[T]$
  else$\alpha = *$ signals invalid ticket.
    $(P, K_P, \alpha) \leftarrow (\perp, \perp, *)$
    $R \leftarrow 0$
  # *α corresponds to the alias for ticket T.*
  # *$\alpha = \perp$ indicates the ticket is fresh. $\alpha \neq \perp$ indicates*
  # *ticket T was previously assigned an alias so not fresh.*
  if $\alpha = \perp$ **then**
  # *Fresh ticket*
    $\alpha \leftarrow_{\$} \{0,1\}^n$
    $R \leftarrow 1$
  else
  # *Reused or invalid ticket*
    $R \leftarrow 0$
  # *Server can see whether it's getting invalid,*
  # *repeated, or fresh ticket.*
  Send to server sid: (sid, Unlock, α)
  # *If server is NOT compromised, we know R.*
  # *If server IS compromised, we must ask it*
  # *how to respond.*
  if Server sid compromised **then**
    Wait for (sid, Unlock, R) # *$R \in \{0,1\}$*
  # *Send back the right response to the user.*
  if $R = 0$ **then**
  # *Server returns $\perp$, no decryption possible.*
    Respond to caller: (sid, Unlock, $\perp$)
  else if $R = 1$ **then**
  # *Server plays straight.*
    if $\hat{P} = P$ **then**
      Respond to caller: (sid, Unlock, K_P)
    else
  # *Server returns value, decryption fails.*
      Respond to caller: (sid, Unlock, $\perp$)

Theorem 5.1. Under the assumption that Π_{ENC} is a CCA2-secure encryption scheme (see Definition A.5), Π_{BSIG} is a 2-move blind signature scheme (see Definition A.7) and the assumptions listed

[6] We remark that Dodis et el. [12] showed that HMAC is not indifferentiable from a random oracle. However, their attack only applies when one allows different sizes for the HMAC key. Since we require B to always be a fixed length, this attack does not apply to our setting–finding two values of B that give identical results from HMAC, implies finding collisions for the underlying hash function.

above, the protocols for Setup, Request and Unlock given in Sections 4.1, 4.2 and 4.3, UC-realize the ideal functionality provided in Algorithms 3 and 4 under adaptive corruptions, with erasures.

We note that our protocols can be generalized to work with multi-round blind signature schemes, and the same security proof goes through.

The proof of this theorem appears in Appendix C.

6 VARIANTS OF THE BASIC PROTOCOL

In this section we discuss some variants and modifications of the basic protocol which may be useful in specific situations.

6.1 Limiting Password Attempts

Ticket-mediated password strengthening permits a user to request a large number of tickets at once, and this may make sense for reasons of efficiency or convenience. However, if the user has chosen a very weak password, it would be helpful to limit any attacker who compromises the user's machine to a very small number of password guesses. For example, many systems have a limit of ten password attempts before locking an account. There is a straightforward way to get this same limit with ticket-mediated password strengthening, even when requesting hundreds or even thousands of tickets at a time.

Suppose the user has successfully created 1000 password-hashing tickets, $T_{0,1,2,\ldots,999}$. Each successful use of a password ticket derives the payload key, K_P. In order to implement a limit of at most ten password guesses, we do the following steps:

(1) Setup:
  (a) $K_T \leftarrow \mathsf{KDF}(K_P, \text{“ticket encryption”}, n)$
  (b) Using any authenticated encryption with associated data (AEAD) scheme, individually encrypt all but ten tickets under the key K_T
(2) Each time a ticket is successfully used to unlock K_P
  (a) $K_T \leftarrow \mathsf{KDF}(K_P, \text{“ticket encryption”}, n)$
  (b) Decrypt the next few encrypted tickets with key K_T, until we have ten tickets left unencrypted.

Consider an attacker who gets access to the stored data at some point. He has only ten tickets available. Assuming the KDF is secure, he cannot decrypt any other tickets without access to K_P, which he can get only by successfully using a ticket.

This technique can be used with our basic protocol or with any of our variants, described below.

6.2 Adding Offline Access

It is possible to add a second offline mode of access to the payload key. This may be a practical requirement in many cases, where a user needs to have access even when internet access is not available. However, this represents an explicit tradeoff between security and usability–the number of tickets no longer provides a limit on how many passwords may be guessed by an attacker who compromises the user's device.

If offline access is added, the first question is: how much computation should be required to unlock the payload key offline? We propose the following steps for adding offline access, if this is necessary, making use of the “pepper” idea of Abadi et al. [26]:

(1) Determine the largest acceptable amount of computing time on the device that would be acceptable to get offline access. Let this parameter be W. For example, we might require ten minutes' continuous computing on the user's device in order to unlock the offline access. (Note that in many cases, the constraint may be battery life rather than time.)
(2) Determine an acceptable time for the initial generation of the offline access information, I, such that $I \times 2^q = W$ for some integer q. For example, I might be 15 seconds on the user's device.
(3) Calculate $q \leftarrow \lg(W/I)$.
(4) Generate a random salt $S^* \leftarrow_{\$} \{0,1\}^n$ for offline access.
(5) Compute $D \leftarrow \mathsf{PH}(S^*, P)$, using parameters for the password hash that require I seconds to compute.
(6) Store $Z^* \leftarrow \mathsf{VE}(D, K_P)$
(7) Set the low q bits of S^* to zeros.
(8) Forget D.

If this is done, we strongly suggest using a memory-hard function for PH, with parameters set to the largest memory requirements that can be reasonably accommodated on the user device–this will make the offline attack more expensive and difficult, and may prevent the attacker using commodity graphics cards to parallelize the attack. Throwing away a few bits of the salt (following Abadi et al.) allows us to only do the work necessary to compute *one* instance of PH during setup, while still requiring an offline user (or attacker) to compute 2^q instances of PH.

6.2.1 Analysis. Consider a user who provides offline access requiring W work, alongside a TPMS scheme for online access. If the user's device is compromised, the attacker is no longer limited to t password guesses–instead, he first makes t password guesses “for free,” and then does W work per additional password guess.

Providing offline access throws away one of the major usability advantages available with ticket-mediated password strengthening: the ability of a user to choose a relatively low-entropy password safely. A random dictionary word or six-digit number provides substantial security against an attacker who has only ten guesses. Further, most users can probably understand what kind of passwords are necessary to withstand an attacker who is limited to ten guesses; few can properly estimate whether their password will survive an offline attack given the attacker's budget and the value of W.

The advantage of using a TMPS scheme in this situation is that it allows the work per offline guess to be set to some extremely high value, hopefully making the offline guessing attack too expensive for an attacker in practice, while the user can still get access her data with very little delay as long as she has internet access.

This technique can be used with our basic protocol, or with any of the variants described below. (Though it would not make much sense for the Offline Variant with HSM.)

6.3 An Offline Variant with HSM

Consider the situation where a user has a trusted computing environment or trusted hardware security module (HSM). We define an offline variant of ticket granting and unlocking protocol which uses an HSM and does not need any external server. However, we emphasize that this variant is secure only if the HSM is secure–an attacker who can extract the secret from or reload past states into the HSM can recover the K_P with a simple password search.

6.3.1 Starting Assumptions. We assume that the HSM can be loaded up with a secret value, B, which can not be released from the HSM afterward. We further assume that the HSM supports one-time use of the value B which is updated at each interface as described in Algorithm 5. Note that HSM_Step must be an atomic operation–if any value of D is returned, then B *must* be updated.

Algorithm 5 Access Secret and Update HSM Internal State

function HSM_STEP(C)
  $D \leftarrow \mathsf{HMAC}(B, C)$
  $B \leftarrow \mathsf{HASH}(B)$
  return(D)

We also assume that the user can load a new value of B into the HSM at any time which overwrites the previous existing value.

6.3.2 HSM REQUEST *Protocol.* In order to generate t tickets, the user first chooses a password, P and generates a random payload key K_P, and then follows the steps listed in Algorithm 6.

Algorithm 6 Create Tickets for the HSM Protocol

function HSM_REQUEST(P, K_P, t)
  $S \leftarrow \{0,1\}^n$
  $B \leftarrow \{0,1\}^n$
  $C \leftarrow \mathsf{PH}(S, P)$
  Load B into the HSM as the new secret value.
  for $i \leftarrow 1 \dots t$ **do**
    $D_i \leftarrow \mathsf{HMAC}(B, C)$
    $Z_i \leftarrow \mathsf{VE}(D_i, K_P)$
    $B \leftarrow \mathsf{HASH}(B)$
  Forget $D_{1,2,\dots,t}$, C, B
  return($S, Z_{1,2,\dots,t}$)

The protocol uses a fixed random salt S for generating all t tickets. As we do not need privacy from the HSM, reusing the salt and getting same C is acceptable. Similarly, there is no need for public key encryptions or signatures. Guessing the password is equally difficult as the password-derived value C is never stored. The value B is updated after each computation of Z_i, resulting t-different D_i's. The t-tickets $\{Z_1, Z_2, \dots, Z_t\}$ consist *only* of the encryptions of K_P under different keys D_i. As a result, the user storage as single S and $Z_{1,2,\dots,t}$.

Note that the same protocol can be used to generate new tickets when the old ones are running out. In that case, the user simply runs the HSM_UNLOCK algorithm (Algorithm 7) to recover K_P, and then runs the HSM_REQUEST algorithm (Algorithm 6) with P and K_P to get more tickets for the same password and payload key.

