Single-Message Credential-Hiding Login

Kevin Lewi*, Payman Mohassel†, and Arnab Roy‡

*Novi Research
†Facebook
‡Fujitsu Laboratories of America

Abstract

The typical login protocol for authenticating a user to a web service involves the client sending a password over a TLS-secured channel to the service, occasionally deployed with the password being prehashed. This widely-deployed paradigm, while simple in nature, is prone to both inadvertent logging and eavesdropping attacks, and has repeatedly led to the exposure of passwords in plaintext.

Partly to address this problem, symmetric and asymmetric PAKE protocols were developed to ensure that the messages exchanged during an authentication protocol reveal nothing about the passwords. However, these protocols inherently require at least two messages to be sent out: one from each party. This limitation hinders wider adoption, as the most common login flow consists of a single message from client to the login server. The ideal solution would retain the password privacy properties of asymmetric PAKEs while allowing the protocol to be a drop-in replacement into legacy password-over-TLS deployments.

With these requirements in mind, we introduce the notion of *credential-hiding login*, which enables a client to authenticate itself by sending a *single message* to the server, while ensuring the correct verification of credentials and maintaining credential privacy in the same strong sense as guaranteed by asymmetric PAKEs. We initiate a formal study of this primitive in the Universal Composability framework, design and implement a practical password-based protocol using identity-based encryption, and report on its performance. We also construct a variant of credential-hiding login for fuzzy secrets (e.g. biometrics), proven secure based on the Learning With Errors (LWE) assumption.

1 Introduction

In a traditional password-based account login protocol over a secure TLS channel, a client submits their username and password combination to a login server, which checks the password against a hashed record of the password established upon account registration. The database containing these hashed records is typically secured using a mixture of techniques including the use of a slow password hashing algorithm along with a salt [PM99, MKR17, Per09, BDK16], and encryption on top of the hashed records under keys stored in a separate database [Muf15, ECS+15, BLMR13, AMMR18]. These methods were designed primarily to protect passwords in the event of a server compromise.

Nevertheless, there have been a recurring series of incidents from major technology companies that have self-reportedly stored passwords in plaintext, including GitHub [Git18] and Twitter [Twi18] in 2018, and Facebook [Fac19], Google [Goo19], Robinhood [Rob19], and Coinbase [Coi19]

in 2019. In each of these incidents, user passwords underwent hashing server side before being stored, and yet were still persisted in plaintext in some form, frequently through inadvertent logging during login time.

To avoid the logging of plaintext passwords when users log in, servers can employ client-side cryptographic solutions. A simple approach, which we will refer to as "Hash-then-Encrypt" (HtE), requires that clients first hash their passwords, and then encrypt the resulting hash under a single public encryption key for which the server has access to the corresponding decryption key, before sending the resulting payload to the server. The central issue with this approach, which we refer to as "HtE-over-TLS", is that confidentiality of any logged transcript is lost if the login server's decryption key (which needs to constantly persist on the login server) is ever exposed since the same decryption key exposes all hashed passwords to an adversary who can run an offline dictionary attack on any previously logged transcript.[1] These privacy implications are even more pronounced when we consider legitimate but unsuccessful login attempts wherein registered users mistakenly use their credential from one service to login to another, running the risk of having their hashed password decrypted and logged by the wrong server.

Password authenticated key exchange. A promising approach to addressing the confidentiality of passwords during login involves using an asymmetric password authenticated key exchange (aPAKE) protocol. In an aPAKE, a client who holds *only* a password (as opposed to an authentic public key from the server) can establish a secure channel with a server who holds a one-way function of the password. When applied to the account login scenario, the client and server can exchange messages which allow the server to learn whether or not a client has the correct login password without revealing the contents of the password attempt to the server. Using an aPAKE for password-based login is essentially the state-of-the-art solution for mitigating the security issues with persisting the password in a database and handling the credentials during user login.

Crossing the application and network layer boundary. One approach for using aPAKEs for user login, which we refer to as the "aPAKE-into-TLS" approach, involves using aPAKE to bootstrap the TLS session as described in [SKFB19]. Taking the aPAKE-into-TLS approach for user login means that we can benefit from the improved security for passwords without introducing any extra rounds of communication between client and server. However, the main disadvantage is that it requires the application server to tightly integrate its password handling logic with its TLS session setup and management logic. Typically, the servers which handle TLS channel setup and maintainence sit at the edge of the service's network boundary, and have been heavily optimized to reduce latency as much as possible. In other words, this kind of integration requires a merging of functionalities between the network layer and the application layer, which is not always possible or desirable.

In order to maintain the separation between network logic and application logic, we can also consider the "aPAKE-over-TLS" approach, in which the TLS session is first established, *and then* the aPAKE exchange is executed between client and server, purely within the application layer. However, we note that when comparing this solution to the HtE-over-TLS approach, the aPAKE protocol itself introduces an extra round of interaction between the client and server, whereas

[1] This kind of attack holds for any efficiently computable deterministic function applied to the user password before it is sent to the server.

HtE-over-TLS only requires the client to send a *single message* to the server based on some public parameters.

Minimizing round complexity for login. The introduction of the extra message incurred by an aPAKE-over-TLS approach, while acceptable in some scenarios, introduces an overall latency and reliability hit that is exacerbated by low-connectivity environments, and from a purely infrastructural perspective, can pose a challenge when attempting to integrate with systems that are built around the assumption that password-based login can conclude after a single message from the client. In general, the extra round trip required for migration from a HtE-over-TLS login form to an aPAKE-based login form is an undesirable side effect of the security benefits of using an aPAKE.

While an extra round trip seems like a fair price to pay for the added security benefits provided by aPAKEs compared with using HtE (both over TLS), it is often the case, especially in countries with slow networks, that this extra round trip results in seconds of delay during the login protocol. Based on a study on mobile performance optimization [Eve13], just two seconds of latency increase is enough to double the abandonment rate of users, and every additional 100 milliseconds of delay can cause a noticeable impact on revenue for large companies with user populations in low-connectivity regions.

A natural question we ask is if the additional round complexity of PAKEs is truly necessary in order to capture the desired security properties for account login. Typically, in a user login setting between a client and server on the web, a secure channel has already been established through the TLS certificate of the website and the public key infrastructure that it relies on. This means that the protections to address adversaries who aim to impersonate the server to the client do not need to be incorporated into the login protocol.

Given the distinctions between the login-within-TLS setting and PAKE, we introduce and formalize the notion of credential-hiding login, in which our aim is to formally define and capture a primitive which can act as a drop-in improvement in credential privacy for the HtE-over-TLS approach that most companies employ today, without resorting to altering infrastructure to support multiple rounds and the increase to latency that it causes.

1.1 Revisiting the Login Problem

Note that in the single-message setting, there are inherent limitations to the amount of privacy that can be guaranteed for the client's login message to the server. The first issue is that the login server is able to learn information about the credential encoded in this message by simply testing it against any existing registered password files in its database. This is inherent to the functionality of the login scheme, and so this leakage cannot be avoided. The second issue is that the login server can essentially execute a brute-force dictionary attack by running the registration procedure to obtain password files for its password guesses, allowing it to again learn information about the password encoded in the message, even if the password has not been registered previously. Given this attack, it seems that we cannot do any better than HtE in terms of providing confidentiality (since HtE already forces an adversarial server to perform an offline dictionary attack to learn information about the encoded password).

In practice, however, there are usually significant distinctions between the flows which handle user registration and the flows which allow users to log in. For one, the login attack surface

is usually much larger, in terms of frequency (registrations often happen at a frequency orders of magnitude lower than login) and the overall complexity of different mechanisms necessary for login. Furthermore, it is usually the case that the servers that handle registration can be separated from the servers that handle login and their secrets be protected using hardware solutions due to the less stringent latency requirements.

Separating registration from login. In order to provide a more meaningful and applicable approach to defining privacy for login credentials, we capture this intuition formally by allowing the registration procedure to accept secret parameters that the login procedure is unable to access. This means that the registration server can embed secrets into the password file which the login server cannot generate on its own, thereby avoiding the impossibility result stated above. Of course, we still require that the login server be able to determine whether or not a password attempt should succeed *without* access to these secret parameters, and without any interaction with the registration server.

Going beyond passwords. In addition, we apply this framework to handle authentication schemes that are more complicated than checking if a login password is equal to the password registered to an account. Although many prior works have investigated the feasibility of biometric authentication (in which a distance metric is computed between two vectors to determine whether or not they are "close" enough), these works present schemes which do not satisfy the security properties we will lay out for credential-hiding login.

As a result, we believe that if these biometric authentication schemes were deployed in practice, they would in fact provide less security (in terms of protecting the credential itself) than a traditional password-based login scheme, while arguably being an even more important and sensitive type of credential to protect than user passwords.

Our goal with credential-hiding login is to ensure that we can design biometric (and other types of credentials) authentication schemes which piggyback on the benefits of decades of research work that have led to the state-of-the-art best practices for protecting passwords, instead of rediscovering the same pitfalls of insecure credential treatment from the password security literature.

Our central focus in this work is to study a primitive which represents the highest level of security we can achieve for login while also taking into account low-latency practical settings in which multiple rounds of interaction are a nonstarter. To do so, we take into account the distinction made between the registration server and the login server, and the separation between the server validating credentials versus the one performing a key exchange (unlike the aPAKE-into-TLS approach), and define a new primitive, called credential-hiding login (CHL), which realizes the security and efficiency properties we look for in an account login protocol. In Table 1, we illustrate the benefits of using CHL-over-TLS compared with the other approaches discussed previously in this section.

1.2 Credential-Hiding Login Schemes

A single-message credential-hiding login scheme consists of a setup, registration, client login, and server login procedure. The setup algorithm is run to produce public parameters that are issued to clients, typically after already establishing a secure channel between client and server. The registration algorithm is executed between the client and a registration server, and in the simplest case, involves the client sending their credential to the registration server in order for the server to

Login Protocol	Min # of Messages	OSI Separation	Security
HtE-over-TLS	3	✓	Weak
aPAKE-into-TLS	3	✗	Strong
aPAKE-over-TLS	4	✓	Strong
CHL-over-TLS	3	✓	Strong

Table 1: A comparison of the various approaches discussed for handling password-based login. Note that OSI Separation refers to whether or not the protocol respects the separation of application logic from network logic according to the Open Systems Interconnection Model. Our work is described by the last row of the table.

compute the password file. Then, when a client is asked to log in to their account, they run the client login algorithm to produce a credential-embedded message that they send to a login server (credential verifier). The login server then executes the server login algorithm to verify this message and determine whether or not the login attempt is successful.

We formalize the notion of a single-message credential-hiding login scheme in Section 2, along with a security definition under the Universal Composability (UC) framework. Informally, we highlight the two main security properties we capture in our UC definition of security:

- **Credential privacy**, which ensures that a client's message representing the credentials are independent of the credentials, so long as the adversary does not have access to the client's password file or if the client has not registered this particular credential with the registration server.
- **Credential verifier security**, which captures the notion that an adversary who has access to the server-side storage of the user's credentials established during registration would still need to execute brute-force dictionary attacks to recover the underlying credentials, even if all secrets in the scheme have been leaked.

We note that these properties alone are not enough to satisfy the formal UC definition we present in Section 2, but merely serve as a more intuitive framing of the requirements that a credential-hiding login scheme aims to satisfy.

Satisfying credential privacy implies that the client login algorithm produces messages which do not encode information about the password from the login server's perspective. This stands in contrast to the HtE approach, where the login server uses a master decryption key to effectively remove all entropy from the client login's message. A credential-hiding login scheme ensures that the login server is unable to remove this entropy without access to the client's password file.

Our constructions leverage hash functions that we model as random oracles to satisfy credential verifier security. Although not ideal, such a non-black box assumption is required to detect offline guesses from the adversary and has been traditionally used in UC aPAKEs [GMR06, JKX18, JR18]. We can instantiate these hash functions with computationally-expensive and memory-hard hash functions such as scrypt [PJ16], Argon2 [BDK16] or balloon hashing [BCS16] in order to increase the difficulty of executing an offline dictionary attack on credentials which come from a low-entropy distribution.

Defining CHL under universal composability. We model our UC definition for credential-hiding login along similar lines as aPAKEs, but with several significant distinctions. For one, aPAKEs aim to establish a shared secret, whereas CHL simply aims for login success or failure. Also for aPAKEs, a client only needs to assume that the public parameters come from a trusted source, whereas in CHL, the public parameters come from the registration server which also holds a secret corresponding to this parameter. While we do not allow the registration server to generate these parameters maliciously, we do allow the adversary the choice to obtain the secret upfront. This distinction in our setting seems intrinsic to our design, as we elaborate in more depth in Section 2.3. Furthermore, we do not model pre-computation attacks for CHL, as is done by strong aPAKEs such as OPAQUE [JKX18]. And finally, our definition of CHLs accommodates more general credential verification predicates, rather than just equality checks for passwords.

To reiterate, these properties are not new in the password-based authentication literature—indeed, many works have defined similar notions for authentication protocols which capture a subset or superset of the properties we study here. However, our aim is to focus specifically on the most important properties which are desirable in a *login protocol* (as opposed to key exchange) and provide a framework that enables future works to explore extensions to more general credentials beyond passwords and equality checks.

Authenticated login and composition with TLS. In Section 2.4, we define a simpler and more general abstraction of the CHL ideal functionality, which we refer to as "authenticated login". The authenticated login definition abstracts away passwords and credentials analogously to standard key exchange versus password-based key exchange. We show that the UC notion of secure channels can also be obtained in the authenticated login UC hybrid model.

However, the typical deployment of authenticated login in practice that we envision within TLS *does not require such a bootstrapping to secure channels*. We instead focus on leveraging authenticated login to establish the authenticity of the user and the continuation of the subsequent interactions based on the *already established* server-authenticated TLS channel. Recall that this corresponds to the CHL-over-TLS description given earlier. Although this suffices for the primary applications that we target (namely, a login form over the web), we note that this model provides weaker guarantees than a standard secure channel, as it does impose on the user application to not send any sensitive content which assumes mutual trust before the authenticated login flow is completed.

1.3 Our Contributions

In this work, we initiate the study of credential-hiding login with a concrete formulation of security in the UC framework, formalized in Section 2 along with the definition of authenticated login. Additionally, we present:

- A construction Π_{pwd} from anonymous identity-based encryption that handles password credentials,
- An implementation and concrete instantiation of Π_{pwd} along with benchmarks, and
- A generalized scheme Π_{fz} which achieves credential-hiding login for a set of distance functions including Hamming distance based on the Learning With Errors (LWE) assumption.

Credential-hiding login for passwords. For Π_{pwd}, the password verifier (i.e. password file) for a password y consists of two components: an IBE secret key for the identity y, along with a one-way function of a random oracle evaluation of y. The key intuition here is that we use the password verifiers to hold IBE secret keys, and the client login algorithm produces IBE ciphertexts where the identity is set to the password y and the payload being encrypted is a random oracle output of y. Then, when the server wants to verify the credential, it can retrieve the password verifier containing the IBE secret key and attempt to decrypt the ciphertext. Successful decryption occurs only if the password tied to the password verifier matches the password embedded in the ciphertext, as desired. Credential privacy follows from the anonymity and IND-CCA security of the IBE scheme; in particular, we need the property that the ciphertexts completely hide both the identity and the payload when the corresponding secret key which can decrypt the ciphertext is not present. This allows us to prove that credential login transcripts without a corresponding password file truly hide the credential from the login server. Furthermore, the use of a random oracle output for the scheme ensures that even if the registration key is leaked along with the entire password verifier database, an attacker would still need to run a brute-force attack on the hash function we model after a random oracle in order to recover any plaintext passwords. A formal proof of these properties, along with a complete description of Π_{pwd} can be found in Section 3.

We also provide a concrete instantiation of Π_{pwd} in Section 4, along with performance benchmarks which show that incorporating Π_{pwd} into an existing login protocol is indeed realistic. We emphasize that this protocol offers better security when compared to the traditional hash-then-encrypt approach, while also avoiding multiple-message flows that are inherent with aPAKEs. Although our implementation serves as a stepping stone towards realizing credential-hiding login in practice, it still remains primarily as a proof-of-concept, and we hope that it encourages further improvements to the efficiency of credential-hiding login schemes, as well as more production-ready integrations into existing certificate-based server authentication protocols.

Going beyond password credentials. In Section 5, we present our generalized Π_{fz} construction, which leverages obfuscation primitives while still being provably secure from "standard" assumptions (albeit with the use of a random oracle). Although this construction is significantly less practical than Π_{pwd}, it establishes the feasibility of achieving credential-hiding login for more complex credentials than passwords, and that further research work in this area could directly benefit the field of biometric (and other) forms of authentication without having to reinvent and redefine the security principles that the community has established for password-based authentication.