6.3.3 HSM UNLOCK *Protocol.* The process of unlocking the tickets is straightforward; however, it needs sequential runs of the protocol starting from ticket number 1 to t and hence, requires a strong synchronization between the HSM and the user. Specifically, the synchronization ensures the computation of the correct value of B and finally the K_P when correct password is provided. The protocol as shown in Algorithm 7 starts with accepting a password $\hat{P}$ from the user, which is used to derive a challenge value $\hat{C}$. The HMAC of this challenge value along with the current value of B inside the HSM is computed by the HSM and returned to the user as $\hat{D}$, and then B is again updated by the HSM. These computations inside the HSM follows the steps of Algorithm 5. Finally, the correctness of $\hat{D}$ is verified by analyzing the output K obtained at Step 4 of the Algorithm 7. The correct value of $\hat{D}$ implies K is the desired K_P.

Algorithm 7 Use an HSM Ticket to Unlock K_P

1: **function** HSM_UNLOCK($\hat{P}, S, Z$)
2:   $\hat{C} \leftarrow \mathsf{PH}(S, \hat{P})$
3:   $\hat{D} \leftarrow$ HSM_Step($\hat{C}$)
4:   $K \leftarrow \mathsf{DV}(\hat{D}, Z_i)$
5:   **if** $K = \perp$ **then**
6:     **return**($\perp$)
7:   **else**
8:     **return**(K)

The user must delete all old values of Z, in order to ensure that he can always determine which ticket is to be used next.

It is possible that some software error will lead to the HSM and user software getting out-of-synch. The best strategy for handling this is to attempt to unlock the payload key using the password and the final ticket, and to keep trying until the payload key is unlocked.

6.3.4 Analysis. Note that this scheme is not covered by our security proofs. Here, we provide some arguments for the security of the scheme.

Consider the situation where the user has produced t tickets, and then her laptop was stolen by an attacker who cannot violate the security of the HSM. Informally, what can we say about the attacker's chances of learning K_P?

The attacker needs to guess the correct value of $C = \mathsf{PH}(S, P)$. Each guess of the password leads to a guess of C.

The user has tickets corresponding to the next t values of B that will be used by the HSM. For $i = 1, 2, \dots, t$, a ticket Z_i is used as follows:

$$D_i = \mathsf{HMAC}(B_i, C)$$
$$Z_i = \mathsf{VE}(D_i, K_P)$$
$$B_{i+1} = \mathsf{HASH}(B_i)$$

The HSM will only carry out one computation with each value of B_i–this can be used to derive the decryption key D_i, but only if the attacker guesses the right value of C. Each value of B_i inside the HSM allows a new guess of C, and each value of Z_i in the attacker's hands allows the guess to be checked.

After t guesses, the HSM has a value of B for which the attacker has no corresponding values of Z. At this point, the attacker can learn nothing about K_P from interacting with the HSM. Since he also cannot break the encryption, this imposes a hard limit–the attacker gets only t guesses of the password.

In this setting, we have no privacy concerns with respect to the HSM. However, it's worth noting that the HSM never sees the password or any value it could use to check a password guess. (Though if the HSM was also used to generate the salt S, this would no longer be true.).

We assume that the HSM is able to securely keep B secret. Along with whatever tamper-resistance features are incorporated into the HSM, since B is updated each time it is used, side-channel attacks are very unlikely to succeed.

6.4 Different Ways to Authorize Tickets

In the basic protocol, we assume that the server issues blind signatures to allow the server to limit how much assistance it is required to provide. (That is, the server's owner presumably wants it to only provide TMPS services to users who have paid for them, or to users who are somehow affiliated with the entity running the server.). A blind signature works well to protect the user's privacy, but makes strong demands on the signature scheme used. In particular, most proposed post-quantum signature schemes have no known blind signature defined. Below, we discuss alternative ways for the server to limit access to its services *without* the need for a blind signature.

Our possible approaches fall into two broad categories:

(1) Offline–the Request operation is done without any interaction with the server or any other party.
(2) Online–the Request protocol is almost unchanged, but some other operation is substituted for the blind signature protocol.

Note that the security proof on our basic protocol doesn't cover these variants, though we believe it could be modified to cover them without too much difficulty. For each variant, we provide a short sketch of why we believe the variant is secure. Also note that we still assume that the public key encryption used below is CCA2–specifically, it must not be possible to modify a ciphertext without changing the plaintext.

6.4.1 *Third Party Signer (Online).*

A very lightweight (but imperfect) technique for ensuring user privacy from the server is simply to separate the authorization of getting a ticket from the unlocking of tickets. Suppose we have two trusted parties: the Bank and the Server. The Bank authorizes tickets, and can recognize a ticket, but is never shown tickets by the Server; the Server unlocks tickets but can't recognize them. In this protocol, we need an ordinary signature, nothing more.

The Request protocol works as follows. Note that this is almost the same protocol as with the blind signatures, except it is done with the Banker instead of the Server.

Protocol: ThirdParty_Request(P, K_P, t):

User | **Banker**

for $i = 1 \ldots t$
  $B \leftarrow_{\$} \{0,1\}^n$
  $E \leftarrow \mathsf{ENC}(PK_S, B)$
$\xrightarrow{E}$
      $F \leftarrow \text{sign}(SK_B, E)$
$\xleftarrow{F}$
  $S \leftarrow_{\$} \{0,1\}^n$
  $C \leftarrow \mathsf{PH}(S, P)$
  $D \leftarrow \mathsf{HMAC}(B, C)$
  $Z \leftarrow \mathsf{VE}(D, K_P)$
  Forget B, C, D, E^*, σ^*
  $T_i \leftarrow (S, E, F, Z)$
endfor
return($T_{1,2,\ldots,t}$)

The unlocking protocol is exactly the same except for the public key used to verify the signature.

Security. This scheme is almost identical to the basic protocol–the only difference is that the Request protocol is run with a different server, and no blind signature is used. The user's privacy from the server is ensured by separation of information–the banker knows the signatures it issued to the user, but doesn't share that information with the server. An attacker who compromises the user's device has exactly the same probability of success in this scheme as in the main protocol.

6.4.2 *Group Signatures (Offline).*

If we want to use TMPS with a signature scheme which doesn't allow blind signatures, but allows group or ring signatures, then a small variation of the protocol can be done. We assume here that the user has a group public key PK and a personal private key SK_U for the group signature scheme. We also assume that the server knows PK.

Algorithm 8 Use Group Signature to Create Tickets

```
function Group_Request(P, K_P, t)
    for i ← 1 . . . t do
        B ←$ {0, 1}^n
        E ← ENC(PK_S, B)
        F ← GroupSign(SK_U, E)
        S ←$ {0, 1}^n
        C ← PH(S, P)
        D ← HMAC(B, C)
        Z ← VE(D_i, K_P)
        Forget D_{1,2,...,N}, C, B
        T_i ← (S, E, F, Z)
    return(T_{1,2,...,t})
```

The corresponding Unlock protocol is almost unchanged–the server simply verifies that F signs E using the group public key PK rather than its own signature public key PK'_S.

Security. Consider an attacker who compromises the user's device, and thus learns his tickets and his signing key. The attacks possible to him are the same as in the basic protocol–he can request new tickets, but without knowing K_P or P, these will give him no help in learning the correct value of P or K_P. Despite knowing the signing key, he cannot alter tickets to give himself more guesses, because the public key encryption used is CCA2.

The server can't learn which user is Unlocking her key at any given time, because the group signature tells the server only that she is a member of the group.

6.4.3 *Proof of Work (Offline).*

If the server's main concern is having its resources wasted rather than being paid for its services, a simple alternative is to require a proof of work for each new ticket. Let's add two new functions:

$y \leftarrow \texttt{MakePOW}(x, W)$ does approximately W work to create a proof of work, y, associated with input value x.

$\texttt{CheckPOW}(x, y, W)$ returns 1 if the proof of work is valid, and 0 otherwise.

With these two, we can define an entirely offline proof-of-work version of our protocol, where W is assumed to be a known systemwide parameter.

Algorithm 9 Use Proof of Work to Create Tickets

```
function POW_Request(P, K_P, t)
    for i ← 1 ... t do
        B ←$ {0,1}^n
        E ← ENC(PK_S, B)
        F ← MakePOW(E, W)
        S ←$ {0,1}^n
        C ← PH(S, P)
        D ← HMAC(B, C)
        Z ← VE(D_i, K_P)
        Forget D_{1,2,...,N}, C, B
        T_i ← (S, E, F, Z)
    return(T_{1,2,...,t})
```

Protocol: POW_Unlock$(S, E, F, Z, \hat{P})$:

User		**Server**
$\hat{C} \leftarrow \texttt{PH}(S, \hat{P})$		
	$\xrightarrow{E, F, \hat{C}}$	
		IF
		E fresh AND
		$\texttt{CheckPOW}(F, E, W)$
		THEN
		$B \leftarrow \texttt{DEC}(SK_S, E)$
		$D \leftarrow \texttt{HMAC}(B, \hat{C})$
		ELSE
		$D \leftarrow \perp$
	$\xleftarrow{D}$	
$K_P \leftarrow \texttt{DV}(D, Z)$		
return(K_P)		

Security. The use of the proof of work eliminates any information about the user in the ticket, and in fact, makes the Request protocol non-interactive.

An attacker who compromises the user's device can make up additional tickets, but without knowing K_P or P, these do not help him learn the correct values of P or K_P. Again, the attacker cannot alter the value of E, because the public key encryption is CCA2 secure.

7 PERFORMANCE AND IMPLEMENTATION

The TMPS protocol requires several primitives:

(1) Password hashing (e.g., PBKDF#2 or Argon2)
(2) Public key encryption (e.g. RSA or El Gamal)
(3) Blind signatures (e.g. RSA or ElGamal)
(4) Hash functions and HMAC (e.g., using SHA256 or Blake2)

The protocol permits a great deal of flexibility in choice of underlying cryptographic primitives. Notably, there are proposed post-quantum algorithms that meet these requirements.

7.1 Prototype Implementation

We implemented our protocol in Python[7], using the Cryptography module, which provides a Python frontend for OpenSSL calls. The protocol allows a choice of underlying primitives; we used RSA with 3072-bit moduli for (blind) signatures and public key encryption, along with SHA256 for hashing, and PBKDF2_HMAC_SHA2 for password-hashing.

All measurements were performed on a Macbook Pro (3.5 GHz Intel Core i7). While this is not an optimized implementation, it allows us to obtain concrete performance numbers, and it demonstrates the practicality of the scheme.

[7] We will make source code available on a public-facing git repository

7.2 Requesting a Ticket

On the user device, each ticket REQUEST requires the following operations:

- One password hash computation.
- Generating $2n$ random bits.
- One public key encryption.
- Blinding and unblinding one signature request.
- One HMAC computation.
- Two hash operations.

With RSA, this is comparable to the work needed to set up a TLS connection. Thus, devices that can set up a TLS connection can REQUEST tickets. The slowest part of this process on the user device is likely to be the password hash computation, which can be controlled by choosing its hardness parameters.

In our implementation, each REQUEST required about 0.008 seconds on the user side.

On the server, each REQUEST requires only a blind signature. With RSA, this is approximately the same cost as a normal RSA signature[8].

In our implementation, each REQUEST required about 0.076 seconds on the server side.

7.3 Unlocking a Ticket

On the user device, each UNLOCK requires the following operations:

- One password hash computation.
- One HMAC computation.
- Two hash operations.

Again, the password hash is almost certain to be the slowest part of this process.

In our implementation, each UNLOCK required about 0.0049 seconds on the user side.

On the server, an UNLOCK call requires a few operations:

- Looking up a value in a list of previously-used tickets.
- A signature verification
- A public key decryption
- An HMAC computation

The cryptography used here is comparable to setting up a TLS connection, and so should be no problem for any server.

In our implementation, each UNLOCK required about 0.002 seconds on the server side.

7.4 Storage

Keeping track of the previously-used tickets requires some storage, but not a huge amount. We can hash the value of E from the ticket (the public-key encrypted value) into 128-bits[9] (16 bytes), e.g., by truncating SHA256 outputs at 128 bits.

A user who makes ten UNLOCK calls per day will go through fewer than 4096 tickets in one year. The server needs 64 KiB to store one 16-bit hash for each of those tickets. If the server supports 1000 users, it will need about 64 MiB for a year's worth of tickets–a hash table with these values in it will fit into RAM.

Using 3072-bit RSA, each ticket requires less than 1 KiB on the user device. Thus, even low-end devices like tablets and smartphones can easily store a year's supply of tickets.

[8] The extra work for getting a blind RSA signature is done by the person requesting the blind signature–they must blind the signature request, and unblind the value they get back from the signer.

[9] We can use a relatively short hash because we don't care about collisions–an attacker who forces two tickets to collide simply deprives himself of the use of one of his tickets.

8 CONCLUSION AND OPEN QUESTIONS

In this paper, we have proposed a new mechanism for strengthening password-based key derivations, called TMPS (Ticket-Mediated Password Strengthening). We have also proposed a set of protocols that implements a TMPS scheme, proven its security in the UC model, and provided a number of variant schemes which allow for different implementation constraints and tradeoffs.

There are several questions left open by this research.

- *Is it possible to construct TMPS schemes which provide privacy for the user, allow the server to restrict access to only authorized users, and do not need blind or group signatures.* The variant protocols in Section 6 that avoided these primitives imposed other requirements–either a willingness to trust a third party with user privacy, or a willingness to provide the service to all comers. Are there better options?
- *Are there other settings where one can use tickets bound to a computation to obtain a novel functionality?* For example, could we use this kind of mechanism to limit accesses to a local encrypted database, or computations of a key derivation function.
- *Are there are more elaborate restrictions that can be imposed on these tickets, without losing the users' privacy?* For example, is it possible to rate-limit UNLOCK requests from a given user without revealing which user was using the scheme?
- Finally, a number of additional features would be useful in implementing this scheme on a large scale. For example, support for key rollover on the server would be useful, as would a specific solution for backed-up data (where most or all of the stored tickets in the backup may have already been used).

ACKNOWLEDGMENTS

The authors would like to thank Bart Preneel, Frank Piessens, Vincent Rijmen, Kristen Greene, Ray Perlner, Lily Chen, and the participants of the May 2019 DC Area Crypto Day workshop for many useful comments and suggestions.

REFERENCES

[1] Shashank Agrawal, Peihan Miao, Payman Mohassel, and Pratyay Mukherjee. 2018. PASTA: PASsword-based Threshold Authentication. In *ACM Conference on Computer and Communications Security*. ACM, 2042–2059.

[2] Devdatta Akhawe. 2016. How Dropbox securely stores your passwords. https://blogs.dropbox.com/tech/2016/09/how-dropbox-securely-stores-your-passwords/. Online; accessed 4 January 2019.

[3] Mihir Bellare, Sriram Keelveedhi, and Thomas Ristenpart. 2013. DupLESS: Server-Aided Encryption for Deduplicated Storage. *IACR Cryptology ePrint Archive* 2013 (2013), 429. http://eprint.iacr.org/2013/429

[4] Mihir Bellare, Daniele Micciancio, and Bogdan Warinschi. 2003. Foundations of Group Signatures: Formal Definitions, Simplified Requirements, and a Construction Based on General Assumptions. In *EUROCRYPT 2003 (LNCS)*, Eli Biham (Ed.), Vol. 2656. Springer, Heidelberg, 614–629. https://doi.org/10.1007/3-540-39200-9_38

[5] Mihir Bellare, David Pointcheval, and Phillip Rogaway. 2000. Authenticated Key Exchange Secure Against Dictionary Attacks. In *Proceedings of the 19th International Conference on Theory and Application of Cryptographic Techniques (EUROCRYPT'00)*. Springer-Verlag, Berlin, Heidelberg, 139–155. http://dl.acm.org/citation.cfm?id=1756169.1756185

[6] Steven M. Bellovin and Michael Merritt. 1992. Encrypted Key Exchange: Password-Based Protocols Secure Against Dictionary Attacks. In *IEEE SYMPOSIUM ON RESEARCH IN SECURITY AND PRIVACY*. 72–84.

[7] Jan Camenisch, Manu Drijvers, Tommaso Gagliardoni, Anja Lehmann, and Gregory Neven. 2018. The Wonderful World of Global Random Oracles. In *EUROCRYPT 2018, Part I (LNCS)*, Jesper Buus Nielsen and Vincent Rijmen (Eds.), Vol. 10820. Springer, Heidelberg, 280–312. https://doi.org/10.1007/978-3-319-78381-9_11

[8] Jan Camenisch, Robert R. Enderlein, and Gregory Neven. 2015. Two-Server Password-Authenticated Secret Sharing UC-Secure Against Transient Corruptions. Cryptology ePrint Archive, Report 2015/006. http://eprint.iacr.org/2015/006.

[9] Ran Canetti. 2000. Security and Composition of Multiparty Cryptographic Protocols. *Journal of Cryptology* 13, 1 (Jan. 2000), 143–202. https://doi.org/10.1007/s001459910006

[10] Ran Canetti, Ivan Damgård, Stefan Dziembowski, Yuval Ishai, and Tal Malkin. 2001. On Adaptive vs. Non-adaptive Security of Multiparty Protocols. In *EUROCRYPT 2001 (LNCS)*, Birgit Pfitzmann (Ed.), Vol. 2045. Springer, Heidelberg, 262–279. https://doi.org/10.1007/3-540-44987-6_17

[11] Ran Canetti, Uriel Feige, Oded Goldreich, and Moni Naor. 1996. Adaptively Secure Multi-Party Computation. In *28th ACM STOC*. ACM Press, 639–648. https://doi.org/10.1145/237814.238015

[12] Yevgeniy Dodis, Thomas Ristenpart, John P. Steinberger, and Stefano Tessaro. 2012. To Hash or Not to Hash Again? (In)Differentiability Results for H^2 and HMAC. In *CRYPTO 2012 (LNCS)*, Reihaneh Safavi-Naini and Ran Canetti (Eds.), Vol. 7417. Springer, Heidelberg, 348–366. https://doi.org/10.1007/978-3-642-32009-5_21

[13] Adam Everspaugh, Rahul Chaterjee, Samuel Scott, Ari Juels, and Thomas Ristenpart. 2015. The Pythia PRF Service. In *24th USENIX Security Symposium (USENIX Security 15)*. USENIX Association, Washington, D.C., 547–562. https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/everspaugh

[14] Warwick Ford and Burton S. Kaliski Jr. 2000. Server-Assisted Generation of a Strong Secret from a Password. In *9th IEEE International Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises (WETICE 2000), 4-16 June 2000, Gaithersburg, MD, USA*. IEEE Computer Society, 176–180. http://ieeexplore.ieee.org/xpl/mostRecentIssue.jsp?punumber=7094

[15] Susan Hohenberger, Allison Lewko, and Brent Waters. 2012. Detecting Dangerous Queries: A New Approach for Chosen Ciphertext Security. Cryptology ePrint Archive, Report 2012/006. http://eprint.iacr.org/2012/006.

[16] Stanislaw Jarecki, Aggelos Kiayias, Hugo Krawczyk, and Jiayu Xu. 2017. TOPPSS: Cost-Minimal Password-Protected Secret Sharing Based on Threshold OPRF. In *ACNS 17 (LNCS)*, Dieter Gollmann, Atsuko Miyaji, and Hiroaki Kikuchi (Eds.), Vol. 10355. Springer, Heidelberg, 39–58. https://doi.org/10.1007/978-3-319-61204-1_3

[17] Stanislaw Jarecki, Hugo Krawczyk, and Jiayu Xu. 2018. OPAQUE: An Asymmetric PAKE Protocol Secure Against Pre-computation Attacks. In *EUROCRYPT (3) (Lecture Notes in Computer Science)*, Vol. 10822. Springer, 456–486.