Recently, Erwig et al. [EHOR20] showed how to construct a fuzzy aPAKE from error correcting codes and oblivious transfer. However, for an n-bit credential, their construction only handles $O(\log n)$ errors, essentially by enumerating through all possible credentials that are close to the original registered credential. Our construction Π_{fz} obtains its error rate from the underlying fuzzy extractor, which is not bound by this brute-force technique and can handle a linear fraction of errors while also being asymptotically more efficient.

We conclude in Section 6 with a list of interesting directions for future work.

1.4 Related Work

In the following, we review some of the related areas of research which could be applied to the login scenario.

Password authenticated key exchange. As discussed previously, password-based authenticated key exchange is the most promising prior work for addressing confidentiality of passwords in a login protocol. Symmetric PAKE where both parties hold a low entropy password and want to establish a secure channel was first studied by Bellovin and Merritt [BM93a], and later formalized by Bellare et al. [BPR00] using the game-based indistinguishability approach. Canetti et al. [CHK+05] provided the first formalization of PAKE in the UC framework [Can01]. The asymmetric variant of PAKE where one party (typically the server) only stores a one-way function of the password to protect against server compromise was first introduced by Bellovin and Merritt [BM93b] and later formalized and studied in the simulation-based paradigm [BMP00, Mac01, MPS00]. Several follow-up works study aPAKE in the UC framework [GMR06, JR18], while the recent work of [JKX18] enhances aPAKE by providing protection against pre-computation attacks.

A number of PAKE constructions [AP05, HR10, HS14, HL19, PW17, JKX18] are also being discussed and reviewed by the Crypto Forum Research Group (CFRG) [Gro20] for standardization, with OPAQUE being the winner for aPAKEs and CPace the winner for (balanced) PAKEs.

Kiefer and Manulis [KM14] developed the notion of ZKPPC (zero-knowledge password policy checks) to enable the registration server to check if the client password satisfies a certain policy, while being oblivious of the raw password itself. Although their setting shares some similarities with applications of credential-hiding login in the use of a server-authenticated secure channel, we leave the construction of an analogous ZKPPC for credential-hiding login as an open problem.

Private biometric authentication. Protecting privacy of biometric credentials is even more critical given that they are impossible to change. A line of work building on information reconciliation [BBR88] and fuzzy extractors [DORS08, BDK+05, Boy04, DS05, CFP+16a, WZ17] propose potential solutions to this problem but they are limited in the biometric distributions and distances they support, do not provide password confidentiality during the login process, and do not provide protection against offline dictionary attacks (even in the presence of eavesdroppers).

In the multi-round setting, a line of work based on secure two-party computation [OPJM10, EHKM11, HEKM11, BCF+14], and the recent generalization of PAKE to the fuzzy setting [DHP+18] move closer to the credential privacy guarantees we aim for but are not secure in the face of server compromise (where the server stores the plaintext biometrics), and require three or more rounds of interaction between the client and the server.

2 Definitions

For a credential domain $\mathcal{X}$ and verifier domain $\mathcal{Y}$, a credential-hiding login scheme $\Pi = (\mathsf{Setup}, \mathsf{Register}, \mathsf{ClientLogin}, \mathsf{ServerLogin})$ for a circuit $\mathcal{C} : \mathcal{X} \times \mathcal{X} \to \{0, 1\}$ consists of four algorithms defined as follows:

- $\mathsf{Setup}(1^\lambda) \to (\mathsf{pp}, \mathsf{sk})$. The Setup algorithm outputs public parameters pp and a secret key sk.
- $\mathsf{Register}(\mathsf{sk}, \mathsf{uid}, \mathsf{y}) \to \gamma$. The $\mathsf{Register}$ algorithm takes as input the secret key sk, a unique identifier uid, and a credential $\mathsf{y} \in \mathcal{X}$, and outputs a verifier $\gamma \in \mathcal{Y}$.
- $\mathsf{ClientLogin}(\mathsf{pp}, \mathsf{ssid}, \mathsf{tok}, \mathsf{uid}, \mathsf{x}) \to \alpha$. The client login algorithm $\mathsf{ClientLogin}$ takes as input a set of public parameters pp, a subsession id ssid, a token tok, an identifier uid, an input $\mathsf{x} \in \mathcal{X}$, and outputs a message α.

- $\mathsf{ServerLogin}(\mathsf{pp}, \mathsf{ssid}, \gamma, \alpha) \rightarrow \{\mathsf{tok}, \perp\}$. The server login algorithm ServerLogin takes in the public parameters pp, a subsession id ssid, a verifier γ, the client message α, and outputs either a token tok or $\perp$.

Terminology. We use uid to represent a user identifier, sid to represent a session id, ssid to represent a subsession id, and tok as an optional string that can be input to ClientLogin by the client. To draw an analogy to a typical real-world setting of a login form between a user and a web service, we can think of uid as the unique username used for login by the client, sid a unique identifier to represent the name of the web service, ssid as the session identifier produced by the server-established secure channel, and tok an optional informational string (e.g. session cookie) that the client can supply to the server on a successful login attempt. We adopt the convention within the UC framework that the sid parameter does not appear in the scheme definition, but is present in our real world experiment.

We use $\mathcal{X}$ to denote the credential domain, $\mathcal{I}_1$ to denote the domain of service identifiers (corresponding to sid), $\mathcal{I}_2$ to denote the domain of session identifiers (corresponding to ssid).

Correctness. We say that Π is *correct* if for any two credentials $\mathsf{x}, \mathsf{y} \in \mathcal{X}$, identifiers $\mathsf{uid} \in \mathcal{U}$, $\mathsf{ssid} \in \mathcal{I}_2$ and token $\mathsf{tok} \in \{0,1\}^\lambda$, for $(\mathsf{pp}, \mathsf{sk}) \leftarrow \mathsf{Setup}(1^\lambda)$, if $\mathcal{C}(\mathsf{x}, \mathsf{y}) = 1$, then $\mathsf{ServerLogin}(\mathsf{pp}, \mathsf{ssid}, \mathsf{Register}(\mathsf{sk}, \mathsf{uid}, \mathsf{y}), \mathsf{ClientLogin}(\mathsf{pp}, \mathsf{ssid}, \mathsf{tok}, \mathsf{uid}, \mathsf{x})) = \mathsf{tok}$ with overwhelming probability.

Security within the UC framework. In the following subsections, we define security within the UC framework in the context of a "real world" and an "ideal world". Intuitively, security will hold if for all environments $\mathcal{E}$ and adversaries $\mathcal{A}$, there exists an efficient simulator $\mathcal{A}^*$ for which the interactions in the real and ideal worlds are indistinguishable to $\mathcal{E}$.

2.1 Real World

We describe the "real" experiment between an adversary $\mathcal{A}$ and challenger $\mathcal{B}$ as follows. In the real experiment $\mathsf{Expt}_{\mathsf{real}}(\mathcal{A})$, the adversary has access to a series of oracles that $\mathcal{B}$ responds to. Let $\mathcal{H} : \{0,1\}^* \rightarrow \{0,1\}^*$ be a hash function which we will model after a random oracle. $\mathcal{B}$ maintains a collection of databases (initialized to be empty) as follows:

- A database of registration keys with setup types $\mathcal{D}_{\mathsf{key}} \leftarrow \{\}$ which is used to keep track of setup keypairs $(\mathsf{pp}, \mathsf{sk})$ along with their setup type, being either HideRegKey or RevealRegKey,
- A database of registered credentials $\mathcal{D}_{\mathsf{Reg}} \leftarrow \{\}$ which is used to keep track of the credentials produced by queries to a registration oracle, and
- A map of credential verifier query restriction statuses $\mathcal{D}_{\mathsf{status}} : \{\} \rightarrow \{$Registered, Restricted, Stolen$\}$ which is kept in tandem with $\mathcal{D}_{\mathsf{Reg}}$ and used to keep track of the status of the credential verifiers.

The challenger responds to oracle queries made by the adversary $\mathcal{A}$ as follows. The reader will notice some artificial restrictions we place on the protocol; these are necessary to ensure that security is not trivially broken or our construction can provably ensure security. We explain these restrictions, after defining these oracles:

- **Setup Oracle:** On an input $(\textsc{Setup}, \mathsf{sid}, \mathsf{SetupType})$ from S, with $\mathsf{SetupType} \in \{\textsc{HideRegKey}, \textsc{RevealRegKey}\}$, if there already exists an entry of the form $(\mathsf{sid}, \star, \star, \star) \in \mathcal{D}_{\mathsf{key}}$, then $\mathcal{B}$ outputs $\perp$. Otherwise, $\mathcal{B}$ computes $(\mathsf{pp}, \mathsf{sk}) \leftarrow \mathsf{Setup}(1^\lambda)$. If $\mathsf{SetupType} = \textsc{HideRegKey}$, then $\mathcal{B}$ only returns pp to $\mathcal{A}$; otherwise, it returns $(\mathsf{pp}, \mathsf{sk})$ to $\mathcal{A}$. Finally, $\mathcal{B}$ adds the entry $(\mathsf{sid}, \mathsf{pp}, \mathsf{sk}, \mathsf{SetupType})$ to $\mathcal{D}_{\mathsf{key}}$.

- **Registration Oracle:** On an input $(\textsc{StoreCredFile}, \mathsf{sid}, \mathsf{uid}, \mathsf{y})$ from S, $\mathcal{B}$ computes $\gamma = \mathsf{Register}(\mathsf{sk}, \mathsf{uid}, \mathsf{y})$, adds the entry $(\mathsf{sid}, \mathsf{uid}, \mathsf{y}, \gamma)$ to $\mathcal{D}_{\mathsf{Reg}}$, and sets $\mathcal{D}_{\mathsf{status}}[\mathsf{sid}, \mathsf{uid}] = \textsc{Registered}$.

- **Steal Credential Verifier Oracle:** On an input $(\mathsf{sid}, \mathsf{uid})$ from $\mathcal{A}$, if for some verifier γ there is an entry of the form $(\mathsf{sid}, \mathsf{uid}, \star, \gamma)$ in $\mathcal{D}_{\mathsf{Reg}}$ and $\mathcal{D}_{\mathsf{status}}[\mathsf{sid}, \mathsf{uid}] = \textsc{Registered}$, then $\mathcal{B}$ returns γ to $\mathcal{A}$ and sets $\mathcal{D}_{\mathsf{status}}[\mathsf{sid}, \mathsf{uid}] = \textsc{Stolen}$. If no such entry in $\mathcal{D}_{\mathsf{Reg}}$ or $\mathcal{D}_{\mathsf{status}}$ exists, then $\mathcal{B}$ returns $\perp$ to $\mathcal{A}$.

- **Client Login Oracle:** On an input $(\textsc{ClientLogin}, \mathsf{sid}, \mathsf{ssid}, \mathsf{tok}, \mathsf{uid}, \mathsf{x})$ from U, if there is no entry of the form $(\mathsf{sid}, \star, \star, \textsc{HideRegKey})$ in $\mathcal{D}_{\mathsf{key}}$, then $\mathcal{B}$ returns $\perp$. Similarly, if $\mathcal{D}_{\mathsf{status}}[\mathsf{sid}, \mathsf{uid}] = \textsc{Stolen}$ and there exists an entry $(\mathsf{sid}, \mathsf{uid}, \mathsf{y}, \star)$ in $\mathcal{D}_{\mathsf{Reg}}$ such that $\mathcal{C}(\mathsf{x}, \mathsf{y}) = 1$, then $\mathcal{B}$ also returns $\perp$. Otherwise, let $(\mathsf{pp}, \mathsf{sk})$ be the keypair for which $(\mathsf{sid}, \mathsf{pp}, \mathsf{sk}, \textsc{HideRegKey})$ exists in $\mathcal{D}_{\mathsf{key}}$. $\mathcal{B}$ sets $\alpha \leftarrow \mathsf{ClientLogin}(\mathsf{pp}, \mathsf{ssid}, \mathsf{tok}, \mathsf{uid}, \mathsf{x})$ and returns $(\mathsf{sid}, \mathsf{ssid}, \mathsf{uid}.\alpha)$ to $\mathcal{A}$. Finally, if either there is no entry of the form $(\mathsf{sid}, \mathsf{uid}, \star, \star)$ in $\mathcal{D}_{\mathsf{Reg}}$, or there exists an entry of the form $(\mathsf{sid}, \mathsf{uid}, \mathsf{y}, \star)$ in $\mathcal{D}_{\mathsf{Reg}}$ for which $\mathcal{C}(\mathsf{x}, \mathsf{y}) = 1$ and $\mathcal{D}_{\mathsf{status}}[\mathsf{sid}, \mathsf{uid}] = \textsc{Registered}$, then $\mathcal{B}$ sets $\mathcal{D}_{\mathsf{status}}[\mathsf{sid}, \mathsf{uid}] = \textsc{Restricted}$.

- **Server Login Oracle:** On an input $(\textsc{ServerLogin}, \mathsf{sid}, \mathsf{ssid}, \mathsf{uid})$ from S, wait and receive $(\mathsf{sid}, \mathsf{ssid}, \mathsf{uid}, \alpha)$ from $\mathcal{A}$. Then $\mathcal{B}$ performs a lookup in $\mathcal{D}_{\mathsf{Reg}}$ to find an entry of the form $(\mathsf{sid}, \mathsf{uid}, \star, \gamma)$ for some verifier γ, and another lookup in $\mathcal{D}_{\mathsf{key}}$ to find an entry of the form $(\mathsf{sid}, \mathsf{pp}, \star)$. If no such entry exists in either lookup, $\mathcal{B}$ sets $\mathsf{tok} \leftarrow \perp$. Otherwise, $\mathcal{B}$ sets $\mathsf{tok} \leftarrow \mathsf{ServerLogin}(\mathsf{pp}, \mathsf{ssid}, \gamma, \alpha)$. If $\mathsf{tok} \neq \perp$, then $\mathcal{B}$ sets $\mathsf{flag} \leftarrow \textsc{Success}$ (otherwise, $\mathsf{flag} \leftarrow \textsc{Failure}$). Finally, $\mathcal{B}$ returns the tuple $(\mathsf{sid}, \mathsf{ssid}, \mathsf{uid}, \mathsf{tok}, \mathsf{flag})$ to S.

- **Random Oracle:** On an input $x \in \mathcal{X}$ from $\mathcal{A}$, $\mathcal{B}$ responds with $\mathcal{H}(x)$.

Enforcing restrictions. We provide an intuitive justification for the restrictions in each of the oracle query responses above. First, consider an adversary which obtains the setup secret at the beginning of the experiment. To such an adversary, the client login messages can no longer retain semantic security. Hence, we disallow client login messages to be produced in this case, ensured by referencing the $\mathcal{D}_{\mathsf{key}}$ database. However, even in this case, we still require that the adversary cannot successfully login without providing the correct credentials; this is ensured by keeping the server login oracle oblivious of whether the setup secret was stolen.

Also, consider an adversary which obtains a credential verifier corresponding to uid. To such an adversary, a correctly formed client login message from uid will no longer be semantically secure. Hence, we disallow the Client Login oracle to produce a message based on a correct credential in this case. For the same reason, we also restrict stealing of the credential verifier for a uid for which the adversary obtained a client login message based on correct credentials. All of this extra bookkeeping is ensured by the $\mathcal{D}_{\mathsf{status}}$ database. On the other hand, we still require that client messages with incorrect credentials must be simulatable, regardless of whether the credential verifier was stolen.

2.2 Ideal World

For an adversary $\mathcal{A}$ and simulator $\mathcal{A}^*$, we describe the experiment $\mathsf{Expt}_{\mathsf{ideal}}(\mathcal{A}, \mathcal{A}^*)$ with respect to the functionality $\mathcal{F}_{\mathsf{CHL}}$ which interacts with both the environment $\mathcal{E}$ (which triggers the parties U and S) and simulator $\mathcal{A}^*$ in the ideal world. The signature for each of the functions is described below:

- The queries of type Setup, ClientLogin, and ServerLogin take input from $\mathcal{E}$ and output to $\mathcal{A}^*$.
- The queries of type StoreCredFile take input from $\mathcal{E}$ and outputs to $\mathcal{E}$.
- The queries of type StealCredFile, OfflineTestCred, TestCred, and Login take input from $\mathcal{A}^*$ and output to $\mathcal{A}^*$.

For each of the oracle query types made by either U or S in $\mathsf{Expt}_{\mathsf{real}}$, there is a corresponding function defined in $\mathcal{F}_{\mathsf{CHL}}$ which takes input from $\mathcal{E}$ in $\mathsf{Expt}_{\mathsf{ideal}}$: Setup invoked by Setup, StoreCredFile invoked by Registration, StealCredFile invoked by Steal Credential Verifier, ClientLogin invoked by Client Login, ServerLogin invoked by Server Login. The Random Oracle is an exception to this case, since it has no interaction with $\mathcal{F}_{\mathsf{CHL}}$ and is simulated entirely by the simulator $\mathcal{A}^*$. For these functions for which $\mathcal{F}_{\mathsf{CHL}}$ returns its response to $\mathcal{A}^*$, it is then up to the design of $\mathcal{A}^*$ to return the corresponding oracle query response back to $\mathcal{E}$.

The ideal functionality $\mathcal{F}_{\mathsf{CHL}}$ keeps an internal state consisting of records of the following types:

- **File**: $\langle \text{File}, \mathsf{sid}, \mathsf{uid}, \mathsf{x} \rangle$ which is marked as being Uncompromised or Compromised.
- **Offline**: $\langle \text{Offline}, \mathsf{sid}, \mathsf{x} \rangle$, for each offline credential guess.
- **Session**: $\langle \mathsf{sid}, \mathsf{ssid}, \mathsf{uid}, \mathsf{x}, \mathsf{type}, \mathsf{tok} \rangle$, where $\mathsf{type} \in \{\text{Correct}, \text{Incorrect}, \text{Invalid}\}$, and for which the record is marked with a state that is either: Fresh, Compromised, Interrupted, or Completed.
- **Server**: $\langle \text{Server}, \mathsf{sid}, \mathsf{ssid}, \mathsf{uid}, \mathsf{y} \rangle$ that is marked as either Fresh or Completed.

The complete definition of $\mathcal{F}_{\mathsf{CHL}}$ is encapsulated in Figure 1. We use U to represent the user and S to represent the server, as the participating parties controlled by the environment in the protocol, and $\mathcal{A}^*$ to represent the simulator in the ideal world.

2.3 Comparison to UC aPAKE Functionality

Although the $\mathcal{F}_{\mathsf{CHL}}$ functionality shares many commonalities with UC aPAKEs, it also has significant departures, as we discuss below. We recall the UC aPAKE functionality definition in Appendix D. In the following, we describe the relevant features of aPAKEs in-line, and we encourage the reader to refer to the definitions also presented in [GMR06, JKX18] as a reference, for completeness.

Setup and Registration

- On $(\textsc{Setup}, \mathsf{sid}, \mathsf{SetupType})$ from S, if this is the first $\textsc{Setup}$ message for sid, check if $\mathsf{SetupType} \in \{\textsc{HideRegKey}, \textsc{RevealRegKey}\}$, and then forward $(\mathsf{Setup}, \mathsf{sid}, \mathsf{SetupType})$ to $\mathcal{A}^*$.
- On $(\textsc{StoreCredFile}, \mathsf{sid}, \mathsf{uid}, \mathsf{y})$ from S, if this is the first $\textsc{StoreCredFile}$ message for $(\mathsf{sid}, \mathsf{uid})$, record $\langle \textsc{File}, \mathsf{sid}, \mathsf{uid}, \mathsf{y} \rangle$ and mark it $\textsc{Uncompromised}$.

Stealing Credential Data

- On $(\textsc{StealCredFile}, \mathsf{sid}, \mathsf{uid})$ from $\mathcal{A}^*$, if there is no record $\langle \textsc{File}, \mathsf{sid}, \mathsf{uid}, \mathsf{y} \rangle$, return "$\textsc{No Cred File}$" to $\mathcal{A}^*$. Otherwise, if the record is marked $\textsc{Uncompromised}$, mark it $\textsc{Compromised}$.
  - If there is a record $\langle \textsc{Offline}, \mathsf{sid}, \mathsf{x} \rangle$, send x to $\mathcal{A}^*$.
  - Else, return "$\textsc{Cred File Stolen}$" to $\mathcal{A}^*$.
- On $(\textsc{OfflineTestCred}, \mathsf{sid}, \mathsf{uid}, \mathsf{x})$ from $\mathcal{A}^*$, do:
  - If there is a record $\langle \textsc{File}, \mathsf{sid}, \mathsf{uid}, \mathsf{y} \rangle$ marked $\textsc{Compromised}$, do: if $\mathcal{C}(\mathsf{x}, \mathsf{y}) = 1$, return "$\textsc{Correct Guess}$" to $\mathcal{A}^*$; else return "$\textsc{Incorrect Guess}$".
  - Else if $\mathcal{C}(\mathsf{x}, \mathsf{y}) = 1$, then record $\langle \textsc{Offline}, \mathsf{sid}, \mathsf{x} \rangle$.

Credential Login

- On $(\textsc{ClientLogin}, \mathsf{sid}, \mathsf{ssid}, \mathsf{uid}, \mathsf{x}, \mathsf{tok})$ from U, if this the first $\textsc{ClientLogin}$ message for $(\mathsf{sid}, \mathsf{ssid}, \mathsf{uid})$, try to retrieve $\langle \textsc{File}, \mathsf{sid}, \mathsf{uid}, \mathsf{y} \rangle$ and do:
  - if the retrieval did not work, set $\mathsf{type} = \textsc{Invalid}$.
  - if $\mathcal{C}(\mathsf{x}, \mathsf{y}) = 1$, set $\mathsf{type} = \textsc{Correct}$.
  - else set $\mathsf{type} = \textsc{Incorrect}$.

  Record $\langle \mathsf{sid}, \mathsf{ssid}, \mathsf{uid}, \mathsf{x}, \mathsf{type}, \mathsf{tok} \rangle$ and mark it $\textsc{Fresh}$. Send $(\mathsf{sid}, \mathsf{ssid}, \mathsf{uid}, \mathsf{type})$ to $\mathcal{A}^*$.
- On $(\textsc{ServerLogin}, \mathsf{sid}, \mathsf{ssid}, \mathsf{uid})$ from S, if this is the first $\textsc{ServerLogin}$ message for $(\mathsf{sid}, \mathsf{ssid}, \mathsf{uid})$, retrieve $\langle \textsc{File}, \mathsf{sid}, \mathsf{uid}, \mathsf{y} \rangle$ and send $(\mathsf{sid}, \mathsf{ssid}, \mathsf{uid})$ to $\mathcal{A}^*$. Record $\langle \textsc{Server}, \mathsf{sid}, \mathsf{ssid}, \mathsf{uid}, \mathsf{y} \rangle$ and mark it $\textsc{Fresh}$.

Active Session Attacks

- On $(\textsc{TestCred}, \mathsf{sid}, \mathsf{ssid}, \mathsf{uid}, \mathsf{x})$ from $\mathcal{A}^*$, if there is a record $\langle \textsc{Server}, \mathsf{sid}, \mathsf{ssid}, \mathsf{uid}, \mathsf{y} \rangle$ marked $\textsc{Fresh}$, do: If $\mathcal{C}(\mathsf{x}, \mathsf{y}) = 1$, return "$\textsc{Correct Guess}$" to $\mathcal{A}^*$ and set $\mathsf{state} = \textsc{Compromised}$; else return "$\textsc{Incorrect Guess}$" and set $\mathsf{state} = \textsc{Interrupted}$. Mark $\langle \mathsf{sid}, \mathsf{ssid}, \mathsf{uid}, \mathsf{x}, \mathsf{type}, \mathsf{tok} \rangle$ with state.

Login and Authentication

- On $(\textsc{Login}, \mathsf{sid}, \mathsf{ssid}, \mathsf{uid}, \mathsf{tok}', \mathsf{flag})$ from $\mathcal{A}^*$, if there is a record $\langle \textsc{Server}, \mathsf{sid}, \mathsf{ssid}, \mathsf{uid}, \mathsf{y} \rangle$ marked $\textsc{Fresh}$, then mark it $\textsc{Completed}$:
  - If there is a record $\langle \mathsf{sid}, \mathsf{ssid}, \mathsf{uid}, \mathsf{x}, \mathsf{type}, \mathsf{tok} \rangle$ marked $\textsc{Fresh}$, then mark it $\textsc{Completed}$. Set $\mathsf{flag} = \textsc{Success}$ if $\mathsf{type} = \textsc{Correct}$ and $\mathsf{flag} = \textsc{Failure}$, otherwise.
  - If there is a record $\langle \mathsf{sid}, \mathsf{ssid}, \mathsf{uid}, \mathsf{x}, \mathsf{type}, \mathsf{tok} \rangle$ marked $\textsc{Interrupted}$, then mark it $\textsc{Completed}$ and set $(\mathsf{tok}, \mathsf{flag}) = (\perp, \textsc{Failure})$.
  - If there is a record $\langle \mathsf{sid}, \mathsf{ssid}, \mathsf{uid}, \mathsf{x}, \mathsf{type}, \mathsf{tok} \rangle$ marked $\textsc{Compromised}$, then mark it $\textsc{Completed}$ and set $\mathsf{tok} = \mathsf{tok}'$.

  Output $(\mathsf{sid}, \mathsf{ssid}, \mathsf{uid}, \mathsf{tok}, \mathsf{flag})$ to S.

Figure 1: The ideal functionality $\mathcal{F}_{\mathsf{CHL}}$.

Generalization to credentials. The $\mathcal{F}_{\mathsf{CHL}}$ functionality allows more general forms of authorization than just password equality verification.

Setup and registration. Unlike aPAKEs, $\mathcal{F}_{\mathsf{CHL}}$ allows for two levels of server secrets: one is a universal setup secret which applies to all parties and sessions within the scope of the sid, and the other is a set of credential verifiers (password files), each corresponding to a different user identifier uid. In practice, the setup secret is only kept with a registration server and the verifiers are kept with the login server.

In aPAKEs, the server can generate a credential file itself, given the user credential, but in $\mathcal{F}_{\mathsf{CHL}}$, only the registration server can generate the verifier file—the login server cannot do this on its own. On the other hand, the login process does not involve the registration server.

As in aPAKEs, we consider security properties even when all the server secrets are revealed to the adversary, including the setup secret. However, our constructions require these to be handled differently from when only the login server secrets are stolen. In particular, the setup secret revelation can only be performed at the beginning, whereas the verifier files can be stolen adaptively at any point.

Stealing credential data. This functionality remains exactly same as in aPAKEs, since we model the same behavior that an adversary can perform in order to obtain the credential file for registered users.

Credential login. The ClientLogin query also takes a tok field in addition to the credential. The reason for this extra field is fairly technical, but we provide an intuitive perspective below. The token enables the environment to check whether there is a match between the client session for a given ssid and the server session for the same ssid. In aPAKEs, this is naturally observable by the environment, since in a successful session, the agreed keys would come out to be the same in both the matching sessions. The key is also independently random from all other components of the protocol in the functionality, and so the simulator does not have the ability to obtain it, in case the honest messages were not tampered with. However, in the case of $\mathcal{F}_{\mathsf{CHL}}$, there is no key agreed on; only the output representing login success or failure. This could allow for simulator strategies for insecure protocols, where the simulator can enforce a success without even extracting a credential and calling the TestCred functionality, even when the adversary actively modifies messages. By allowing a token to come from the environment, and requiring a simulator to work for all possible environments, we prevent these kinds of simulator strategies that could work for inherently insecure protocols.

Although the functionality does not transmit the credential to the simulator, it does indicate three types: Correct to signify that the proper credential is being used; Incorrect to indicate that an incorrect credential is being used; and Invalid to indicate that there is no registered credential. Even though this is more information than aPAKEs allow, it still does not harm credential privacy. We need this extra information to allow our constructions to have a simulation strategy for each type. Note also that, unlike aPAKEs, we explicitly handle the case where a client login is initiated, even if there is no corresponding registration. This is to address credential privacy in the practical situations where a client inadvertently logs in to this server mistaking it for some other server.

The Server Login portion remains the same as aPAKEs.

Active session attacks. The TestCred functionality is essentially same as in aPAKEs, except for the generalization to circuits, rather than just equality checks. In aPAKEs, there is also an Impersonate functionality to allow the simulator to compromise a client session if the corresponding credential file was stolen. We do not require this functionality in $\mathcal{F}_{\mathsf{CHL}}$ because the client session does not output anything to the environment.

Login and authentication. This functionality demonstrates the difference in objectives between aPAKEs and $\mathcal{F}_{\mathsf{CHL}}$. While in aPAKEs, the objective is to output a mutually agreed upon random key, here the login part just outputs the success or failure of the protocol. As discussed before, the token field is also output to the environment. In aPAKEs, there is a TestAbort functionality to allow aborting if certain verifications fail. This functionality is redundant in $\mathcal{F}_{\mathsf{CHL}}$, since login failure can already account for such failed verifications.

Corruptions. While in aPAKEs, the only public parameters come from a trusted third party, in $\mathcal{F}_{\mathsf{CHL}}$, the public parameters come from the registration server which also holds a secret corresponding to this parameter. While we do not allow the registration server to generate these parameters maliciously, we do allow the adversary the choice to obtain the secret at the very beginning.

This distinction has some important consequences. Although trusting the registration server is not ideal, we argue that it is still limited in scope. Firstly, the public parameter generation is only done once in the scope of an sid. Secondly, even if the registration server secret is revealed, an attacker still has to do an offline attack, not only to recover a credential, but also to successfully login. Thirdly, the registration server does not get involved at all at the far more frequent login sessions - it's only to used to register credentials and transmit a verifier file to the login server.

We also observe that this level of trust seems to be intrinsic to our setting. If the registration server held no setup secret, then that would allow the login server to be able to generate verifier from the public parameters themselves using it's credential guesses. This would invalidate semantic security for client messages containing inadvertent credentials, an important practical scenario that we did aim to address. On the other hand, our trust assumption for the login server remains the same as in aPAKEs. We allow the adversary to adaptively steal credential verifiers from the login server.

2.4 Authenticated Login

The primary objective of two parties participating in a key exchange protocol is to establish a secure channel. It was shown in [CK02] that the UC notion of secure channel is realizable using a UC key exchange protocol as a subroutine, or more technically in the UC key exchange hybrid model.

To relate our notion of CHL to these existing notions, we define a simpler abstraction of the CHL ideal functionality, called "authenticated login", which abstracts away passwords/credentials in the same sense as standard key exchange is to password-based key exchange. We show that the UC notion of secure channels can also be obtained in the authenticated login hybrid model.

However, the typical use case we consider within TLS does not require such a bootstrapping to secure channels. Instead, we use the CHL protocol to establish the authenticity of the user and continue the subsequent interactions based on the already established server-authenticated TLS channel, which corresponds to the CHL-over-TLS approach described in Section 1. Although this

suffices for our applications, we note that this is still weaker than a standard secure channel as it does impose on the user application to not send any sensitive content which assumes mutual trust before the CHL flow is completed.

Note that the compilation of an authenticated login scheme into a secure channel requires stronger security properties from the wrapper protocol than those needed for the corresponding wrapper protocol needed to wrap key exchange into a secure channel. While a key exchange protocol ensures that the exchanged key has strong entropy and computational independence properties, an authenticated login protocol instead receives the analogue of this key (the token tok) from the environment, and hence does not by itself enjoy such entropy properties. In the presence of stronger wrapper protocols which are already practically deployed, such as TLS with server-side PKI authentication, one can indeed relax such requirements on the core credential verification system itself. Weakening the requirements enables the construction of efficient schemes which are also amenable to modular deployment in common scenarios existing today.

The formal descriptions and proofs are given in Appendix C. The description of the authenticated login functionality $\mathcal{F}_{\mathsf{AL}}$ is given in Figure 2. We provide a simple PKI-based protocol in Figure 3 and prove that it securely realizes the $\mathcal{F}_{\mathsf{AL}}$ functionality. We then show that the UC secure channel functionality $\mathcal{F}_{\mathsf{SC}}$, described in Figure 4 can be securely realized in the $\mathcal{F}_{\mathsf{AL}}$-hybrid model, by a protocol described in Figure 5.

3 Construction from Identity-Based Encryption

In this section, we describe a construction of Credential-Hiding Login (CHL) using an anonymous identity-based encryption (IBE) scheme. We will implicitly use the IND-anon-ID-CCA property of the IBE scheme to satisfy our notion of credential privacy for a CHL scheme. Then, coupled with the use of random oracles, we show that this instantiation satisfies UC security according to the definitions from Section 2.