[18] Jonathan Katz and Yehuda Lindell. 2014. *Introduction to Modern Cryptography, Second Edition.* CRC Press.

[19] John Kelsey, Bruce Schneier, Chris Hall, and David Wagner. 1998. Secure Applications of Low-Entropy Keys. In *ISW'97 (LNCS)*, Eiji Okamoto, George I. Davida, and Masahiro Mambo (Eds.), Vol. 1396. Springer, Heidelberg, 121–134.

[20] Klaudia Krawiecka, Arseny Kurnikov, Andrew Paverd, Mohammad Mannan, and N. Asokan. 2018. SafeKeeper: Protecting Web Passwords using Trusted Execution Environments. In *Proceedings of the 2018 World Wide Web Conference on World Wide Web, WWW 2018, Lyon, France, April 23-27, 2018*, Pierre-Antoine Champin, Fabien L. Gandon, Mounia Lalmas, and Panagiotis G. Ipeirotis (Eds.). ACM, 349–358. https://doi.org/10.1145/3178876

[21] Klaudia Krawiecka, Andrew Paverd, and N. Asokan. 2016. Protecting Password Databases using Trusted Hardware. In *Proceedings of the 1st Workshop on System Software for Trusted Execution, SysTEX@Middleware 2016, Trento, Italy, December 12, 2016*. ACM, 9:1–9:6. https://doi.org/10.1145/3007788

[22] Russell W. F. Lai, Christoph Egger, Manuel Reinert, Sherman S. M. Chow, Matteo Maffei, and Dominique Schröder. 2018. Simple Password-Hardened Encryption Services. In *27th USENIX Security Symposium (USENIX Security 18)*. USENIX Association, Baltimore, MD, 1405–1421. https://www.usenix.org/conference/usenixsecurity18/presentation/lai

[23] Russell W. F. Lai, Christoph Egger, Dominique Schröder, and Sherman S. M. Chow. 2017. Phoenix: Rebirth of a Cryptographic Password-Hardening Service. In *26th USENIX Security Symposium (USENIX Security 17)*. USENIX Association, Vancouver, BC, 899–916. https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/lai

[24] Andrew Y. Lindell. 2009. Adaptively Secure Two-Party Computation with Erasures. In *CT-RSA 2009 (LNCS)*, Marc Fischlin (Ed.), Vol. 5473. Springer, Heidelberg, 117–132. https://doi.org/10.1007/978-3-642-00862-7_8

[25] Arvind Mani. 2015. Life of a Password. Real World Crypto 2015. https://rwc.iacr.org/2015/Slides/RWC-2015-Amani.pdf.

[26] T. Mark A. Lomas Martin Abadi and Roger Needham. 1997. Strengthening passwords. Technical report. https://users.soe.ucsc.edu/~abadi/Papers/pwd-revised/pwd-revised.html.

[27] Alec Muffett. 2014. Facebook: Password Hashing & Authentication. Presentation at Passwords 2014 Conference, NTNU. https://video.adm.ntnu.no/pres/54b660049af94.

[28] Alec Muffett. 2015. Life of a Password. Presentation at Real World Crypto 2015.

[29] Jonas Schneider, Nils Fleischhacker, Dominique Schröder, and Michael Backes. 2016. Efficient Cryptographic Password Hardening Services from Partially Oblivious Commitments. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24-28, 2016*, Edgar R. Weippl, Stefan Katzenbeisser, Christopher Kruegel, Andrew C. Myers, and Shai Halevi (Eds.). ACM, 1192–1203. http://dl.acm.org/citation.cfm?id=2976749

[30] Dominique Schröder and Dominique Unruh. 2012. Security of Blind Signatures Revisited. In *PKC 2012 (LNCS)*, Marc Fischlin, Johannes Buchmann, and Mark Manulis (Eds.), Vol. 7293. Springer, Heidelberg, 662–679. https://doi.org/10.1007/978-3-642-30057-8_39

[31] Meltem Sönmez Turan, Elaine B. Barker, William E. Burr, and Lidong Chen. 2010. *SP 800-132. Recommendation for Password-Based Key Derivation: Part 1: Storage Applications.* Technical Report. National Institute of Standards & Technology, Gaithersburg, MD, United States.

[32] Thomas Wu. 2000. The SRP Authentication and Key Exchange System. *RFC* 2945 (2000), 1–8. https://doi.org/10.17487/RFC2945

A DEFINITIONS

In this section, we mention the key definitions used in the security analysis of our protocol to facilitate better understanding. Our exposition closely follows [4, 15, 18, 30].

Definition A.1. [Encryption System] An encryption system can be defined as a tuple of probabilistic polynomial-time algorithms $\Pi_{\mathsf{ENC}}(\mathsf{GEN}, \mathsf{ENC}, \mathsf{DEC})$ such that:

(1) The key-generation algorithm GEN takes as input the security parameter 1^n and outputs a key K.
(2) The encryption algorithm ENC takes as input a key K and a plaintext message $M \in \{0,1\}^*$, and outputs a ciphertext C where $C \leftarrow \mathsf{ENC}_K(M)$.
(3) The decryption algorithm DEC takes as input a key and a ciphertext, and outputs a message. We assume without loss of generality that the decryption algorithm corresponding ENC_K is DEC_K such that $M = \mathsf{DEC}_K(C)$ and for every n, every key K output by $\mathsf{GEN}(1^n)$, and every $M \in \{0,1\}^*$, it holds that $\mathsf{DEC}_K(\mathsf{ENC}_K(M)) = M$.

The Chosen-Ciphertext Attack (CCA) security experiment $PrivK^{cca}_{\mathcal{A},\Pi_{\mathsf{ENC}}}(n)$**:** Consider the following experiment for an encryption system $\Pi_{\mathsf{ENC}} = (\mathsf{GEN}, \mathsf{ENC}, \mathsf{DEC})$, adversary $\mathcal{A}$, and value n for the security parameter.

(1) A random key K is generated by running $\mathsf{GEN}(1^n)$.
(2) The adversary $\mathcal{A}$ is given input 1^n and oracle access to $\mathsf{ENC}_K(\cdot)$ and $\mathsf{DEC}_K(\cdot)$. It outputs a pair of messages M_0, M_1 of the same length.
(3) A random bit $b \leftarrow \{0,1\}$ is chosen, and then a ciphertext $C \leftarrow \mathsf{ENC}_K(M_b)$ is computed and given to $\mathcal{A}$. We call C the challenge ciphertext.
(4) The adversary $\mathcal{A}$ continues to have oracle access to $\mathsf{ENC}_K(\cdot)$ and $\mathsf{DEC}_K(\cdot)$, but is not allowed to query the latter on the challenge ciphertext itself. Eventually, $\mathcal{A}$ outputs a bit b'

(5) The output of the experiment is defined to be 1 if $b' = b$, and 0 otherwise.

Definition A.2. [CCA Security] An encryption system Π_{ENC} has indistinguishable encryptions under a chosen-ciphertext attack (or is CCA-secure) if for all probabilistic polynomial-time adversaries $\mathcal{A}$ there exists a negligible function negl such that:

$$Pr[PrivK^{cca}_{\mathcal{A},\Pi_{\mathsf{ENC}}}(n) = 1] \leq \frac{1}{2} + negl(n),$$

where the probability is taken over all random coins used in the experiment.

Other variants of the CCA Security definition are defined below.

Definition A.3. [Chosen Plaintext Attack (CPA) Security] Similar to the security experiment of CCA except that the Adversary $\mathcal{A}$ is not given access to decryption oracle at step 2 and step 4.

Definition A.4. [Non-adaptive CCA or CCA1 Security] Similar to the security experiment of CCA except that the Adversary $\mathcal{A}$ is not given access to decryption oracle at step 4.

Definition A.5. [Adaptive CCA or CCA2 Security] Similar to the security experiment of CCA where the Adversary $\mathcal{A}$ is allowed to perform a polynomially bounded number of encryptions, decryptions or other calculations over inputs of its choice except on the challenge ciphertext.

Definition A.6. [Signature Scheme]: A signature scheme is a tuple of probabilistic polynomial-time algorithms $\Pi_{SIG}(\mathsf{GEN}, \mathsf{SIGN}, \mathsf{VERIFY})$ such that:

(1) The key-generation algorithm GEN takes as input a security parameter 1^n and outputs a pair of keys (PK, SK). These are called the public key and the private key, respectively.
(2) The signing algorithm SIGN takes as input a private key SK and a message M from some underlying message space. It outputs a signature F represented as $F \leftarrow \mathsf{SIGN}_{SK}(M)$.
(3) The deterministic verification algorithm VERIFY takes as input a public key PK, a message M, and a signature F. It outputs a bit b represented as $b = \mathsf{VERIFY}_{PK}(M, F)$ where $b = 1$ means valid and $b = 0$ means invalid.

We require that for every n, every (PK, SK) output by $\mathsf{GEN}(1^n)$, and every message M in the appropriate underlying plaintext space, it holds that

$$\mathsf{VERIFY}_{PK}(M, \mathsf{SIGN}_{SK}(M)) = 1.$$

We say F is a valid signature on a message M if $\mathsf{VERIFY}_{PK}(M, F) = 1$.