Identity-based encryption. We first recall the definitions of an IBE scheme $\mathsf{IBE} = (\mathsf{IBE.Setup}, \mathsf{IBE.KeyGen}, \mathsf{IBE.Encrypt}, \mathsf{IBE.Decrypt})$, consisting of four algorithms as follows. The setup algorithm $\mathsf{IBE.Setup}(1^\lambda) \to (\mathsf{pp}, \mathsf{msk})$ outputs a pair of public parameters pp and a master secret key msk. The key generation algorithm $\mathsf{IBE.KeyGen}(\mathsf{msk}, \mathsf{id}) \to \mathsf{sk}_{\mathsf{id}}$ takes the master secret key msk and an identity $\mathsf{id} \in \{0,1\}^*$ and outputs a secret key for the identity $\mathsf{sk}_{\mathsf{id}}$. The encrypt algorithm $\mathsf{IBE.Encrypt}(\mathsf{pp}, \mathsf{id}, \mathsf{m}) \to \mathsf{c}$ takes as input the public parameters pp, an identity $\mathsf{id} \in \{0,1\}^*$, a message $\mathsf{m} \in \{0,1\}^*$, and outputs a ciphertext $\mathsf{c} \in \{0,1\}^*$. The decrypt algorithm $\mathsf{IBE.Decrypt}(\mathsf{sk}_{\mathsf{id}}, \mathsf{c}) \to \{\mathsf{m}, \perp\}$ takes as input a secret key $\mathsf{sk}_{\mathsf{id}}$ and a ciphertext c, and outputs either m if decryption was successful, or $\perp$ otherwise. The correctness guarantee of an IBE scheme is that for any message $\mathsf{m} \in \{0,1\}^*$ and identity $\mathsf{id} \in \{0,1\}^*$, with $(\mathsf{pp}, \mathsf{msk}) \leftarrow \mathsf{IBE.Setup}(1^\lambda)$ and $\mathsf{sk}_{\mathsf{id}} \leftarrow \mathsf{IBE.KeyGen}(\mathsf{msk}, \mathsf{id})$, $\mathsf{IBE.Decrypt}(\mathsf{sk}_{\mathsf{id}}, \mathsf{IBE.Encrypt}(\mathsf{pp}, \mathsf{id}, \mathsf{m})) = \mathsf{m}$ with overwhelming probability.

We define the experiment $\mathsf{Expt}_b^{\mathsf{anon\text{-}cca}}(\mathcal{A})$ as follows:

Definition 1 ($\mathsf{Expt}_b^{\mathsf{anon\text{-}cca}}(\mathcal{A})$)**.** For an adversary $\mathcal{A}$ and bit $b \in \{0,1\}$, we define $\mathsf{Expt}_b^{\mathsf{anon\text{-}cca}}(\mathcal{A})$ as follows. First, the challenger computes $(\mathsf{pp}, \mathsf{msk}) \leftarrow \mathsf{IBE.Setup}(1^\lambda)$ and sends pp to $\mathcal{A}$. The adversary has access to a key generation oracle which, on input an identity id, the oracle returns $\mathsf{sk}_{\mathsf{id}} \leftarrow \mathsf{IBE.KeyGen}(\mathsf{msk}, \mathsf{id})$. The adversary also has access to a challenge oracle, which on input

two pairs of inputs $(\mathsf{id}_1, \mathsf{m}_1)$, $(\mathsf{id}_2, \mathsf{m}_2)$, the challenger returns $\mathsf{IBE.Encrypt}(\mathsf{pp}, \mathsf{id}_b, \mathsf{m}_b)$ to $\mathcal{A}$. The adversary also has access to a decryption oracle, which on input a secret key $\mathsf{sk}_{\mathsf{id}}$ and ciphertext c, returns $\mathsf{IBE.Decrypt}(\mathsf{sk}_{\mathsf{id}}, c)$. If at any point in time, there is a key generation oracle query made for an identity that was also submitted to the challenge oracle, or if the decryption oracle is queried on $(\mathsf{sk}_{\mathsf{id}_1}, \mathsf{m}_1)$ or $(\mathsf{sk}_{\mathsf{id}_2}, \mathsf{m}_2)$, then the experiment outputs $\perp$. At the end of the experiment, $\mathcal{A}$ outputs a bit, which the experiment also outputs.

We say that an IBE scheme satisfies IND-anon-ID-CCA security if the quantity

$$|\Pr[\mathsf{Expt}_b^{\mathsf{anon\text{-}cca}}(\mathcal{A}) = 1] - \Pr[\mathsf{Expt}_1^{\mathsf{anon\text{-}cca}}(\mathcal{A}) = 1]|$$

is negligible.

The construction Π_{pwd}. Let $\mathcal{X}$ be a domain of credentials, $\mathcal{U}$ be a domain of user identifiers, and $\mathcal{H} : \mathcal{U} \times \mathcal{X} \to (\{0,1\}^\lambda, \{0,1\}^\lambda)$ and $\mathsf{f} : \{0,1\}^\lambda \to \{0,1\}^\lambda$ be hash functions which we will model after random oracles. We describe the construction Π_{pwd} as follows:

- $\mathsf{Setup}(1^\lambda) \to (\mathsf{pp}, \mathsf{sk})$. The Setup algorithm outputs $(\mathsf{pp}, \mathsf{sk}) \leftarrow \mathsf{IBE.Setup}(1^\lambda)$.
- $\mathsf{Register}(\mathsf{sk}, \mathsf{uid}, \mathsf{y}) \to \gamma$. The Register algorithm takes as input a credential $\mathsf{y} \in \mathcal{X}$, sets $(\mathsf{s}, \mathsf{t}) = \mathcal{H}(\mathsf{uid}, \mathsf{y})$, and outputs $(\mathsf{IBE.KeyGen}(\mathsf{sk}, \mathsf{s}), \mathsf{f}(\mathsf{t}))$.
- $\mathsf{ClientLogin}(\mathsf{pp}, \mathsf{ssid}, \mathsf{tok}, \mathsf{uid}, \mathsf{x}) \to \alpha$. The client login algorithm takes as input a set of public parameters pp, ssid ssid, token tok, user id $\mathsf{uid} \in \mathcal{U}$, a credential $\mathsf{x} \in \mathcal{X}$, computes $(\mathsf{s}, \mathsf{t}) = \mathcal{H}(\mathsf{uid}, \mathsf{x})$, and outputs a message

$$\alpha \coloneqq \mathsf{IBE.Encrypt}(\mathsf{pp}, \mathsf{id} = \mathsf{s}, (\mathsf{t}, \mathsf{ssid}, \mathsf{tok})).$$

- $\mathsf{ServerLogin}(\mathsf{pp}, \mathsf{ssid}, \mathsf{uid}, \gamma, \alpha) \to \{\mathsf{tok}, \perp\}$. The server login algorithm takes in the public parameters pp, ssid ssid, user id $\mathsf{uid} \in \mathcal{U}$, a credential verifier $\gamma = (\gamma_1, \gamma_2)$, the client message α, and computes $(\mathsf{t}', \mathsf{ssid}', \mathsf{tok}') \leftarrow \mathsf{IBE.Decrypt}(\gamma_1, \alpha)$. If $\mathsf{t}' = \perp$ or $\mathsf{f}(\mathsf{t}') \neq \gamma_2$ or $\mathsf{ssid} \neq \mathsf{ssid}'$, then return $\perp$. Otherwise, return tok'.

Correctness. For any identifier $\mathsf{uid} \in \mathcal{U}$ and any two credentials $\mathsf{x}, \mathsf{y} \in \mathcal{X}$, for $(\mathsf{pp}, \mathsf{sk}) \leftarrow \mathsf{IBE.Setup}(1^\lambda)$, for $(\mathsf{s}_\mathsf{x}, \mathsf{t}_\mathsf{x}) = \mathcal{H}(\mathsf{uid}, \mathsf{x})$ and $(\mathsf{s}_\mathsf{y}, \mathsf{t}_\mathsf{y}) = \mathcal{H}(\mathsf{uid}, \mathsf{y})$, let the verifier be set as $(\gamma_1, \gamma_2) = (\mathsf{IBE.KeyGen}(\mathsf{sk}, \mathsf{s}_\mathsf{y}), \mathsf{f}(\mathsf{t}_\mathsf{y})) \leftarrow \mathsf{Register}(\mathsf{sk}, \mathsf{uid}, \mathsf{y})$. Let $\alpha = \mathsf{IBE.Encrypt}(\mathsf{pp}, \mathsf{id} = \mathsf{s}_\mathsf{x}, (\mathsf{t}_\mathsf{x}, \mathsf{ssid}, \mathsf{tok}))$. If $\mathsf{x} = \mathsf{y}$, then by the correctness of the IBE scheme, we have that $\mathsf{ServerLogin}(\mathsf{pp}, (\gamma_1, \gamma_2), \alpha) = \mathsf{tok}$.

3.1 Description of the Simulator

The simulator $\mathcal{A}^*$ will maintain the following databases across oracle queries:

- A database of registration keys with setup types $\mathcal{D}_{\mathsf{key}} \leftarrow \{\}$ which is used to keep track of setup keypairs $(\mathsf{pp}, \mathsf{sk})$ along with their setup type, being either HideRegKey or RevealRegKey,
- A map of credential file statuses $\mathcal{D}_{\mathsf{status}} : \mathcal{I}_1 \times \mathcal{U} \to \{\text{Registered}, \text{Restricted}, \text{Stolen}\}$,
- A map $\mathcal{D}_{\mathsf{Reg}} : \mathcal{I}_1 \times \mathcal{U} \to \mathcal{X}$ of known registered passwords collected by the simulator,

- A database of client login requests $\mathcal{D}_{\mathsf{sent}} \leftarrow \{\}$ which is used to keep track of entries submitted to the Client Login oracle, and
- A database of random oracle queries $\mathcal{D}_{\mathsf{RO}} : \mathcal{I}_1 \times \mathcal{U} \times \mathcal{X} \rightarrow (\{0,1\}^\lambda \times \{0,1\}^\lambda)$.

For each oracle query made by an adversary $\mathcal{A}$, we show how the simulator $\mathcal{A}^*$ responds by taking advantage of the interface provided by the ideal functionality $\mathcal{F}_{\mathsf{CHL}}$.

- **Setup Oracle:** On input $(\mathsf{sid}, \mathsf{SetupType})$ from $\mathcal{F}_{\mathsf{CHL}}$, the simulator samples $(\mathsf{pp}, \mathsf{sk}) \leftarrow \Pi_{\mathsf{pwd}}.\mathsf{Setup}(1^\lambda)$. If $\mathsf{SetupType} = \textsc{HideRegKey}$, then send pp to $\mathcal{A}$ and retain sk. If $\mathsf{SetupType} = \textsc{RevealRegKey}$, then send $(\mathsf{pp}, \mathsf{sk})$ to $\mathcal{A}$. In either case, add the entry $(\mathsf{sid}, \mathsf{pp}, \mathsf{sk}, \mathsf{SetupType})$ to $\mathcal{D}_{\mathsf{key}}$.
- **Steal Credential Verifier Oracle:** On input $(\mathsf{sid}, \mathsf{uid})$ from $\mathcal{A}$, if $\mathcal{D}_{\mathsf{status}}[\mathsf{sid}, \mathsf{uid}] = \textsc{Restricted}$, the simulator returns $\perp$ to $\mathcal{A}$. Otherwise, the simulator $\mathcal{A}^*$ first calls $\mathcal{F}_{\mathsf{CHL}}$ with the query $(\textsc{StealCredFile}, \mathsf{sid}, \mathsf{uid})$. If it receives "No Cred File" as a response, then it also sends $\perp$ to $\mathcal{A}$. Otherwise, there are two cases to consider:
  - **Case 1:** $\mathcal{F}_{\mathsf{CHL}}$ returns "Cred File Stolen". Then, if there is an entry $(\mathsf{sid}, \mathsf{uid})$ in $\mathcal{D}_{\mathsf{Reg}}$, then set $\mathsf{x} \leftarrow \mathcal{D}_{\mathsf{Reg}}[\mathsf{sid}, \mathsf{uid}]$ and set $(\mathsf{s}, \mathsf{t}) \leftarrow \mathcal{D}_{\mathsf{RO}}[\mathsf{sid}, \mathsf{uid}, \mathsf{x}]$. Otherwise, the simulator samples $(\mathsf{s}, \mathsf{t}) \leftarrow_{\mathsf{R}} \{0,1\}^\lambda \times \{0,1\}^\lambda$ uniformly at random and sets $\mathcal{D}_{\mathsf{RO}}[\mathsf{sid}, \mathsf{uid}, \star] \leftarrow (\mathsf{s}, \mathsf{t})$.
  - **Case 2:** $\mathcal{F}_{\mathsf{CHL}}$ returns a credential x. Then, the simulator sets $(\mathsf{s}, \mathsf{t}) \leftarrow \mathcal{D}_{\mathsf{RO}}[\mathsf{sid}, \mathsf{uid}, \mathsf{x}]$.[2]

  In either case, $\mathcal{A}^*$ returns $\gamma \leftarrow (\mathsf{IBE.KeyGen}(\mathsf{sk}, \mathsf{s}), \mathsf{f}(\mathsf{t}))$ to the adversary.
- **Client Login Oracle:** Let $(\textsc{ClientLogin}, \mathsf{sid}, \mathsf{ssid}, \mathsf{uid}, \mathsf{x}^*, \mathsf{tok}^*, \mathsf{type})$ be the query sent to $\mathcal{F}_{\mathsf{CHL}}$. On input $(\mathsf{sid}, \mathsf{ssid}, \mathsf{uid}, \mathsf{type})$ from $\mathcal{F}_{\mathsf{CHL}}$, if there is an entry of the form $(\mathsf{sid}, \star, \star, \textsc{RevealRegKey})$ in $\mathcal{D}_{\mathsf{key}}$, or if there is no entry of the form $(\mathsf{sid}, \star, \star, \star)$ in $\mathcal{D}_{\mathsf{key}}$, or $\mathcal{D}_{\mathsf{status}}[\mathsf{sid}, \mathsf{uid}] =$ "Cred File Stolen" and $\mathsf{type} = \textsc{Correct}$, then $\mathcal{A}^*$ returns $\perp$ to $\mathcal{A}$. Otherwise, let pp be the unique parameter for which $(\mathsf{sid}, \mathsf{pp}, \star, \star)$ exists in $\mathcal{D}_{\mathsf{key}}$. Then, $\mathcal{A}^*$ randomly samples $\mathsf{id} \leftarrow_{\mathsf{R}} \{0,1\}^\lambda$ and $\mathsf{m} \leftarrow_{\mathsf{R}} \{0,1\}^\lambda$, and sends $\alpha \leftarrow \mathsf{IBE.Encrypt}(\mathsf{pp}, \mathsf{id}, \mathsf{m})$ to $\mathcal{A}$. Finally, if $\mathsf{type} \in \{\textsc{Invalid}, \textsc{Correct}\}$, then $\mathcal{A}^*$ sets $\mathcal{D}_{\mathsf{status}}[\mathsf{sid}, \mathsf{uid}] \leftarrow \textsc{Restricted}$ and adds the entry $(\mathsf{sid}, \mathsf{ssid}, \mathsf{uid}, \alpha)$ to $\mathcal{D}_{\mathsf{sent}}$.
- **Server Login Oracle:** On input $(\mathsf{sid}, \mathsf{ssid}, \mathsf{uid}, \alpha)$ from $\mathcal{A}$, let $(\alpha_1, \alpha_2) = \alpha$. There are two cases to consider:
  - **Case 1:** There is already an entry of the form $(\star, \star, \star, \alpha)$ in $\mathcal{D}_{\mathsf{sent}}$. Let $(\mathsf{sid}', \mathsf{ssid}', \mathsf{uid}', \alpha)$ be this entry. If the two tuples $(\mathsf{sid}, \mathsf{ssid}, \mathsf{uid})$ and $(\mathsf{sid}', \mathsf{ssid}', \mathsf{uid}')$ are equal in all positions, then set $\mathsf{tok} \leftarrow \perp$ and $\mathsf{flag} \leftarrow \perp$. Otherwise, set $\mathsf{tok} \leftarrow \perp$, $\mathsf{flag} \leftarrow \textsc{Failure}$, and call $(\textsc{TestCred}, \mathsf{sid}, \mathsf{ssid}, \mathsf{uid}, \perp)$ from $\mathcal{F}_{\mathsf{CHL}}$.
  - **Case 2:** There is no entry of the form $(\star, \star, \star, \alpha)$ in $\mathcal{D}_{\mathsf{sent}}$. There are two subcases to consider:
    - **Case 2a:** There is an entry of the form $(\mathsf{sid}, \mathsf{uid}, \star)$ in $\mathcal{D}_{\mathsf{RO}}$. Let $(\mathsf{s}, \mathsf{t}) \leftarrow \mathcal{D}_{\mathsf{RO}}[\mathsf{sid}, \mathsf{uid}, \star]$, and compute $(\mathsf{v}, \mathsf{ssid}', \mathsf{tok}) \leftarrow \mathsf{IBE.Decrypt}(\mathsf{IBE.KeyGen}(\mathsf{sk}, \mathsf{s}), \alpha)$. If $\mathsf{f}(\mathsf{v}) = \mathsf{f}(\mathsf{t})$, then the simulator aborts.[3] Otherwise, set $\mathsf{tok} \leftarrow \perp$ and $\mathsf{flag} \leftarrow \textsc{Failure}$, and call $\mathcal{F}_{\mathsf{CHL}}$

[2] Note that at this point, the simulator must have already called $\mathcal{F}_{\mathsf{CHL}}$ with an $\textsc{OfflineTestCred}$ query with the correct password.

[3] We will show in the security proof that the probability of this event happening is negligible.

with (TestCred, sid, ssid, uid, $\perp$).

* **Case 2b:** There is no entry of the form $(\mathsf{sid}, \mathsf{uid}, \star)$ in $\mathcal{D}_{\mathsf{RO}}$. Then, perform a linear search for every entry in $\mathcal{D}_{\mathsf{RO}}$ of the form $(\mathsf{sid}, \mathsf{uid}, \mathsf{x}) \rightarrow (\mathsf{s}, \mathsf{t})$, by checking if there exists a tuple $(\mathsf{t}', \mathsf{ssid}', \mathsf{tok})$ such that $\mathsf{IBE.Decrypt}(\mathsf{IBE.KeyGen}(\mathsf{sk}, \mathsf{s}), \alpha) = (\mathsf{t}', \mathsf{ssid}', \mathsf{tok})$, $\mathsf{f}(\mathsf{t}') = \mathsf{f}(\mathsf{t})$, and $\mathsf{ssid} = \mathsf{ssid}'$. If so, then call $\mathcal{F}_{\mathsf{CHL}}$ with (TestCred, sid, ssid, uid, x). If $\mathcal{F}_{\mathsf{CHL}}$ returns "Correct Guess", then set $\mathsf{flag} \leftarrow$ Success and set $\mathcal{D}_{\mathsf{Reg}}[\mathsf{sid}, \mathsf{uid}] \leftarrow \mathsf{x}$; otherwise, set $\mathsf{tok} \leftarrow \perp$ and $\mathsf{flag} \leftarrow$ Failure. If the linear search fails, then set $\mathsf{tok} \leftarrow \perp$, $\mathsf{flag} \leftarrow$ Failure, and call $\mathcal{F}_{\mathsf{CHL}}$ with (TestCred, sid, ssid, uid, $\perp$).

After all cases, call $\mathcal{F}_{\mathsf{CHL}}$ with (Login, sid, ssid, uid, tok, flag).

- **Random Oracle:** On input $(\mathsf{sid}, \mathsf{uid}, \mathsf{x})$ from $\mathcal{A}$, check if this entry is in $\mathcal{D}_{\mathsf{RO}}$, returning $(\mathsf{s}, \mathsf{t}) \leftarrow \mathcal{D}_{\mathsf{RO}}[\mathsf{sid}, \mathsf{uid}, \mathsf{x}]$ if so. Otherwise, call $\mathcal{F}_{\mathsf{CHL}}$ with the query (OfflineTestCred, sid, uid, x). There are two cases to consider:

  – $\mathcal{F}_{\mathsf{CHL}}$ return "Correct Guess". Then first check if there exists a record $(\mathsf{sid}, \mathsf{uid}, \star) \rightarrow (\mathsf{s}, \mathsf{t})$ in $\mathcal{D}_{\mathsf{RO}}$, replacing it with $\mathcal{D}_{\mathsf{RO}}[\mathsf{sid}, \mathsf{uid}, \mathsf{x}] \leftarrow (\mathsf{s}, \mathsf{t})$ if possible. Otherwise, $\mathcal{A}^*$ samples $(\mathsf{s}, \mathsf{t}) \leftarrow_{\mathsf{R}} \{0,1\}^\lambda \times \{0,1\}^\lambda$ uniformly at random, sets $\mathcal{D}_{\mathsf{RO}}[\mathsf{sid}, \mathsf{uid}, \mathsf{x}] \leftarrow (\mathsf{s}, \mathsf{t})$ and sets $\mathcal{D}_{\mathsf{Reg}}[\mathsf{sid}, \mathsf{uid}] \leftarrow \mathsf{x}$.

  – $\mathcal{F}_{\mathsf{CHL}}$ returns "Incorrect Guess". Then $\mathcal{A}^*$ samples $(\mathsf{s}, \mathsf{t}) \leftarrow_{\mathsf{R}} \{0,1\}^\lambda \times \{0,1\}^\lambda$ uniformly at random and sets $\mathcal{D}_{\mathsf{RO}}[\mathsf{sid}, \mathsf{uid}, \mathsf{x}] \leftarrow (\mathsf{s}, \mathsf{t})$.

  After both cases, $\mathcal{A}^*$ finishes by returning (s, t) to $\mathcal{A}$.

3.2 Security

We show that our scheme is secure in the UC model. While we defer the full proof to Appendix A, here we provide an intuitive proof sketch.

Theorem 1. *The construction Π_{pwd} is a secure credential-hiding login protocol, assuming* IBE *is an IND-anon-ID-CCA IBE scheme, and $\mathcal{H}$ and f are random oracles.*

Proof Sketch. We show that a protocol adversary cannot distinguish between the real world challenger and the simulator that just uses the $\mathcal{F}_{\mathsf{CHL}}$ functionality. This is done by transitioning through a series of hybrid games (denoted Game_i for $i \in [1, 4]$), beginning with the simulator $\mathcal{A}^*$ in the ideal world, and ending with the real world challenger.

While $\mathcal{A}^*$ does not see any passwords in the clear, the challenger for Game_1 obtains them from the environment. This enables the Game_1 challenger to resolve the case where the simulator aborts in response to a Server Login Oracle query. Indistinguishability then follows by observing that the probability of the abort event is negligible, assuming f is a random oracle. The random oracle simulation used here is critical, since if f were only collision-resistant and one-way, it is not clear how to provide a consistent simulation.

The challenger for Game_2 directly uses the password information, which the Game_1 challenger was still performing a search on. Indistinguishability follows by using the random oracle properties of $\mathcal{H}$ and the collision resistance property of f.

While the Game_2 challenger was encrypting random elements for the Client Login code, the Game_3 challenger instead encrypts using the real password. Indistinguishability follows by using

the IND-anon-ID-CCA property of the IBE scheme. Anonymity is needed because the id also depends on the password. CCA security is needed because in the Server Login code, we may need to perform decryption on the same id for which the encryption oracle may have responded in the Client Login code.

Finally, in Game_4 we change the challenger code which was keeping track of previously sent encryptions, to instead just behave as the real world challenger. Indistinguishability follows by the random oracle assumption on $\mathcal{H}$ and the collision resistance property of f. We then observe that the Game_4 challenger mirrors real world challenger, which concludes the proof.

4 Implementation of IBE Construction

In order to assess the practicality of Π_{pwd}, we give a concrete instantiation of the credential-hiding login scheme along with performance benchmarks. In the following, we describe this direct construction using a concrete CCA-secure anonymous IBE scheme, aimed for practical applications which intend to use credential-hiding login for the specific case of equality checks (passwords).

The Π_{BF} construction. More specifically, we can instantiate Π_{pwd} with the Boneh-Franklin anonymous IBE [BF01], where security is based on the Decisional Bilinear Diffie Hellman assumption. Let $p \in \Omega(\mathsf{poly}(\lambda))$ be the order of the groups $\mathbb{G}_1$, $\mathbb{G}_2$, and $\mathbb{G}_T$, and let $e : \mathbb{G}_1 \times \mathbb{G}_2 \to \mathbb{G}_T$ represent the pairing operation for a target group $\mathbb{G}_T$. Recall that we use $\mathcal{X}$ to represent the credential domain. Let $\mathcal{H}_1 : \mathcal{U} \times \mathcal{X} \to (\mathbb{G}_1, \mathbb{G}_T)$, $\mathcal{H}_2 : \mathbb{G}_T \to \{0,1\}^\lambda$, $\mathcal{H}_3 : \{0,1\}^\lambda \times \mathcal{X} \to \mathbb{Z}_p^*$, $\mathcal{H}_4 : \{0,1\}^\lambda \to (\{0,1\}^\lambda \times \mathcal{I}_2 \times \{0,1\}^\lambda)$, and $\mathcal{H}_5 : \{0,1\}^\lambda \to \{0,1\}^\lambda$ be hash functions modeled as random oracles. We describe the construction Π_{BF} as follows:

- $\mathsf{Setup}(1^\lambda) \to (\mathsf{pp}, \mathsf{sk})$. The Setup algorithm samples a random group element $\mathsf{g} \in \mathbb{G}_2$, a random secret scalar $k \in \mathbb{Z}_p^*$, and computes $\mathsf{h} = \mathsf{g}^k$. The public parameters are $\mathsf{pp} = (\mathsf{g}, \mathsf{h})$ and the secret key is $\mathsf{sk} = k$.

- $\mathsf{Register}(\mathsf{sk}, \mathsf{uid}, \mathsf{y}) \to \mathsf{v}$. The $\mathsf{Register}$ algorithm takes as input a credential $\mathsf{y} \in \mathcal{X}$, sets $(\mathsf{s}, \mathsf{t}) = \mathcal{H}_1(\mathsf{uid}, \mathsf{y})$, and outputs $\gamma = (\mathsf{s}^k, \mathcal{H}_5(\mathsf{t}))$.

- $\mathsf{ClientLogin}(\mathsf{pp}, \mathsf{ssid}, \mathsf{tok}, \mathsf{uid}, \mathsf{x}) \to \alpha$. The client login algorithm takes as input a set of public parameters pp, a subsession id $\mathsf{ssid} \in \mathcal{I}_2$, a token $\mathsf{tok} \in \{0,1\}^\lambda$, a user identifier $\mathsf{uid} \in \mathcal{U}$, and credential $\mathsf{x} \in \mathcal{X}$. It computes $(\mathsf{s}, \mathsf{t}) \leftarrow \mathcal{H}_1(\mathsf{uid}, \mathsf{x})$, selects a random $\sigma \leftarrow_{\mathsf{R}} \{0,1\}^\lambda$ and sets $r \leftarrow \mathcal{H}_3(\sigma, \mathsf{t})$, and then outputs the message

$$\alpha \coloneqq (\mathsf{g}^r, \sigma \oplus \mathcal{H}_2(e(\mathsf{s}, \mathsf{h}^r)), (\mathsf{t}, \mathsf{ssid}, \mathsf{tok}) \oplus \mathcal{H}_4(\sigma)).$$

- $\mathsf{ServerLogin}(\mathsf{pp}, \mathsf{ssid}, \gamma, \alpha) \to \mathsf{tok}$. The server login algorithm takes in the public parameters pp, a subsession id ssid^*, a credential verifier $(\gamma_1, \gamma_2) \coloneqq \gamma$, and the client message $(\alpha_1, \alpha_2, \alpha_3) \coloneqq \alpha$. It computes $\sigma \leftarrow \alpha_2 \oplus \mathcal{H}_2(e(\alpha_1, \gamma_1))$, and then sets $(\mathsf{t}, \mathsf{ssid}^*, \mathsf{tok}) \leftarrow \alpha_3 \oplus \mathcal{H}_4(\sigma)$. Next, it computes $r \leftarrow \mathcal{H}_3(\sigma, \mathsf{t})$ and checks if $\alpha_1 = \mathsf{g}^r$, $\mathsf{ssid}^* = \mathsf{ssid}$, and $\gamma_2 = \mathcal{H}_5(\mathsf{t})$. If any of these checks fail, then the output is $\perp$. Otherwise, it outputs tok.

Note that both the Client Login and Server Login steps only require a single pairing operation, which we have experimentally verified is on the same order of magnitude as doing an exponentiation.

Procedures	Hash Evaluations	Overall Runtime (μs)
Setup	0	668
Register	2	382
Client Login	4	3106
Server Login	4	2641

Table 2: Our experimental benchmarks measured in microseconds of wall time, averaged over 1000 iterations on a 2.4GHz Intel Skylake CPU with 16MiB L3 cache.

4.1 Concrete Instantiation

Our implementation is written using Rust and is available open-source[4]. Overall, we target 128 bits of security for our selection of primitives. We use the MIRACL Rust library for our base pairing operations over the BLS12-381 curve. We model our SHA3 evaluations (in SHAKE256 mode) as random oracle evaluations in Π_{BF}. For producing random scalars from seeded randomness, we use the HKDF "Extract" and "Expand" functions to obtain pseudorandom outputs. We remark that in a production implementation of our credential-hiding login scheme (and more generally, any login protocol), the server must keep a list of credential files for each registered user, and typically uses the uid parameter as a key to locate the associated credential file.

Furthermore, the subsession id parameter (denoted by ssid) is a session identifier that can be optionally reused from an encapsulating TLS session if the goal is the establish a secure session. This parameter, along with the token tok used to signify a login success, can also be omitted (or simply set to the empty string) depending on the application.

Password hashing functions. We note that in a typical production implementation of a login protocol, the credentials stored within the server are protected against offline dictionary attacks in the event of a server compromise through the use of memory-hard password hashing functions such as scrypt and PBKDF2. The number of iterations for these password hashing functions is often selected to be as expensive as possible, without having a significant impact on user experience. Therefore, it is likely to be the case that the execution of the iterations of the password hashing function would dominate the execution time of the remaining components of Π_{BF}.

Nevertheless, we demonstrate in the following section that Π_{BF} is quite reasonably performant without factoring in the password hashing function, and when compared against other existing solutions.

Evaluation. Our benchmarks are presented in Table 2, highlighting the total number of hash evaluations and overall runtime for each procedure. We emphasize that these benchmarks are intended primarily as a means to gauge the overall practicality of Π_{pwd}. In addition to the performance, we note that each client login message consists of three 256-bit components, for a total of 96 bytes per login message[5]. The password file is a single group element and is hence only 32 bytes.

[4] `https://github.com/pmohassel/chl`

[5] This is assuming that ssid and tok are unused, which may often be the case for a normal password-based login mechanism over the web.

We believe that these metrics are quite comparable to existing password login based schemes today (both in terms of bandwidth per client login message and space occupied per password file).

5 Construction for Fuzzy Credentials

In this section, we present a construction of credential-hiding login for fuzzy credentials, along with a description of the simulator for our UC definition. Our construction relies on a multitide of primitives, which we review in the following definitions. Recall the definition of a CCA-secure IBE scheme from Section 3, though without the need for the anonymity property (unlike Π_{pwd}). We then review the definitions for a CCA-secure public-key encryption scheme, a reusable t-fuzzy extractor, and a distributional virtual black-box (VBB) obfuscator.

5.1 Primitives

Let $\mathsf{PKE} = (\mathsf{PKE.Setup}, \mathsf{PKE.Encrypt}, \mathsf{PKE.Decrypt})$ be a public-key encryption scheme. Security is captured with the following definition.

Definition 2 ($\mathsf{Expt}_b^{\mathsf{CCA}}(\mathcal{A})$)**.** For a bit b, recall the experiment $\mathsf{Expt}_b^{\mathsf{CCA}}$ being defined as follows. First, the challenger runs $(\mathsf{pp}, \mathsf{sk}) \leftarrow \mathsf{PKE.Setup}(1^\lambda)$ and sends pp to $\mathcal{A}$. For an integer Q, the adversary $\mathcal{A}$ can (adaptively) make Q queries to a decryption oracle hosted by the challenger, by submitting ciphertexts c_i for each $i \in [1, Q]$ and receiving $\mathsf{PKE.Decrypt}(\mathsf{sk}, c_i)$ as a response from the challenger. At some point, the adversary submits two challenge messages m_0^* and m_1^*, and the challenger returns $c^* \leftarrow \mathsf{PKE.Encrypt}(\mathsf{pp}, m_b^*)$ to $\mathcal{A}$. If there exists some $i \in [1, Q]$ for which $c^* = c_i$, then the experiment halts and outputs $\perp$. The adversary outputs a bit b', which is also the output of the experiment. We say that PKE is CCA-secure if for every efficient adversary $\mathcal{A}$, the quantity

$$\left|\Pr[\mathsf{Expt}_0^{\mathsf{CCA}}(\mathcal{A}) = 1] - \Pr[\mathsf{Expt}_1^{\mathsf{CCA}}(\mathcal{A}) = 1]\right|$$

is negligible in the security parameter λ.