Definition A.7. [Blind Signature] A 2-move blind signature scheme is an interactive signature scheme with signer $\mathcal{S}$ and user $\mathcal{U}$ and can be defined as a tuple of probabilistic polynomial-time algorithms $\Pi_{BSIG} = (\mathsf{GEN}, \mathsf{BLIND}, \mathsf{UBLIND}, \mathsf{SIGN}, \mathsf{BVERIFY})$ such that:

(1) The key-generation algorithm Gen takes as input a security parameter 1^n and outputs a pair of keys (PK, SK). These are called the public key and the private key, respectively.
(2) Signature Issuing. The parties execute the following protocol, denoted $\langle \mathcal{U}(PK, M), \mathcal{S}(SK)\rangle$:
   (a) $M^* \leftarrow \mathsf{BLIND}(M)$: The user blinds the message M to obtain M^* and sends to the signer.
   (b) $F^* \leftarrow \mathsf{SIGN}_{SK}(M^*)$: The signer outputs a signature F^* on input of message M^* and private key SK and sends to the user.
   (c) $F \leftarrow \mathsf{UBLIND}(F^*)$: The user unblinds the signature F^* to obtain F. Note that the user inputs additional private state to the UBLIND algorithm, which we leave implicit.
(3) The deterministic verification algorithm BVERIFY takes as input a public key PK, a message M, and a signature F. It outputs a bit b where $b = 1$ means valid and $b = 0$ means invalid.

We require that for every n, every (PK, SK) output by $\mathsf{GEN}(1^n)$, and every message $M \in \{0, 1\}^n$ and any F output by $\mathcal{U}$ in the joint execution of $\langle \mathcal{U}(PK, M), \mathcal{S}(SK)\rangle$, it holds that

$$\mathsf{BVERIFY}_{PK}(M, F) = 1.$$

The security of blind signature schemes requires two properties, namely unforgeability and blindness.

Definition A.8. [Unforgeability] A 2-move blind signature scheme $\Pi_{BSIG} = (\mathsf{GEN}, \mathsf{BLIND}, \mathsf{UBLIND}, \mathsf{SIGN}, \mathsf{BVERIFY})$ is called unforgeable if for any efficient algorithm $\mathcal{A}$ the probability that experiment $\text{Unforge}^{\Pi_{BSIG}}_{\mathcal{A}}(n)$ evaluates to 1 is negligible (as a function of n) where

Experiment $\mathbf{Forge}^{\mathcal{A}}_{\Pi_{BSIG}}$

(1) $(SK, PK) \leftarrow \mathsf{GEN}(1^n)$
(2) $((M_1, F_1), \cdots, (M_{k+1}, F_{k+1})) \leftarrow \mathcal{A}^{\langle\cdot,\mathcal{S}(SK)\rangle^\infty}(PK)$ Return 1 iff
   (a) $M_i \neq M_j$ for $1 \leq i < j \leq k + 1$ and
   (b) $\mathsf{BVERIFY}_{PK}(M_i, F_i) = 1$ for all $i = 1, 2, \cdots, k + 1$, and
   (c) at most k interactions with $\langle\cdot, \mathcal{S}(SK)\rangle^\infty$ were completed.

Definition A.9. [Blindness] A 2-move blind signature scheme $\Pi_{BSIG} = (\mathsf{GEN}, \mathsf{BLIND}, \mathsf{UBLIND}, \mathsf{SIGN}, \mathsf{BVERIFY})$ is called blind if for any efficient algorithm $\mathcal{A}$ the probability that experiment $Blind^{\Pi_{BSIG}}_{\mathsf{BSIGN}^*}(n)$ evaluates to 1 is negligibly close to $\frac{1}{2}$ where

Experiment $\mathbf{Blind}^{\Pi_{BSIG}}_{\mathsf{BSIGN}^*}$

(1) $(PK, M_0, M_1, st_{find}) \leftarrow \mathcal{A}(find, 1^n)$
(2) $b \leftarrow \{0, 1\}$
(3) $st_{issue} \leftarrow \mathcal{A}^{\langle\mathcal{U}(PK,M_b),\cdot\rangle^1, \langle\mathcal{U}(PK,M_{1-b}),\cdot\rangle^1}(issue, st_{find})$ and let F_b, F_{1-b} denote the (possibly undefined) local outputs of $\mathcal{U}(PK, M_b)$ resp. $\mathcal{U}(PK, M_{1-b})$
(4) set $(F_0, F_1) = (\perp, \perp)$ if $F_0 = \perp$ or $F_1 = \perp$
(5) $b^* = \mathcal{A}(guess, F_0, F_1, st_{issue})$
(6) return 1 iff $b = b^*$.

Definition A.10. [Group Signature] A group signature scheme $\Pi_{GSIG} = (GK_g, \mathsf{GSIGN}, \mathsf{GVERIFY}, \mathsf{OPEN})$ consists of four polynomial-time algorithms:

(1) The randomized group key generation algorithm GK_g takes input a security parameter 1^n and 1^m where $m \in \mathbb{N}$ is the group size and outputs a tuple $(gPK, gmSK, gSK)$, where gPK is the group public key, $gmSK$ is the group manager's secret key, and gSK is an n-vector of keys with $gSK[i]$ being a secret signing key for player $i \in [m]$.

(2) The randomized group signing algorithm GSIGN takes as input a secret signing key $gSK[i]$ and a message M to return a signature of M under $gSK[i]$ $i \in [m]$.
(3) The deterministic group signature verification algorithm GVERIFY takes as input the group public key gPK, a message M, and a candidate signature F for M to return either 1 or 0.
(4) The deterministic opening algorithm OPEN takes as input the group manager secret key $gmSK$, a message M, and a signature F of M to return an identity i or the symbol $\perp$ to indicate failure.

Correctness: The scheme must satisfy the following correctness requirement. For all $n, m \in \mathbb{N}$, all $(gPK, gmSK, gSK) \in [GK_g(1^n, 1^m)]$, all $i \in [n]$ and all $M \in \{0, 1\}^*$

$$\mathtt{GVERIFY}(gPK, M, \mathtt{GSIGN}(gSK[i], M)) = 1 \text{ and}$$

$$\mathtt{OPEN}(gmSK, M, \mathtt{GSIGN}(gSK[i], M)) = i$$

Definitions of security in the Universal Composability (UC) framework. We refer to previous work [9, 10, 24] for definitions of UC secure computation in the adaptive-corruption setting.

B GAME-BASED SECURITY DEFINITIONS AND PROOFS

Theorem 5.1 ensures that the basic protocol behaves like the ideal functionality, but does not tell us exactly what security properties the ideal functionality provides. In this section, we consider some game-based security definitions, and show that the ideal functionality makes it easy to prove a bound on an attacker's probability of winning these games. Combined with Theorem 5.1, we thus prove that no attacker interacting with our basic protocol can win these games with probability more than negligibly higher than these bounds.

B.1 User Compromise: Stealing Tickets

The most practically important attack to consider involves the compromise of a user's data. For example, Bob steals Alice's laptop with an encrypted hard drive; he knows all her tickets, and can impersonate her to the server, but he doesn't know Alice's password. The security goal of our scheme in this case is straightforward: after Bob steals Alice's t tickets, he gets only t guesses for her password. We can define this in terms of the following game:

Security Game: User Compromise

(1) The game is parametrized by security parameter n, dictionary size N and number of tickets t, where $t < N$.
(2) The challenger generates a list of N distinct passwords, $P_{1,\ldots,N}$.
(3) The challenger randomly generates a payload key, $K_P \leftarrow \{0, 1\}^n$.
(4) The challenger chooses a random $\ell \in \{1, \ldots, N\}$.
(5) The challenger honestly runs `ServerSetup` and `RequestTickets` the to generate t tickets $T_{1,\ldots,t}$ using password P_ℓ and payload key K_P.
(6) The challenger provides the attacker with the list of N passwords and t tickets.
(7) The attacker may send any messages he likes to the server, and may do any computation he likes, up to some very generous limits.
(8) The attacker may request new tickets by running `RequestTickets` with the challenger.
(9) The attacker must return a guess of K_P. If his guess is correct, he wins; otherwise he loses.

In this game, we explicitly assume that the server being used is *not* compromised.

Consider an attacker allowed to make at most 2^k queries to the server, and at most 2^k trial decryption attempts to DV. The protocol meets its security target against this attacker if he wins the game with probability at most $\frac{t}{N} + \mathsf{negl}(n-k)$.

B.1.1 Practical Relevance. This game directly relates to the attack we are most concerned with in this system–the one where Bob learns all of Alice's stored information, and tries to guess her password so he can decrypt all her files. If Alice keeps only t tickets on her computer, this must translate to Bob getting no more than t guesses at her password, in order for our system to be secure.

B.1.2 Proof. We now will prove that an adversary interacting with the ideal functionality and limited to at most 2^k trial decryptions with DV has at most a $\frac{t}{N} + \mathsf{negl}(n)\, n-k$ probability of winning this game.

Definition B.1. *Winning Transcript*
An attacker has a *winning transcript* (WT) when the transcript of his interactions with the ideal functionality includes at least one response to an Unlock request which contains the correct value of K_P.

A *losing transcript* (LT) is any transcript which is not a winning transcript.

Fact. Given a winning transcript, the attacker has a probability of one of winning the game.

We write $Pr[WT]$ to denote the probability of getting a winning transcript, and $Pr[WT|A]$ to denote the probability of adversary A getting a winning transcript.

Fact. An Unlock with an incorrect password reveals no information about the correct password other than the fact that the attempted password was incorrect.

This follows trivially from the ideal functionality–the only thing returned in step 21 of the ideal functionality is $\perp$, which reveals no information about the correct password.