Let $\mathsf{FuzzyExt} = (\mathsf{Gen}, \mathsf{Rep})$ be a reusable t-fuzzy extractor from [CFP+16b]. Correctness implies that, for a domain of inputs $\{0,1\}^\lambda$, for every $x, y \in \{0,1\}^\lambda$, if $d(x, y) \leq t$, then for $(r, p) \leftarrow \mathsf{Gen}(x)$, we have that $\mathsf{Rep}(y, p) = r$. Security is captured with the following definition.

Definition 3 ($\mathsf{Expt}_b^{\mathsf{FE}}(\mathcal{A})$)**.** For a bit b, recall the experiment $\mathsf{Expt}_b^{\mathsf{FE}}$ between a challenger and adversary $\mathcal{A}$ being defined as follows. For a distribution W of inputs and integer $N = \mathsf{poly}(\lambda)$, the challenger samples inputs $w_1, \ldots, w_N \leftarrow_{\mathsf{R}} W$ and computes $(r_i, p_i) \leftarrow \mathsf{Gen}(w_i)$. If $b = 0$, it sends (r_i, p_i) to $\mathcal{A}$; otherwise if $b = 1$, it samples random $u_1, \ldots, u_N \leftarrow_{\mathsf{R}} \{0,1\}^\lambda$ and sends (u_i, p_i) to $\mathcal{A}$. The adversary outputs a bit, which is also the output of the experiment. We say that $\mathsf{FuzzyExt}$ is secure if for every efficient adversary $\mathcal{A}$, the quantity

$$\left|\Pr[\mathsf{Expt}_0^{\mathsf{FE}}(\mathcal{A}) = 1] - \Pr[\mathsf{Expt}_1^{\mathsf{FE}}(\mathcal{A}) = 1]\right|$$

is negligible in λ.

Let Obf represent a distributional VBB obfuscator as defined in [WZ17]. Obf takes as input a program $P : \mathcal{X} \to \{0,1\}^\lambda \cup \{\perp\}$ and outputs an obfuscated program $\hat{P}$, such that for all inputs $\alpha \in$

$\mathcal{X}$, $P(\alpha) = \hat{P}(\alpha)$. Security intuitively captures the property that having access to the description of the obfuscated program can be simulated by having only oracle access to the program itself, under the assumption that the program P is sampled from a sufficiently unpredictable distribution. In this work, we will focus on the obfuscation of multi-bit compute-and-compare (MBCC) programs. We write $\mathbf{MBCC}[f, y, z]$ to denote the program that on input α, outputs z if $f(\alpha) = y$, and outputs $\perp$ otherwise.

Definition 4 (Computationally unpredictable MBCC distribution)**.** We say that a distribution D of MBCC program is unpredictable if the following holds. Denoting $\mathbf{MBCC}[f, y, z]$ as the program sampled from D by a challenger, the probability that an adversary, given only f and z, can output y, is negligible in λ.

Definition 5 (Distributional VBB obfuscator for MBCC programs)**.** Let $\mathcal{A}$ be an adversary, and let D be a computationally unpredictable MBCC distribution of programs. Then, there exists a simulator Sim, which has only oracle access to $P(\cdot)$, for which the quantity

$$\left|\Pr[\mathcal{A}(\mathsf{Obf}(P)) = 1] - \Pr[\mathsf{Sim}^{P(\cdot)}(1^\lambda) = 1]\right|$$

is negligible in λ.[6]

Wichs and Zirdelis [WZ17] and Goyal, Koppula and Waters [GKW17] show that a distributional VBB obfuscator for MBCC programs can be realized based on the Learning with Errors assumption.

5.2 The Construction Π_{fz}

For simplicity, we assume that all key domains will be in $\{0,1\}^\lambda$, and let $\mathcal{X}$ be the domain of the credentials. Let $\mathsf{PKE} = (\mathsf{PKE.Setup}, \mathsf{PKE.Encrypt}, \mathsf{PKE.Decrypt})$ represent a CCA-secure public-key encryption scheme, and let $\mathsf{IBE} = (\mathsf{IBE.Setup}, \mathsf{IBE.KeyGen}, \mathsf{IBE.Encrypt}, \mathsf{IBE.Decrypt})$ represent a (public-key) CCA-secure identity-based encryption scheme. Let Obf represent an obfuscator for compute-and-compare programs which satisfies distributional virtual black-box security. Let $\mathcal{H}$ be a hash function modeled after a random oracle. Let $\mathsf{FuzzyExt} = (\mathsf{Gen}, \mathsf{Rep})$ represent a t-reusable fuzzy extractor. We define the following function which represents an inner routine for the program we will obfuscate:

$$f_{\mathsf{sk_{pke}},\mathsf{uid},\sigma}(\alpha) \coloneqq \mathcal{H}(\mathsf{uid}, \mathsf{Rep}(\mathsf{PKE.Decrypt}(\mathsf{sk_{pke}}, \alpha), \sigma)).$$

The construction Π_{fz} is described as follows:

- $\mathsf{Setup}(1^\lambda) \to (\mathsf{pp}, \mathsf{sk})$. The Setup algorithm computes $(\mathsf{pp_{pke}}, \mathsf{sk_{pke}}) \leftarrow \mathsf{PKE.Setup}(1^\lambda)$ and $(\mathsf{pp_{ibe}}, \mathsf{sk_{ibe}}) \leftarrow \mathsf{IBE.Setup}(1^\lambda)$ and outputs $\mathsf{pp} = (\mathsf{pp_{pke}}, \mathsf{pp_{ibe}})$ and $\mathsf{sk} = (\mathsf{sk_{pke}}, \mathsf{sk_{ibe}})$.
- $\mathsf{Register}(\mathsf{sk}, \mathsf{uid}, \mathsf{y}) \to \mathsf{v}$. The $\mathsf{Register}$ algorithm takes as input a secret key $\mathsf{sk} = (\mathsf{sk_{pke}}, \mathsf{sk_{ibe}})$, an identifier uid, a credential y, computes $(r, \sigma) \leftarrow \mathsf{Gen}(\mathsf{y})$, computes $\mathsf{sk_{uid}} \leftarrow \mathsf{IBE.KeyGen}(\mathsf{sk_{ibe}}, \mathsf{uid})$, and outputs $\gamma = \hat{P} \leftarrow \mathsf{Obf}(1^\lambda, \mathbf{MBCC}[f_{\mathsf{sk_{pke}},\mathsf{uid},\sigma}, \mathcal{H}(\mathsf{uid}, r), \mathsf{sk_{uid}}])$.
- $\mathsf{ClientLogin}(\mathsf{pp}, \mathsf{ssid}, \mathsf{tok}, \mathsf{uid}, \mathsf{x}) \to \alpha$. The client login algorithm takes as input a set of public parameters pp, an identifier uid, a credential x, computes $\alpha_1 \leftarrow \mathsf{PKE.Encrypt}(\mathsf{pp_{pke}}, \mathsf{x})$, $\alpha_2 \leftarrow \mathsf{IBE.Encrypt}(\mathsf{pp_{ibe}}, (\mathsf{ssid}, \mathsf{tok}, \alpha_1), \mathsf{uid})$, and returns $\alpha \leftarrow (\alpha_1, \alpha_2)$.

[6] Note that we omit the inclusion of auxiliary information and the program parameters in this description for simplicity, but these are certainly accounted for in the security proof.

- $\mathsf{ServerLogin}(\mathsf{pp}, \mathsf{ssid}, \gamma, \alpha) \to \mathsf{tok}$. The server login algorithm takes in the public parameters pp, a credential verifier $\gamma = \hat{P}$, the client message $\alpha = (\alpha_1, \alpha_2)$, and computes $\mathsf{sk}_{\mathsf{uid}} \leftarrow \hat{P}(\alpha_1)$, outputting $\perp$ if $\mathsf{sk}_{\mathsf{uid}} = \perp$. Then, it computes $(\mathsf{ssid}', \mathsf{tok}, \alpha_1') \leftarrow \mathsf{IBE.Decrypt}(\mathsf{sk}_{\mathsf{uid}}, \alpha_2)$, outputting $\perp$ if either decryption fails, $\alpha_1 \neq \alpha_1'$, or $\mathsf{ssid} \neq \mathsf{ssid}'$. Otherwise, it outputs tok.

Note that the security of all of the primitives used in this construction can be based on random oracles and the LWE assumption.

Correctness. For any identifier uid and any two credentials $\mathsf{x}, \mathsf{y} \in \mathcal{X}$, for $(\mathsf{pp}, \mathsf{sk}) \leftarrow \mathsf{PKE.Setup}(1^\lambda)$, and $\gamma \leftarrow \hat{P}$, with $(r, \sigma) \leftarrow \mathsf{Gen}(\mathsf{y})$, by the correctness of the obfuscator and fuzzy extractor, the $\mathsf{ServerLogin}$ algorithm outputs $\hat{P}(\alpha) = \mathcal{H}(\mathsf{uid}, \mathsf{Rep}(\mathsf{PKE.Decrypt}(\mathsf{sk}, \mathsf{PKE.Encrypt}(\mathsf{pp}, \mathsf{x})), \sigma))$, which is equal to $\mathcal{H}(\mathsf{uid}, r)$ if and only if $\mathsf{d}(\mathsf{x}, \mathsf{y}) \leq t$. Hence, the obfuscator outputs 1 if and only if $\mathsf{d}(\mathsf{x}, \mathsf{y}) \leq t$, which is the $\mathsf{ServerLogin}$ output as well.

5.3 Description of the Simulator

In the following description, we consider a simulator which additionally responds to Obfuscator Oracle queries made by $\mathcal{A}$. The adversary can submit queries to this obfuscator oracle for program identifiers for which it has received the corresponding obfuscations as computed by the Steal Credential Verifier Oracle. Consequently, the adversary does not receive any response from the Steal Credential Verifier Oracle other than a program identifier pid.

Note that this deviates from the adversary's interface described in the real world in Section 2. But in Appendix B, we show how, assuming our obfuscator satisfies the distributional VBB property, that the indistinguishability between the real world and ideal world with this extended adversary's interface implies the indistinguishability between the real world and ideal world *without* the addition of the obfuscator oracle. For the ease of exposition, we defer this argument to the conclusion of the proof of security, and begin with a description of the simulator with the obfuscator oracle as follows.

The simulator $\mathcal{A}^*$ will maintain the following databases across oracle queries:

- A database of registration keys with setup types $\mathcal{D}_{\mathsf{key}} \leftarrow \{\}$ which is used to keep track of setup keypairs $(\mathsf{pp}, \mathsf{sk})$ along with their setup type, being either HideRegKey or RevealRegKey,
- A map of credential file statuses $\mathcal{D}_{\mathsf{status}} : \mathcal{I}_1 \times \mathcal{U} \to \{\text{Registered}, \text{Restricted}, \text{Stolen}\}$,
- A database of client login requests $\mathcal{D}_{\mathsf{sent}} \leftarrow \{\}$ which is used to keep track of entries submitted to the Client Login oracle,
- A map of ciphertexts for PKE to their corresponding plaintexts $\mathcal{D}_{\mathsf{PKE}} : \{0,1\}^* \to \{0,1\}^*$ to keep track of encryption calls made for the Client Login oracle,
- A database of random oracle queries $\mathcal{D}_{\mathsf{RO}} : \mathcal{U} \times \{0,1\}^\lambda \to \{0,1\}^\lambda$, and
- A map of program ids pid to program descriptions $\mathcal{D}_{\mathsf{obf}}$ that have been obfuscated.

For each oracle query made by an adversary $\mathcal{A}$, we show how the simulator $\mathcal{A}^*$ responds by taking advantage of the interface provided by the ideal functionality $\mathcal{F}_{\mathsf{CHL}}$.

- **Setup Oracle:** On input $(\mathsf{sid}, \mathsf{SetupType})$ from $\mathcal{F}_{\mathsf{CHL}}$, the simulator samples $(\mathsf{pp}, \mathsf{sk}) \leftarrow \Pi_{\mathsf{fz}}.\mathsf{Setup}(1^\lambda)$. If $\mathsf{SetupType} = \textsc{HideRegKey}$, then send pp to $\mathcal{A}$ and retain sk. If $\mathsf{SetupType} = \textsc{RevealRegKey}$, then send $(\mathsf{pp}, \mathsf{sk})$ to $\mathcal{A}$. In either case, add the entry $(\mathsf{sid}, \mathsf{pp}, \mathsf{sk}, \mathsf{SetupType})$ to $\mathcal{D}_{\mathsf{key}}$.

- **Steal Credential Verifier Oracle:** On input $(\mathsf{sid}, \mathsf{uid})$ from $\mathcal{A}$, if $\mathcal{D}_{\mathsf{status}}[\mathsf{sid}, \mathsf{uid}] = \textsc{Restricted}$, the simulator returns $\perp$ to $\mathcal{A}$. Otherwise, it computes $\mathsf{sk}_{\mathsf{uid}} \leftarrow \mathsf{IBE.KeyGen}(\mathsf{sk}_{\mathsf{ibe}}, \mathsf{uid})$, and then calls $\mathcal{F}_{\mathsf{CHL}}$ with the query $(\textsc{StealCredFile}, \mathsf{sid}, \mathsf{uid})$. If $\mathcal{A}^*$ receives "No Cred File" as a response, then it also sends $\perp$ to $\mathcal{A}$. Otherwise:

  - If it receives "Cred File Stolen", then Sim picks a random $s \leftarrow_{\mathsf{R}} \{0,1\}^\lambda$ and sets $\mathcal{D}_{\mathsf{RO}}[\mathsf{uid}, \perp] \leftarrow s$.
  - If it receives a credential y, then Sim picks a random $s \leftarrow_{\mathsf{R}} \{0,1\}^\lambda$ and sets $\mathcal{D}_{\mathsf{RO}}[\mathsf{uid}, \perp] \leftarrow s$.

  In either case, $\mathcal{A}^*$ responds by picking a unique identifier pid, adding the mapping $\mathsf{pid} \rightarrow \mathbf{MBCC}[f_{\mathsf{sk}_{\mathsf{pke}}, \mathsf{uid}, \sigma}, s, \mathsf{sk}_{\mathsf{uid}}]$ to $\mathcal{D}_{\mathsf{obf}}$, and returning pid to $\mathcal{A}$.

- **Client Login Oracle:** Let $(\textsc{ClientLogin}, \mathsf{sid}, \mathsf{ssid}, \mathsf{uid}, \mathsf{x}^*, \mathsf{tok}^*, \mathsf{type})$ be the query sent to $\mathcal{F}_{\mathsf{CHL}}$. On input $(\mathsf{sid}, \mathsf{ssid}, \mathsf{uid}, \mathsf{type})$ from $\mathcal{F}_{\mathsf{CHL}}$, if there is an entry of the form $(\mathsf{sid}, \star, \star, \textsc{RevealRegKey})$ in $\mathcal{D}_{\mathsf{key}}$, or if there is no entry of the form $(\mathsf{sid}, \star, \star, \star)$ in $\mathcal{D}_{\mathsf{key}}$, or $\mathcal{D}_{\mathsf{status}}[\mathsf{sid}, \mathsf{uid}] =$ "Cred File Stolen" and $\mathsf{type} = \textsc{Correct}$, then $\mathcal{A}^*$ returns $\perp$ to $\mathcal{A}$. Otherwise, let pp be the unique parameter for which $(\mathsf{sid}, \mathsf{pp}, \star, \star)$ exists in $\mathcal{D}_{\mathsf{key}}$. Then, $\mathcal{A}^*$ randomly samples a credential $\mathsf{x} \leftarrow \mathcal{X}$ and token $\mathsf{tok} \in \{0,1\}^\lambda$ and returns $\alpha = (\alpha_1, \alpha_2) \leftarrow \Pi_{\mathsf{fz}}.\mathsf{ClientLogin}(\mathsf{pp}, \mathsf{ssid}, \mathsf{tok}, \mathsf{uid}, \mathsf{x})$ to $\mathcal{A}$. Finally, if $\mathsf{type} \in \{\textsc{Invalid}, \textsc{Correct}\}$, then $\mathcal{A}^*$ sets $\mathcal{D}_{\mathsf{status}}[\mathsf{sid}, \mathsf{uid}] \leftarrow \textsc{Restricted}$, adds the entry $(\mathsf{sid}, \mathsf{ssid}, \mathsf{uid}, \alpha)$ to $\mathcal{D}_{\mathsf{sent}}$, and sets $\mathcal{D}_{\mathsf{PKE}}[\alpha_1] \leftarrow \mathsf{x}$.

- **Server Login Oracle:** On input $(\mathsf{sid}, \mathsf{ssid}, \mathsf{uid}, \alpha)$ from $\mathcal{A}$, let $(\alpha_1, \alpha_2) = \alpha$. There are two cases to consider:

  - **Case 1:** There is already an entry of the form $(\star, \star, \star, \alpha)$ in $\mathcal{D}_{\mathsf{sent}}$. Let $(\mathsf{sid}', \mathsf{ssid}', \mathsf{uid}', \alpha)$ be this entry. If the two tuples $(\mathsf{sid}, \mathsf{ssid}, \mathsf{uid})$ and $(\mathsf{sid}', \mathsf{ssid}', \mathsf{uid}')$ are equal in all positions, then set $\mathsf{tok} = \perp$ and $\mathsf{flag} = \perp$. Otherwise, set $\mathsf{tok} = \perp$, $\mathsf{flag} = \textsc{Failure}$, and call $\mathcal{F}_{\mathsf{CHL}}$ with the query $(\textsc{TestCred}, \mathsf{sid}, \mathsf{ssid}, \mathsf{uid}, \perp)$.
  - **Case 2:** There is no entry of the form $(\star, \star, \star, \alpha)$ in $\mathcal{D}_{\mathsf{sent}}$. The simulator computes $\mathsf{sk}_{\mathsf{uid}} \leftarrow \mathsf{IBE.KeyGen}(\mathsf{sk}_{\mathsf{ibe}}, \mathsf{uid})$ and $(\mathsf{ssid}', \mathsf{tok}', \alpha_1') \leftarrow \mathsf{IBE.Decrypt}(\mathsf{sk}_{\mathsf{uid}}, \alpha_2)$ and checks if $\mathsf{ssid} = \mathsf{ssid}'$ and $\alpha_1 = \alpha_1'$, and then computes $\mathsf{x} \leftarrow \mathsf{PKE.Decrypt}(\mathsf{sk}_{\mathsf{pke}}, \alpha_1)$. If all of these checks pass, then the simulator sets $\mathsf{tok} \leftarrow \mathsf{tok}'$, $\mathsf{flag} = \textsc{Success}$, and calls $\mathcal{F}_{\mathsf{CHL}}$ with $(\textsc{TestCred}, \mathsf{sid}, \mathsf{ssid}, \mathsf{uid}, \mathsf{x})$. Otherwise, if any of the above steps fail, then it sets $\mathsf{tok} = \perp$, $\mathsf{flag} = \textsc{Failure}$, and calls $\mathcal{F}_{\mathsf{CHL}}$ with $(\textsc{TestCred}, \mathsf{sid}, \mathsf{ssid}, \mathsf{uid}, \perp)$.

  After all cases, call $\mathcal{F}_{\mathsf{CHL}}$ with $(\textsc{Login}, \mathsf{sid}, \mathsf{ssid}, \mathsf{uid}, \mathsf{tok}, \mathsf{flag})$.

- **Random Oracle:** On input (uid, r) from $\mathcal{A}$, the simulator returns $\mathcal{D}_{\mathsf{RO}}[\mathsf{uid}, r]$ if it exists. Otherwise, it samples an $s \leftarrow_{\mathsf{R}} \{0,1\}^\lambda$ uniformly at random, sets $\mathcal{D}_{\mathsf{RO}}[\mathsf{uid}, r] \leftarrow s$, and returns s.

- **Obfuscator Oracle:** On input a program identifier pid and input α from $\mathcal{A}$, Sim first checks if pid is an index in $\mathcal{D}_{\mathsf{obf}}$, returning $\perp$ if not. Otherwise, let $\mathbf{MBCC}[f_{\mathsf{sk}_{\mathsf{pke}},\mathsf{uid},\sigma}, s, \mathsf{sk}_{\mathsf{uid}}]$ be the program associated with pid. Sim checks if α is set in $\mathcal{D}_{\mathsf{PKE}}$, letting $\mathsf{x} \leftarrow \mathcal{D}_{\mathsf{PKE}}[\alpha]$ if so. Otherwise, Sim computes $\mathsf{x} \leftarrow \mathsf{PKE.Decrypt}(\mathsf{sk}_{\mathsf{PKE}}, \alpha)$ and calls $\mathcal{F}_{\mathsf{CHL}}$ with ($\textsc{OfflineTestCred}, \mathsf{sid}, \mathsf{uid}, \mathsf{x}$). If the response is "$\textsc{Correct Guess}$", then:

  - Sim checks if there exists an entry of the form $\mathcal{D}_{\mathsf{RO}}[\mathsf{uid}, \perp] = s$, and if so, then Sim picks a random $r \leftarrow_{\mathsf{R}} \{0,1\}^{\lambda}$ and sets $\mathcal{D}_{\mathsf{RO}}[\mathsf{uid}, r] \leftarrow s$ (erasing the entry for $\mathcal{D}_{\mathsf{RO}}[\mathsf{uid}, \perp]$).
  - Then, Sim returns $\mathsf{sk}_{\mathsf{uid}}$ to $\mathcal{A}$.

  Otherwise, Sim returns $\perp$ to $\mathcal{A}$.

We use the simulator defined above in order to prove the following theorem.

Theorem 2. *The protocol Π_{fz} securely realizes the $\mathcal{F}_{\mathsf{CHL}}$ functionality in the UC model, assuming $\mathsf{FuzzyExt}$ is a reusable fuzzy extractor, IBE satisfies IND-ID-CCA security, PKE satisfies CCA security, $\mathcal{H}$ is a random oracle, and the input credential distribution $\mathcal{X}$ has min-entropy at least λ.*

We note that all of the primitives described above can be achieved from the LWE assumption (except for the modeling of $\mathcal{H}$ as a random oracle).

Proof sketch. To prove that Π_{fz} satisfies our UC formulation of a credential-hiding login protocol, we can proceed with a series of intermediate games (similar to the proof of security for Π_{pwd}), with the notable exception that we ultimately rely on the distributional VBB obfuscation property to establish indistinguishability between the real-world protocol and the above simulator description which gives the adversary access to the Obfuscator Oracle.

In order to satisfy the requirements for applying distributional VBB obfuscation for a given function $\mathbf{MBCC}[f, y, z]$, we claim that the value y is computationally unpredictable given access to the description of f and any auxiliary information required for evaluation. Recall that in our instantiation, we obfuscate the function $\mathbf{MBCC}[f_{\mathsf{sk}_{\mathsf{pke}},\mathsf{uid},\sigma}, \mathcal{H}(\mathsf{uid}, r), \mathsf{sk}_{\mathsf{uid}}]$, where $(r, \sigma) \leftarrow \mathsf{Gen}(\mathsf{y})$. The security of the fuzzy extractor states that if y comes from a distribution of sufficient min-entropy, then r is sufficiently unpredictable given only σ. Putting this together, we can then conclude that $\mathcal{H}(\mathsf{uid}, r)$ is indeed unpredictable to $\mathcal{A}$, which then allows us to replace the adversary's access to $\hat{P}$ with the oracle access to the Obfuscator Oracle in our simulator. We provide the full proof in Appendix B.

6 Conclusions

In this work, we introduced the notion of credential-hiding login, along with a formulation of security within the UC framework. Credential-hiding login can be seen as a special case of the more general primitive: authenticated login, which when used within an already-established server-authenticated channel, suffices for the user login scenario. We then presented two constructions of credential-hiding login: Π_{pwd} for handling password-based login that we also implemented as a proof-of-concept, and Π_{fz} for handling fuzzy credentials, as a generalization of the equality checks done by passwords to Hamming distance checks for approximate equality.

To conclude, we present the following directions for future study:

- Can we have CHL constructions resistant to precomputation attacks (as defined by the distinction between "strong aPAKEs" versus regular aPAKEs)?
- Can we further improve the performance of password-based credential-hiding login, and show how Π_{pwd} performs when instantiated within a production-grade deployment of a server-authenticated channel along with a real-world password login form?
- In our work, we have shown that the constructions Π_{pwd} and Π_{fz} satisfy the notion of credential-hiding login. Can a simple modification of these constructions be shown to meet the security notions for UC aPAKEs and UC fuzzy aPAKEs, respectively?
- More generally, can we show black box reductions or separations among CHL and aPAKE functionalities?

References

[AMMR18] Shashank Agrawal, Payman Mohassel, Pratyay Mukherjee, and Peter Rindal. Dise: distributed symmetric-key encryption. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 1993–2010, 2018.

[AP05] Michel Abdalla and David Pointcheval. Simple password-based encrypted key exchange protocols. In *Cryptographers' track at the RSA conference*, pages 191–208. Springer, 2005.

[BBR88] Charles H Bennett, Gilles Brassard, and Jean-Marc Robert. Privacy amplification by public discussion. *SIAM journal on Computing*, 17(2):210–229, 1988.

[BCF+14] Julien Bringer, Herve Chabanne, Melanie Favre, Alain Patey, Thomas Schneider, and Michael Zohner. Gshade: faster privacy-preserving distance computation and biometric identification. In *Proceedings of the 2nd ACM workshop on Information hiding and multimedia security*, pages 187–198, 2014.

[BCS16] Dan Boneh, Henry Corrigan-Gibbs, and Stuart E. Schechter. Balloon hashing: A memory-hard function providing provable protection against sequential attacks. In Jung Hee Cheon and Tsuyoshi Takagi, editors, *ASIACRYPT 2016*, volume 10031 of *Lecture Notes in Computer Science*, pages 220–248, 2016.

[BDK+05] Xavier Boyen, Yevgeniy Dodis, Jonathan Katz, Rafail Ostrovsky, and Adam Smith. Secure remote authentication using biometric data. In *annual international conference on the theory and applications of cryptographic techniques*, pages 147–163. Springer, 2005.

[BDK16] Alex Biryukov, Daniel Dinu, and Dmitry Khovratovich. Argon2: New generation of memory-hard functions for password hashing and other applications. In *IEEE European Symposium on Security and Privacy, EuroS&P 2016, Saarbrücken, Germany, March 21-24, 2016*, pages 292–302. IEEE, 2016.

[BF01] Dan Boneh and Matt Franklin. Identity-based encryption from the weil pairing. In *Annual international cryptology conference*, pages 213–229. Springer, 2001.

[BLMR13] Dan Boneh, Kevin Lewi, Hart Montgomery, and Ananth Raghunathan. Key homomorphic prfs and their applications. In *Annual Cryptology Conference*, pages 410–428. Springer, 2013.

[BM93a] Steven M Bellovin and Michael Merritt. Augmented encrypted key exchange: a password-based protocol secure against dictionary attacks and password file compromise. In *CCS*, pages 244–250, 1993.

[BM93b] Steven M Bellovin and Michael Merritt. Augmented encrypted key exchange: a password-based protocol secure against dictionary attacks and password file compromise. In *Proceedings of the 1st ACM Conference on Computer and Communications Security*, pages 244–250, 1993.

[BMP00] Victor Boyko, Philip MacKenzie, and Sarvar Patel. Provably secure password-authenticated key exchange using diffie-hellman. In *International Conference on the Theory and Applications of Cryptographic Techniques*, pages 156–171. Springer, 2000.

[Boy04] Xavier Boyen. Reusable cryptographic fuzzy extractors. In *Proceedings of the 11th ACM conference on Computer and communications security*, pages 82–91, 2004.

[BPR00] Mihir Bellare, David Pointcheval, and Phillip Rogaway. Authenticated key exchange secure against dictionary attacks. In *International conference on the theory and applications of cryptographic techniques*, pages 139–155. Springer, 2000.

[Can01] Ran Canetti. Universally composable security: A new paradigm for cryptographic protocols. In *FOCS*, pages 136–145. IEEE, 2001.

[CFP+16a] Ran Canetti, Benjamin Fuller, Omer Paneth, Leonid Reyzin, and Adam Smith. Reusable fuzzy extractors for low-entropy distributions. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 117–146. Springer, 2016.

[CFP+16b] Ran Canetti, Benjamin Fuller, Omer Paneth, Leonid Reyzin, and Adam D. Smith. Reusable fuzzy extractors for low-entropy distributions. In *EUROCRYPT*, pages 117–146, 2016.

[CHK+05] Ran Canetti, Shai Halevi, Jonathan Katz, Yehuda Lindell, and Philip D. MacKenzie. Universally composable password-based key exchange. In *EUROCRYPT*, pages 404–421, 2005.

[CK02] Ran Canetti and Hugo Krawczyk. Universally composable notions of key exchange and secure channels. In *International Conference on the Theory and Applications of Cryptographic Techniques*, pages 337–351. Springer, 2002.

[Coi19] Post Mortem: A closer look at a password storage issue affecting 3,420 customers. `https://blog.coinbase.com/post-mortem-a-closer-look-at-a-password-storage-issue-affecting-3-420-customers-e23cfc8a0363`, August 16, 2019.

[DHP+18] Pierre-Alain Dupont, Julia Hesse, David Pointcheval, Leonid Reyzin, and Sophia Yakoubov. Fuzzy password-authenticated key exchange. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 393–424. Springer, 2018.

[DORS08] Yevgeniy Dodis, Rafail Ostrovsky, Leonid Reyzin, and Adam Smith. Fuzzy extractors: How to generate strong keys from biometrics and other noisy data. *SIAM journal on computing*, 38(1):97–139, 2008.

[DS05] Yevgeniy Dodis and Adam Smith. Correcting errors without leaking partial information. In *Proceedings of the thirty-seventh annual ACM symposium on Theory of computing*, pages 654–663, 2005.

[ECS+15] Adam Everspaugh, Rahul Chaterjee, Samuel Scott, Ari Juels, and Thomas Ristenpart. The Pythia PRF Service. In *USENIX Security Symposium*, pages 547–562, 2015.

[EHKM11] David Evans, Yan Huang, Jonathan Katz, and Lior Malka. Efficient privacy-preserving biometric identification. In *Proceedings of the 17th conference Network and Distributed System Security Symposium, NDSS*, volume 68, 2011.

[EHOR20] Andreas Erwig, Julia Hesse, Maximilian Orlt, and Siavash Riahi. Fuzzy asymmetric password-authenticated key exchange. *IACR Cryptol. ePrint Arch.*, 2020:987, 2020.

[Eve13] Tammy Everts. Rules for mobile performance optimization. *Communications of the ACM*, 56:52–59, 08 2013.

[Fac19] Keeping Passwords Secure. `https://about.fb.com/news/2019/03/keeping-passwords-secure/`, March 21, 2019.

[Git18] GitHub says bug exposed some plaintext passwords. `https://www.zdnet.com/article/github-says-bug-exposed-account-passwords/`, May 1, 2018.

[GKW17] Rishab Goyal, Venkata Koppula, and Brent Waters. Lockable obfuscation. In *2017 IEEE 58th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 612–621. IEEE, 2017.

[GMR06] Craig Gentry, Philip D. MacKenzie, and Zulfikar Ramzan. A method for making password-based key exchange resilient to server compromise. In *Advances in Cryptology - CRYPTO 2006, 26th Annual International Cryptology Conference, Santa Barbara, California, USA, August 20-24, 2006, Proceedings*, pages 142–159, 2006.

[Goo19] Notifying administrators about unhashed password storage. `https://cloud.google.com/blog/products/g-suite/notifying-administrators-about-unhashed-password-storage`, May 21, 2019.

[Gro20] Crypto Forum Research Group. *CFRG PAKE Selection Process*, 2019 (accessed February, 2020).

[HEKM11] Yan Huang, David Evans, Jonathan Katz, and Lior Malka. Faster secure two-party computation using garbled circuits. In *USENIX Security Symposium*, volume 201, pages 331–335, 2011.

[HL19] Björn Haase and Benoît Labrique. Aucpace: Efficient verifier-based pake protocol tailored for the iiot. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pages 1–48, 2019.

[HR10] Feng Hao and Peter Ryan. J-pake: authenticated key exchange without pki. In *Transactions on computational science XI*, pages 192–206. Springer, 2010.

[HS14] Feng Hao and Siamak F. Shahandashti. The speke protocol revisited. Cryptology ePrint Archive, Report 2014/585, 2014. `https://eprint.iacr.org/2014/585`.

[JKX18] Stanislaw Jarecki, Hugo Krawczyk, and Jiayu Xu. Opaque: an asymmetric pake protocol secure against pre-computation attacks. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 456–486. Springer, 2018.