Definition B.2. *Well-Behaved Adversary*
Consider an adversary given tickets $T_{1,2,\ldots,t}$ and passwords $P_{1,2,\ldots,N}$. A well-behaved adversary (WBA) makes queries to the ideal functionality according to the following rules:

(1) Never make a Request query.
(2) For each Unlock query:
  (a) Use a ticket $T \in T_{1,\ldots,t}$.
  (b) Never use the same ticket in more than one Unlock query.
  (c) Use a password $\hat{P} \in P_{1,\ldots,N}$.

(d) Never use the same password in more than one UNLOCK query.

Lemma B.1. A WBA has a probability of at most $\frac{t}{N}$ of having a winning transcript.

Proof:

(1) Every WBA UNLOCK query has a valid ticket. (Definition of WBA (a,b) and Definition of Game (step 5))
(2) Every WBA UNLOCK query returns $(1, \perp)$ if its password is incorrect, and $(1, K_P)$ if its password is correct. (Ideal functionality, lines 26 and 29)
(3) WBA makes at most t queries. (WBA definition, (a,b))
(4) $P[GT] = P[$ correct password appears in one of WBA's UNLOCK queries]. (Implication of step 2.)
(5) $P[$correct password appears in one of WBA's queries$] \leq \frac{t}{N}$
  (a) WBA makes t queries, each with a different password.
  (b) WBA has no better way to choose password guesses than choosing randomly from the list of unused passwords. (Definition of game, Fact B.1.2.).
  (c) Thus, WBA has at best a $\frac{t}{N}$ chance of including the right password in one of its queries.
(6) Thus, a WBA has at most a $\frac{t}{N}$ probability of having a winning transcript. (Previous two steps.)

Lemma B.2. There is no adversary A such that $Pr[WT|A] > Pr[WT|WBA] + 2^{k-n}$.

Proof: By showing that violating any of the five conditions of being a WBA can never raise $P[WT]$ by more than a negligible amount.

(1) Making a REQUEST call never raises $P[WT]$.
  (a) A WT is a transcript in which an UNLOCK returns K_P.
  (b) Tickets generated by a REQUEST permit an UNLOCK with the same payload key as appeared in the REQUEST.
  (c) The value of K_P is selected randomly from all possible n-bit strings.
  (d) The adversary may make at most 2^k REQUEST calls.
  (e) Thus, the probability of the adversary putting the right value of K_P in one of those REQUEST calls (which will enable an UNLOCK call with the right value of K_P) is no greater than 2^{k-n}.
(2) Using a ticket that's not in the valid set of tickets never raises $P[WT]$. (Only tickets $T_{1,2,\ldots,t}$ have payload key K_P.
(3) Reusing a ticket never raises $P[WT]$. (A reused ticket always gets $(\perp, \perp)$.
(4) Using a password that's not in $P_{1,\ldots,N}$ never raises $P[WT]$. (Only an UNLOCK with P_ℓ will get back $(1, K_P)$, any other password will get $(1, \perp)$.)
(5) Reusing a password never raises $P[WT]$. (If the password was used with a valid fresh ticket in a previous UNLOCK call, then future UNLOCK calls with fresh tickets will get the same result. Thus, the probability of K_P appearing in the transcript is never raised.)
(6) Thus, $P[WT|\text{non-}WBA] \leq P[WT|WBA] + 2^{k-n}$. □

Theorem B.3. With the ideal functionality and an uncompromised server, no adversary who can make at most 2^k queries to DV or the ideal functionality can win this game with probability higher than $\frac{t}{N} + 2^{k-n} + 2^{-n}$.

Proof:

(1) **GT:** No adversary has more than $\frac{t}{N} + 2^{k-n}$ probability of getting a winning transcript when interacting with the ideal functionality. (Lemma B.2.)
(2) **LT:** Given a losing transcript, an adversary must make a guess about the correct value of the randomly-selected K_P. The probability that this guess will be correct is 2^{-n} (Game definition with K_P chosen randomly.)
(3) **Union Bound:** The probability of the attacker knowing K_P is thus no higher than $\frac{t}{N} + 2^{k-n} + 2^{-n}$ (Summing GT and LT conditions.)

□

B.2 Server Compromise: Learning the User's Password

Another critical security property of this scheme is that the server must never learn anything about the user's password. We capture this with the following game, in which we assume the server is corrupted:

Security Game: Learn User's Password

(1) The game is parametrized by security parameter n, dictionary size N and number of tickets t, where $t < N$.
(2) The challenger generates two random passwords, P_1, P_2.
(3) The challenger randomly generates a payload key, $K_P \leftarrow \{0, 1\}^n$.
(4) The attacker is allowed to play the role of the server in the protocol.
(5) The challenger honestly runs `ServerSetup` and `RequestTickets` with the attacker playing the role of the server, using password P_1 and payload key K_P. to generate t tickets $T_{1,\ldots,t}$.
(6) The challenger runs UNLOCK using password P_1 and each ticket in succession.
(7) The challenger generates a random bit b, and sends the attacker $P_b, P_{b\oplus 1}$.
(8) The attacker must guess b to win the game.

B.2.1 Practical Relevance. If a compromised server can learn anything about the user's password, then it becomes a major security threat–a single server being compromised might lead to the leak of thousands of users' passwords. If there's no attacker who can win this game with probability more than $\frac{1}{2}$, then the server learns nothing at all about the user's password–not even enough to distinguish the correct password from an incorrect one when given both values. An attacker who can't distinguish correct and incorrect passwords also cannot mount a brute-force password search.

B.2.2 Proof.

Theorem B.4. No attacker can win the Learn User's Password game when the attacker and challenger are interacting via the ideal functionality with probability higher than $\frac{1}{2}$.

Proof: In the ideal functionality, the server never receives any information about the password P provided by the user. With no information about the correct password, the attacker has no strategy better than a random guess for determining b. □

B.3 Server Compromise: Violating the User's Privacy

The user trusts the server to assist her in key derivation, but may not want the server to be able to determine when she is deriving her key. This game captures a critical privacy property–the server must not be able to determine which user is unlocking her key at any given time.

Security Game: Violate User Privacy

(1) The game is parametrized by security parameter n, and number of tickets t.
(2) The challenger generates two random passwords, P_1, P_2, and two payload keys K_P1 and K_P2.
(3) The attacker is allowed to play the role of the server in the protocol.
(4) The challenger honestly runs `ServerSetup`.
(5) The challenger honestly runs Request to generate t tickets with password P_1 and payload key K_P1, identifying itself as user 1.
(6) The challenger honestly runs Request to generate t tickets with password P_2 and payload key K_P2, identifying itself as user 2.
(7) For $i = 1 \ldots t-1$:
  (a) The challenger asks the attacker which user should make the next Unlock call, and whether he should use the right password or not.
  (b) The challenger makes the Unlock call as directed.
(8) The challenger generates a random bit b.
(9) The challenger runs Unlock using password P_b and one of user $b+1$'s tickets.
(10) The attacker must guess b to win the game.

B.3.1 Practical Relevance. We want to guarantee that the user retains privacy from the server–she doesn't give the server the power to track each time she decrypts her hard drive. This game captures the user's privacy goal–an attacker who has compromised the server cannot learn which user ran the Unlock protocol with him in any given instance, even if he knows which user requested each ticket and has observed many other uses of the same password. Note that this assumes that the server isn't able to simply track the user by IP address–network-level anonymization of the user is outside the scope of our work.

B.3.2 Proof. The proof is trivial: the ideal functionality does not inform the server which user is making an Unlock call, so the server dealing with the ideal functionality never learns this information.

C PROOF OF THEOREM 5.1

Before re-stating our theorem, we note that the only random oracle that gets programmed in the proof is H_{VE}.[10] We also assume that honest users securely erase their tickets after an unlock attempt with that ticket has been made (as well as any other part of their state which no longer needs to be stored).

Theorem 5.1. Under the assumption that Π_{ENC} is a CCA2-secure encryption scheme (see Definition A.5), Π_{SIG} is a 2-move blind signature scheme (see Definition A.7) and the assumptions listed in Section 5, the protocols for Setup, Request and Unlock given in Sections 4.1, 4.2 and 4.3, UC-realize the ideal functionality provided in Algorithms 3 and 4 under adaptive corruptions, with erasures.

To prove the theorem, we provide a simulator Sim and prove that the resulting Ideal and Real distributions are computationally indistinguishable. Throughout, we assume that the same ticket (resp. alias) is never issued twice during a Request (resp. Unlock) procedure in an Ideal execution with a single sid. Since each of these events occurs with at most $\lambda'^2/2^n$ probability, where λ' is the total number of tickets issued, this assumption can only reduce the adversarial distinguishing probability by at most $2 \cdot \lambda'^2/2^n$, which is negligible.

C.1 Description of Simulator Sim

Simulator Sim *under adaptive corruptions of parties.* Note that since we assume secure channels, Sim only needs to begin simulating the view at the moment that some party is corrupted.

Fix an environment Env, Server Server, users $\mathcal{U}_1, \ldots, \mathcal{U}_m$ and adversary $\mathcal{A}$. Recall that we allow the environment Env to choose the inputs of all parties. Simulator Sim does the following:

(1) Initialization: Initialize tables $\mathcal{B}, \mathcal{E}, \mathcal{S}, \mathcal{Z}, \mathcal{T}_{\text{gen}}, \mathcal{T}_{\text{used}}$ to empty and counters count_i for $i \in [m]$ to 0.
(2) Preprocessing: Let λ'_i be the maximum number of tickets for each party $\mathcal{U}_i$. For $i \in [m]$, $j \in [\lambda'_i]$: Generate $B^i_j \leftarrow \{0,1\}^n$, $S^i_j \leftarrow \{0,1\}^n$, $Z^i_j \leftarrow \{0,1\}^{2n}$. Add all generated B^i_j (resp. S^i_j, Z^i_j) values to $\mathcal{B}$ (resp. $\mathcal{S}, \mathcal{Z}$). Let λ' be the total number of (B^i_j, S^i_j, Z^i_j) tuples generated.
(3) Responding to corruption requests:
  Corruption of a party $\mathcal{U}_i$: Sim corrupts the corresponding ideal party and obtains its internal state, consisting of unused tickets $t^i_1, \ldots, t^i_{\lambda_i}$. For $j \in [\mathsf{count}_i]$, modify entry $(U^i, S^i_j, B^i_j, E^i_j, F^i_j, Z^i_j, \perp) \in \mathcal{T}$ to $(U^i, S^i_j, B^i_j, E^i_j, F^i_j, Z^i_j, t^i_j)$. For $j \in \{\mathsf{count}_i + 1, \ldots, \lambda_i\}$:
  (a) Generate $E^i_j = \mathsf{ENC}_{PK_S}(B^i_j)$ and F^i_j as a blind signature of E^i_j using SK_S (note that since $\lambda_i - \mathsf{count}_i > 0$, Sim must have already generated $(PK_S, SK_S, PK'_S, SK'_S)$).
  (b) Add $(U^i, S^i_j, E^i_j, F^i_j, Z^i_j, t^i_j)$ to $\mathcal{T}$ and E^i_j to set $\mathcal{E}$.
  Sim releases tickets $(S^i_j, E^i_j, F^i_j, Z^i_j)$.
  Corruption of Server: Sim corrupts the corresponding ideal party and obtains its ideal internal state If an Initialize

[10] We note that for UC composition to hold in the programmable random oracle model, one must, in general, assume that an independent random oracle is used for each sid instance. In our case, we essentially use the programmability of the random oracle to implement a non-committing encryption scheme (see [11]), by adjusting the outcome of H_{VE} to ensure that the string Z_i decrypts to the correct K_P value. Camenisch et al. [7] showed that some natural non-committing encryption schemes in the programmable random oracle model can be proven secure in the UC setting, since the simulator only needs to program the random oracle at random inputs, which have negligible chance of being already queried or programmed. We anticipate that a similar argument would work for our scheme, since D^i_j is unpredictable and with very high probability will not be queried in any other session before being programmed in the target session. However, our formal proof is only for the case where an independent random oracle is assumed for each session.

query has not yet been submitted to the ideal functionality, Sim returns $\perp$. Otherwise, if the server's keys have not yet been sampled, Sim samples $(PK_S, SK_S, PK'_S, SK'_S)$. Let $\alpha_1, \ldots, \alpha_\lambda$ be the aliases in the ideal internal state (if any). Associate each row in $\mathcal{T}_{\text{used}}$ with a random alias so each entry in $\mathcal{T}_{\text{used}}$ contains a value from $\{\alpha_1, \ldots, \alpha_\lambda\}$ in its final column. For $i \in [\lambda - |\mathcal{T}_{\text{used}}|]$, Generate $B_i \leftarrow \{0,1\}^n$, $E_i = \mathsf{ENC}_{PK_S}(B_i)$ and F_i as a blind signature of E_i. Add all tuples $(B_i, E_i, F_i, \alpha_i)$ to $\mathcal{T}_{\text{used}}$. For each row of $\mathcal{T}_{\text{used}}$, release (E_i, F_i).

(4) Responding to random oracle queries to $H_{\text{pw}}, H_{\text{KD}}$: Sim forwards the query to the oracle and forwards the response back.

(5) Responding to random oracle queries to H_{VE}: Sim maintains a table $\mathcal{T}_{H_{\text{VE}}}$. The table is initialized as empty. Each time $\mathcal{A}$ queries H_{VE} on input x, Sim checks the table to see if an entry of the form (x, y) appears in the table for some y. If yes, Sim returns y. Otherwise, Sim chooses a random y, adds entry (x, y) to $\mathcal{T}_{H_{\text{VE}}}$ and returns y to A.

(6) When responding to oracle queries, Sim also does the following:
- **Bad Event 1:** If Server is corrupted and $\mathcal{A}$ makes a query to H_{pw} with input of the form $S^i_j||\hat{P}^i_j$, where $S^i_j \in \mathcal{S}$ and $(S^i_j, \cdot, \cdot, \cdot, t^i_j) \notin \mathcal{T}$ (for $t^i_j \neq \perp$) then Sim aborts.
- **Bad Event 2:** If Server is not corrupted and $\mathcal{A}$ makes a query to H_{KD} with input of the form $(B^i_j||\hat{C}^i_j)$, where $B^i_j \in \mathcal{B}$, then Sim aborts.
- If Server is corrupted and $\mathcal{A}$ makes a query to H_{pw} with input of the form $S^i_j||\hat{P}^i_j$ where $S^i_j \in \mathcal{S}$, Sim finds the tuple of the form $(S^i_j, \cdot, \cdot, \cdot, t^i_j) \in \mathcal{T}$ and submits $\textsc{Unlock}(\textsc{sid}, t^i_j, \hat{P}^i_j)$ to the ideal functionality. Sim receives $(\textsc{Unlock}, \textsc{sid}, \alpha)$ from the ideal functionality, and returns $(\textsc{sid}, \textsc{Unlock}, 1)$. If the ideal functionality returns $\perp$, Sim forwards $\hat{C}^i_j = H_{\text{pw}}(S^i_j||\hat{P}^i_j)$ to $\mathcal{A}$. If the ideal functionality returns K_P, Sim computes $\hat{C}^i_j = H_{\text{pw}}(S^i_j||\hat{P}^i_j)$, $D^i_j = H_{\text{KD}}(B^i_j||\hat{C}^i_j)$ and entries for $(0||D^i_j, y_1), (1||D^i_j, y_2)$ such that $y_1||y_2 = Z^i_j \oplus (0^n, K_P))$ to $\mathcal{T}_{H_{\text{VE}}}$. Sim returns $\hat{C}^i_j$ to A. **Bad Event 3:** If at this point $0||D^i_j$ or $1||D^i_j$ have already been queried to H_{VE}, Sim aborts.

(7) Responding to messages from the Request protocol issued by a corrupted $\mathcal{U}_i$ when Server is not corrupted. Sim does the following:
  (a) Generate $(PK_S, SK_S, PK'_S, SK'_S)$ if not yet generated.
  (b) Submit $\textsc{Request}(\mathcal{U}_i, \textsc{sid}, 0, 0)$ to the ideal functionality and receive back ticket t.
  (c) Place $(U_i, *, *, *, *, t) \in \mathcal{T}_{\text{gen}}$.
  (d) Play the part of an honest signer with secret key SK'_S in the blind signature protocol with the corrupted user.

(8) Responding to $(\textsc{sid}, \textsc{Request}, U_i)$ messages from Ideal Functionality. Sim does the following:
  (a) Set $\mathsf{count}_i := \mathsf{count}_i + 1$ and $j := \mathsf{count}_i$.
  (b) Generate $E^i_j := \mathsf{ENC}_{PK_S}(B^i_j)$.
  (c) Participate in a blind signature protocol on message E^i_j with the corrupted Server to obtain signature F^i_j.
  (d) Store $(U_i, S^i_j, B^i_j, E^i_j, F^i_j, Z^i_j, \perp) \in \mathcal{T}_{\text{gen}}$.

(9) Responding to messages from the Unlock protocol issued by adversary $\mathcal{A}$ when Server is not corrupted. $\mathcal{A}$ sends $(\hat{E}, \hat{F}, \hat{C})$ to the server.
- If a tuple of the form $(\cdot, \hat{E}, \cdot, \hat{t}, *) \in \mathcal{T}_{\text{used}}$, then send $\textsc{Unlock}(\textsc{sid}, \hat{t}, \perp)$ to the ideal functionality.
- Otherwise, if the signature does not verify submit $\textsc{Unlock}(\textsc{sid}, \perp, \perp)$ to the ideal functionality.
- Otherwise, if $\hat{E} = E^i_j \in \mathcal{E}$:
  (a) Find an entry of the form $(\cdot, \cdot, \hat{E}, \cdot, \hat{t}) \in \mathcal{T}$. Add $(\hat{B}, \hat{E}, \hat{F}, t, *)$ to $\mathcal{T}_{\text{used}}$.
  (b) **Bad Event 4:** If there is more than one oracle query that returned $\hat{C}$, Sim aborts.
  (c) If the unique query exists, extract the password guess $\hat{P}$ (with bit length at most n'). If it does not exist, set $\hat{P}$ to $\perp$. Send $\textsc{Unlock}(\textsc{sid}, \hat{t}, \hat{P})$ to the ideal functionality. **Bad Event 5:** If $\hat{C} = H_{\text{pw}}(S^i_j, *)$, for some $S^i_j \in \mathcal{S}$, but $\mathcal{A}$ did not make an oracle query returning $\hat{C}$, Sim aborts.
  (d) If the ideal functionality returns a value K_P, then set $D^i_j = H_{\text{KD}}(B^i_j||\hat{C})$. Add $(0||D^i_j, y_1), (1||D^i_j, y_2)$ to $\mathcal{T}_{H_{\text{VE}}}$ such that $y_1||y_2 = Z^i_j \oplus (0^n, K_P))$ Return D_j to $\mathcal{A}$. **Bad Event 6:** If $\mathcal{A}$ has already queried H_{VE} on $0||D^i_j$ or $1||D^i_j$, Sim aborts.
  (e) Otherwise, return $D^i_j = H_{\text{KD}}(B^i_j||\hat{C}^i_j)$.
- Otherwise if $\hat{E} \notin \mathcal{E}$, Sim does the following:
  (a) **Bad Event 7:** If there is no entry of the form $(*, *, *, *, \hat{t}) \in \mathcal{T}$, Sim aborts.
  (b) Find an entry of the form $(*, *, *, *, \hat{t}) \in \mathcal{T}$ and remove it.
  (c) Decrypt $\hat{E}$ using SK_S to obtain $\hat{B}$. **Bad Event 8:** If $\hat{B} \in \mathcal{B}$, Sim aborts.
  (d) Make an Unlock request to the ideal functionality $\textsc{Unlock}(\textsc{sid}, \hat{t}, \perp)$
  (e) Continue the execution honestly to recover $\hat{D} = H_{\text{KD}}(\hat{B}||\hat{C})$. Return $\hat{D}$ to $\mathcal{A}$.

(10) Responding to $(\textsc{Unlock}, \textsc{sid}, \alpha)$ messages from Ideal Functionality. If Sim receives a message $(\textsc{sid}, \textsc{Unlock}, \alpha)$ (which does not stem from an Unlock request submitted by Sim) then Sim does the following:
  (a) If there is some $(\hat{B}, \hat{E}, \hat{F}, *, \alpha) \in \mathcal{T}_{\text{used}}$. Then Sim forwards $(\hat{E}, \hat{F})$ to Server, along with a random value for $\hat{C}$.
  (b) If not, update the next tuple of the form $(\hat{B}, \hat{E}, \hat{F}, *, \perp) \in \mathcal{T}_{\text{used}}$, to $(\hat{B}, \hat{E}, \hat{F}, *, \alpha)$. Forward $(\hat{E}, \hat{F})$ to Server, along with a random value for $\hat{C}$.
  (c) If Server returns $\perp$, then return 0 to the ideal functionality.
  (d) Otherwise, Sim receives back a D value from Server and checks whether D was computed correctly with respect to $\hat{B}$ and $\hat{C}$. If yes, Sim sends $(\textsc{sid}, \textsc{Unlock}, 1)$ to the ideal functionality. Otherwise, Sim sends $(\textsc{sid}, \textsc{Unlock}, 0)$ to the ideal functionality. If tuples of the form $(0||D, y_1)$, $(1||D, y_2)$ are not in $\mathcal{T}_{H_{\text{VE}}}$, Sim chooses random y_1, y_2 and adds $(0||D, y_1), (1||D, y_2)$ to $\mathcal{T}_{H_{\text{VE}}}$. **Bad Event 9:** If $y_1||y_2 \oplus Z = 0^n||*$, for some $Z \in \mathcal{Z}$, Sim aborts.

We begin by bounding the probability that the Bad Events occur. It is clear by inspection that Bad Event 1 occurs with probability at most $q \cdot \lambda'/2^n$, and that Bad Event 4 occurs with probability at most $q^2/2^n$, where q is the total number of oracle queries made by the adversary and Sim. Moreover, it is clear that if Bad Event 2 does not occur, then Bad Events 3 and 6 occur with probability at most $q^2/2^n$ each. We proceed to bound the remaining events (Events 2, 5, 7, 8).

Lemma C.1. Bad Event 5 occurs with at most negligible probability in the Ideal experiment.

We upper bound the probability of Bad Event 5 by analyzing the probability that $\hat{C} = H(S_j^i, x)$, for some value of $x \in \{0,1\}^{n'}$. This probability can be upper bounded by $\frac{2^{n'}}{2^n}$, since there are $2^{n'}$ possible strings of the form $S_j^i||x$ and each of these gets mapped to a particular string $\hat{C}$ with probability $\frac{1}{2^n}$. Setting parameters appropriately, we have that $\frac{2^{n'}}{2^n}$ is negligible.

Lemma C.2. Assuming the CCA2 security of encryption scheme ENC (see Definition A.5), the probability that Bad Event 2 or Bad Event 8 occurs is at most negligible in the Ideal experiment.

The proof proceeds by showing that if Bad Event 2 or Bad Event 8 occurs with non-negligible probability, then there must be some $i \in [m]$, $j \in [\lambda'_i]$ and efficient Env, $\mathcal{A}$ (who did not corrupt Server) such that $\mathcal{A}$ queries H_{KD} on the value, B_j^i, or, in an Unlock request, sends an encryption $\hat{E} \notin \mathcal{E}$ that decrypts to B_j^i, with non-negligible probability. We will use Env, $\mathcal{A}$ to obtain another efficient adversary $\mathcal{A}'$ who breaks the security of the CCA2 encryption scheme ENC.

The adversary $\mathcal{A}'$ breaking the CCA2 security of the encryption scheme ENC proceeds as follows: $\mathcal{A}'$ plays the part of Sim in the Ideal experiment, with the exception that (1) It knows all the honest users passwords and keys (since it controls Env); (2) It receives PK_S externally from its CCA2 challenger (and does not know the corresponding SK_S), (3) It aborts and outputs 0, 1 with probability 1/2 if $\mathcal{A}$ requests a Server corruption. Sim chooses random strings $B_j^i, B'^i_j\mathcal{B}$. Upon corruption of party $\mathcal{U}_i$, $\mathcal{A}'$ Sim sends B_j^i, B'^i_j back to its CCA2 challenger. The CCA2 challenger chooses $\tilde{b} \leftarrow \{0,1\}$ and returns an encryption of B_j^i if $\tilde{b} = 0$ and an encryption of B'^i_j if $\tilde{b} = 1$. Let E^* denote the challenge ciphertext that $\mathcal{A}'$ receives in return. $\mathcal{A}'$ continues to play the part of Sim, but includes challenge ciphertext E^* in the information returned for the corruption request for party $\mathcal{U}_i$, instead of a newly generated ciphertext. When responding to Unlock queries $(\hat{E}, \hat{F})$, Sim must decrypt using SK_S if $\hat{E} \notin \mathcal{E}$. But in this case, either (1) $\mathcal{A}'$ has not yet requested/received its challenge ciphertext from the CCA2 challenger or (2) $\hat{E} \neq E^*$, since $E^* \in \mathcal{E}$. So $\mathcal{A}'$ forwards the decryption query $\hat{E}$ to its CCA2 oracle. Recall that throughout the experiment, $\mathcal{A}'$ (playing the part of Sim) monitors all queries made to the random oracles. If an Unlock request is made with a valid ticket that includes E^* and a $\hat{C}_j^i$ value corresponding to the correct password, $\mathcal{A}'$ chooses a value for D_j^i at random (without querying oracle H_{KD}). If, at any point, **Case 1:** a query to H_{KD} of the form $(B_j^i, *)$ is made or some CCA2 decryption oracle query yields value B_j^i, then $\mathcal{A}'$ aborts the experiment and returns 0 to its challenger. If, at any point, **Case 2:** a query to H_{KD} of the form $(B'^i_j, *)$ is made or some CCA2 decryption oracle query yields value B'^i_j, then $\mathcal{A}'$ aborts the experiment and returns 1 to its challenger. If the experiment completes without the above cases occurring, $\mathcal{A}'$ flips a coin and returns the outcome to its challenger.

Now, note that if Bad Event 2 or 8 occur with non-negligible probability $\rho = \rho(n)$, then we must have that $\Pr[\tilde{b} = 0 \wedge$ **Case 1** occurs $] = \Pr[\tilde{b} = 1 \wedge$ **Case 2** occurs $] = \rho/2$.

On the other hand, it is always the case that $\Pr[\tilde{b} = 0 \wedge$ **Case 2** occurs $] = \Pr[\tilde{b} = 1 \wedge$ **Case 1** occurs $] = q/2^{n+1} + \lambda'/2^{n+1}$, where q is the total number of distinct oracle queries made during the experiment. This is because when $\tilde{b} = 0$, there is no information at all about B'^i_j contained in adversary $\mathcal{A}$'s view (unless $B'^i_j = B^{i'}_{j'}$ for some $(i', j') \neq (i, j)$, which occurs with probability at most $\lambda'/2^{n+1}$) and so $\mathcal{A}$ can only happen to query the oracle on B'^i_j at random. The case for $\tilde{b} = 1$ follows by identical reasoning.

Thus, the distinguishing advantage of CCA2 adversary $\mathcal{A}'$ is $\rho/2 - q/2^{n+1} - \lambda'/2^{n+1}$, which is non-negligible, since ρ is non-negligible. This implies a contradiction to the CCA2 security of the underlying encryption scheme.

Lemma C.3. Assuming the unforgeability of the blind signature scheme (see Definition A.7), Bad Event 7 occurs with at most negligible probability in the Ideal experiment.

The proof proceeds by showing that if Bad Event 7 occurs with non-negligible probability for some efficient adversary A, then, by definition, we obtain an efficient adversary $\mathcal{A}'$ who submits a larger number of valid Unlock requests than there are valid tickets obtained from the ideal functionality. But note that each valid Unlock request is accompanied by a fresh blind signature $\hat{F}$. Moreover, the number of valid signatures obtained from the signer corresponds to the number of valid tickets obtained. Thus, adversary $\mathcal{A}$ can be used to obtain adversary $\mathcal{A}'$ such that, according to Definition A.7, breaks the security of the blind signature scheme.

Conditioned on the Bad Events not occurring, the only difference between a Real and Ideal execution, is that in the Ideal execution in Step (10b) the simulator submits the next available $(\hat{E}, \hat{F})$ pair, whereas in the Real execution the order of submitted $(\hat{E}, \hat{F})$ pairs depends on which party is making the Unlock request. However, the blindness property of the blind signature scheme ensures that given a set of interactions and message signature pairs, the signer cannot tell in which order the message signature pairs were generated. Indeed, this is the case even when (PK'_S, SK'_S) are adversarially generated. Thus, the view of the adversary is indistinguishable in the two cases. We therefore conclude with the following lemma.

Lemma C.4. Assuming the blindness of the blind signature scheme (see Definition A.7), the Ideal and Real output distributions are computationally indistinguishable.