[JR18] Charanjit S Jutla and Arnab Roy. Smooth nizk arguments. In *Theory of Cryptography Conference*, pages 235–262. Springer, 2018.

[KM14] Franziskus Kiefer and Mark Manulis. Zero-knowledge password policy checks and verifier-based pake. In *European Symposium on Research in Computer Security*, pages 295–312. Springer, 2014.

[Mac01] Philip Mackenzie. More efficient password-authenticated key exchange. In *Cryptographers' Track at the RSA Conference*, pages 361–377. Springer, 2001.

[MKR17] Kathleen Moriarty, Burt Kaliski, and Andreas Rusch. Pkcs# 5: Password-based cryptography specification version 2.1. *Internet Engineering Task Force (IETF)*, 2017.

[MPS00] Philip MacKenzie, Sarvar Patel, and Ram Swaminathan. Password-authenticated key exchange based on rsa. In *International conference on the theory and application of cryptology and information security*, pages 599–613. Springer, 2000.

[Muf15] Alec Muffet. Facebook: Password hashing & authentication. *Presentation at Real World Crypto*, 2015.

[OPJM10] Margarita Osadchy, Benny Pinkas, Ayman Jarrous, and Boaz Moskovich. Scifi-a system for secure face identification. In *2010 IEEE Symposium on Security and Privacy*, pages 239–254. IEEE, 2010.

[Per09] Colin Percival. Stronger key derivation via sequential memory-hard functions, 2009.

[PJ16] Colin Percival and Simon Josefsson. The scrypt Password-Based Key Derivation Function. RFC 7914, August 2016.

[PM99] Niels Provos and David Mazieres. A future-adaptable password scheme. In *USENIX Annual Technical Conference, FREENIX Track*, pages 81–91, 1999.

[PW17] David Pointcheval and Guilin Wang. Vtbpeke: verifier-based two-basis password exponential key exchange. In *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security*, pages 301–312, 2017.

[Rob19] Robinhood stored passwords in plaintext, so change yours now. https://techcrunch.com/2019/07/24/robinhood-stored-passwords-in-plaintext-so-change-yours-now/, July 24, 2019.

[SKFB19] Nick Sullivan, Dr. Hugo Krawczyk, Owen Friel, and Richard Barnes. Usage of OPAQUE with TLS 1.3. Internet-Draft draft-sullivan-tls-opaque-00, Internet Engineering Task Force, March 2019. Work in Progress.

[Twi18] Twitter says bug exposed user plaintext passwords. https://www.zdnet.com/article/twitter-says-bug-exposed-passwords-in-plaintext/, May 3, 2018.

[WZ17] Daniel Wichs and Giorgos Zirdelis. Obfuscating compute-and-compare programs under lwe. In *2017 IEEE 58th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 600–611. IEEE, 2017.

Appendix

A UC Security Proof for Π_{pwd}

In the following, we use $\mathcal{A}^*$ to denote the simulator described in Section 3.1. For an adversary $\mathcal{A}$, we define a series of games, represented as $\mathsf{Game}_i(\mathcal{A})$:

- **Game 0:** Game_0 is identical to $\mathsf{Expt}_{\mathsf{ideal}}(\mathcal{A}, \mathcal{A}^*)$.
- **Game 1:** Game_1 differs from Game_0 in the following: when $\mathcal{A}$ calls the Registration Oracle, $\mathcal{D}_{\mathsf{Reg}}$ is populated as in the real world, and then when responding to the Server Login Oracle query:
  - In the abort condition of Case 2a, rather than aborting, the simulator checks $\mathcal{D}_{\mathsf{Reg}}$ for the corresponding registered credential x and instead sets $\mathsf{flag} \leftarrow$ SUCCESS and calls $\mathcal{F}_{\mathsf{CHL}}$ with (TESTCRED, $\mathsf{sid}, \mathsf{ssid}, \mathsf{uid}, \mathsf{x}$). All other steps remain unchanged.
  - In Case 2b, rather than performing a linear search over the entries of $\mathcal{D}_{\mathsf{RO}}$, the simulator checks $\mathcal{D}_{\mathsf{Reg}}$ for the corresponding registered credential x and computes $(\mathsf{s}, \mathsf{t}) \leftarrow \mathcal{H}(\mathsf{sid}, \mathsf{uid}, \mathsf{x})$, and $(\mathsf{v}, \mathsf{ssid}', \mathsf{tok}) \leftarrow \mathsf{IBE.Decrypt}(\mathsf{IBE.KeyGen}(\mathsf{sk}, \mathsf{id} = \mathsf{s}), \alpha)$. The remainder steps remain the same.
- **Game 2:** Game_2 differs from Game_1 in responding to Client Login Oracle queries: instead of computing $\alpha \leftarrow \mathsf{IBE.Encrypt}(\mathsf{pp}, \mathsf{id}, \mathsf{m})$ for randomly sampled $\mathsf{id}, m \leftarrow_{\mathsf{R}} \{0,1\}^\lambda$, it computes $\alpha \leftarrow \mathsf{IBE.Encrypt}(\mathsf{pp}, \mathsf{s}, \mathsf{t})$, where $(\mathsf{s}, \mathsf{t}) \leftarrow \mathcal{H}(\mathsf{sid}, \mathsf{uid}, \mathsf{x})$. Note that this is identical to returning $\alpha \leftarrow \mathsf{ClientLogin}(\mathsf{pp}, \mathsf{ssid}, \mathsf{tok}^*, \mathsf{uid}, \mathsf{x}^*)$.
- **Game 3:** Game_3 differs from Game_2 in the Server Login Oracle query response: in Case 2, the simulator checks $\mathcal{D}_{\mathsf{Reg}}$ for the corresponding registered verifier γ, and simply returns the result of computing $\mathsf{ServerLogin}(\mathsf{pp}, \mathsf{ssid}, \gamma, \alpha)$.

In the following lemmas, for two experiments $E_1(\mathcal{A})$ and $E_2(\mathcal{A})$, we use the notation $E_1(\mathcal{A}) \approx E_2(\mathcal{A})$ as shorthand to represent the notion that the quantity $|\Pr[E_1(\mathcal{A}) = 1] - \Pr[E_2(\mathcal{A}) = 1]|$ is negligible in λ.

Lemma 1. *For an adversary $\mathcal{A}$, $\mathsf{Game}_0(\mathcal{A}) \approx \mathsf{Game}_1(\mathcal{A})$ when $\mathcal{H}$ is modeled after a random oracle.*

Proof. To handle the change in Case 2a, we show that the probability of $\mathsf{f}(v) = \mathsf{f}(\mathsf{t})$ is negligible given that f is a random oracle. So the indistinguishability of Games 0 and 1 for this specific case follows statistically. The presence of $((\mathsf{sid}, \mathsf{uid}, *), (\mathsf{s}, \mathsf{t}))$ in $\mathcal{D}_{\mathsf{RO}}$ indicates that the password file was stolen. Now note that if we model f as a random oracle, then t and $\mathsf{f}(\mathsf{t})$ are independently random quantities. In case the adversary never does any successful online or offline attempt previously, it does not receive any more information about t, other than $\mathsf{f}(\mathsf{t})$ provided in the stolen password file. So there is a negligible chance of the event $\mathsf{v} = \mathsf{t}$ occurring and hence also of $\mathsf{f}(\mathsf{v}) = \mathsf{f}(\mathsf{t})$.

To handle the change in Case 2b, indistinguishability follows by again using the random oracle properties of $\mathcal{H}$ and the collision resistance property of f. We need two key properties in order to ensure the consistency of the trial and error search over the entries of $\mathcal{D}_{\mathsf{RO}}$:

Let $(\mathsf{s},\mathsf{t}) = \mathcal{H}(\mathsf{sid},\mathsf{uid},\mathsf{x})$ and $(\mathsf{s}',\mathsf{t}') = \mathcal{H}(\mathsf{sid}',\mathsf{uid}',\mathsf{x}')$ for any $\mathsf{x} \neq \mathsf{x}'$ and identifiers $\mathsf{sid},\mathsf{sid}' \in \mathcal{I}_2$ and $\mathsf{uid},\mathsf{uid}' \in \mathcal{U}$, and tokens $\mathsf{tok},\mathsf{tok}' \in \{0,1\}^\lambda$. Then, we argue that for any α, the following holds with statistically overwhelming probability:

$$\mathsf{IBE.Decrypt}(\mathsf{IBE.KeyGen}(\mathsf{sk},\mathsf{s}),\alpha) = (\mathsf{t},\mathsf{ssid},\mathsf{tok})$$

$$\implies \neg(\mathsf{IBE.Decrypt}(\mathsf{IBE.KeyGen}(\mathsf{sk},\mathsf{s}'),\alpha) = (\mathsf{t}',\mathsf{ssid},\mathsf{tok}') \wedge \mathsf{f}(\mathsf{t}') = \mathsf{f}(\mathsf{t}))$$

This easily follows, as $\mathsf{x} \neq \mathsf{x}'$ and $\mathcal{H}$ is a random oracle, which makes t' independently random from $\mathsf{s},\mathsf{s}',\mathsf{t}$.

We also argue that if the random oracle $\mathcal{H}$ is not called with $(\mathsf{sid},\mathsf{uid},\mathsf{x})$, then the following event is statistically negligible in probability:

$$\mathsf{IBE.Decrypt}(\mathsf{IBE.KeyGen}(\mathsf{sk},\mathsf{s}),\alpha) = (\mathsf{t},\mathsf{ssid},\mathsf{tok})$$

This is because t is independently random of the adversary's view if the random oracle is not called with the correct input. This concludes the proof. □

Note that if f were only a CROWF (Collision-Resistant OWF), the proof of Lemma 1 would not hold. Given a one-wayness challenge to invert u, we can implicitly set $\mathsf{f}(\mathsf{t}) = u$ for a randomly chosen $(\mathsf{sid},\mathsf{uid},\mathsf{x})$ registration. If the IBE decryption of an adversarially generated client login message yields a t', such that $\mathsf{f}(\mathsf{t}') = u$, then this provides the one-wayness response. However, consider the case, where an offline test takes place with the correct password in between these two events. In this case, the simulator cannot provide a valid response without inverting the challenge itself. Hence, provability seems difficult just assuming CROWF.

Lemma 2. *For an adversary $\mathcal{A}$, $\mathsf{Game}_1(\mathcal{A}) \approx \mathsf{Game}_2(\mathcal{A})$ based on the IND-anon-ID-CCA security of* IBE.

Proof. We show that Indistinguishability follows by IND-anon-ID-CCA property of the IBE scheme in Lemma 2. In the following proof, we say that sid is marked with a type $\mathsf{SetupType} \in \{$ HideRegKey, RevealRegKey$\}$ to represent that there exists an entry in $\mathcal{D}_{\mathsf{key}}$ of the form $(\mathsf{sid},\star,\star,\mathsf{SetupType})$. We describe the simulator acting as an adversary for $\mathsf{Expt}_b^{\mathsf{anon\text{-}cca}}$ as follows:

- **On receiving Setup from** S**.** Let the call parameter from S be $(\mathsf{sid},\mathsf{type})$. The simulator calls $\mathsf{IBE.Setup}(1^\lambda)$ and receives pp. If type = HideRegKey, then send pp to $\mathcal{A}$. Mark sid with HideRegKey. If type = RevealRegKey, then the simulator ignores the IBE challenger and generates $(\mathsf{pp},\mathsf{sk}) \leftarrow \mathsf{IBE.Setup}(1^\lambda)$ itself and sends $(\mathsf{pp},\mathsf{sk})$ to $\mathcal{A}$. Mark sid with RevealRegKey.

- **On receiving StealCredFile from** $\mathcal{A}$**.** This is same as Game_2, except it calls the $\mathsf{IBE.KeyGen}$ oracle provided by the challenger in order to construct its responses.

- **On receiving ClientLogin from** U**:** Let the call parameter from U be $(\mathsf{sid},\mathsf{ssid},\mathsf{uid},\mathsf{type})$. If sid is marked with RevealRegKey, or if $\mathcal{D}_{\mathsf{status}}[\mathsf{sid},\mathsf{uid}]$ = Stolen, then the simulator sends $\perp$ to the adversary. Otherwise, the simulator samples (id,m) from random and computes $(\mathsf{s},\mathsf{t}) = \mathcal{H}(\mathsf{sid},\mathsf{uid},\mathsf{x}')$ using the random oracle, and sends $((\mathsf{id},\mathsf{m}),(\mathsf{s},(\mathsf{t},\mathsf{ssid},\mathsf{tok}))$ to the IBE Encryption oracle, obtaining α. Note that this step is not performed if the registration key was revealed. It then sends $(\mathsf{sid},\mathsf{ssid},\alpha)$ to $\mathcal{A}$. If type = Invalid or type = Correct, then set $\mathcal{D}_{\mathsf{status}}[\mathsf{sid},\mathsf{uid}] \leftarrow$ Restricted.

- **On receiving a Server Login Oracle query from S.** This is same as Game 2, except it uses the IBE Decryption oracle to respond to the query.

Now we argue that the admissibility conditions for the CCA security game are obeyed. Firstly, due to the conditions in the Server Login Oracle response, decryption is never called on messages responded with by the encryption queries. Secondly, due to the RESTRICTED conditions, the IBE KeyGen oracle is never called for $(\mathsf{uid}, \mathsf{x})$ tuples for which Client Login was simulated. Lastly, Client Login is never simulated if the registration secret was revealed.

Since $\mathsf{Expt}_0^{\mathsf{anon\text{-}cca}}$ corresponds to Game_1 and $\mathsf{Expt}_1^{\mathsf{anon\text{-}cca}}$ corresponds to Game_2, the lemma follows. □

Lemma 3. *For an adversary $\mathcal{A}$, $\mathsf{Game}_2(\mathcal{A}) \approx \mathsf{Game}_3(\mathcal{A})$ when $\mathcal{H}$ is modeled after a random oracle.*

Proof. The only difference between the two games is the way the following situation is handled: $(\mathsf{sid}', \mathsf{ssid}', \mathsf{uid}', \alpha')$ was sent by an honest client with associated type type. Let's divide the cases exhaustively as follows.

First consider the case $(\mathsf{sid}', \mathsf{uid}', \mathsf{type}) = (\mathsf{sid}, \mathsf{uid}, \text{CORRECT})$. In this case, the decryption will be with the same id as the encryption. Now due to IBE correctness, login will succeed in Game 3 if the ssid's are same, and will fail if not. This behavior is same as in Game 2.

Second, where $(\mathsf{sid}', \mathsf{uid}', \mathsf{type}) \neq (\mathsf{sid}, \mathsf{uid}, \text{CORRECT})$, we will have a different decryption id than was used for encryption. In this case login cannot succeed without $\mathsf{f}(\mathsf{t}') = \mathsf{f}(\mathsf{t})$ holding. This is either a collision for f, or we have $\mathsf{t}' = \mathsf{t}$. The probability of the former is bounded by the collision-resistance of f and the probability of the later is statistically negligible as the RHS is information theoretically independent of the LHS due to the random oracle. Indistinguishability thus holds by the collision resistance of f and the random oracle assumption on $\mathcal{H}$. □

Note that Game_3 is identical to $\mathsf{Expt}_{\mathsf{real}}(\mathcal{A})$. Thus, putting together Lemmas 1, 2, and 3 concludes the proof of Theorem 1.

B UC Security Proof for Π_{fz}

In the following proof, we will consider a slight extension of $\mathsf{Expt}_{\mathsf{real}}$ and $\mathsf{Expt}_{\mathsf{ideal}}$ to support an "obfuscator oracle", in order to incorporate the use of the MBCC obfuscator — we will denote these experiments as $\mathsf{Expt}^*_{\mathsf{real}}$ and $\mathsf{Expt}^*_{\mathsf{ideal}}$. Intuitively, this extension allows for the adversary to query an oracle that provides black-box access to the program associated with the credential verifier stored upon a call to the registration oracle in the real world. Thus, instead of receiving the credential verifier upon a successful call to the Steal Credential Verifier Oracle, the adversary must instead issue queries to the Obfuscator Oracle to receive information about the credential verifier.

For an input adversary $\mathcal{A}$, we define a series of games, represented as $\mathsf{Game}_i(\mathcal{A})$:

- **Game 0:** Game_0 is identical to $\mathsf{Expt}^*_{\mathsf{ideal}}(\mathcal{A}, \mathsf{Sim})$.
- **Game 1:** Game_1 differs from Game_0 in responding to the Steal Credential Verifier Oracle query: after Sim picks $s \leftarrow_{\mathsf{R}} \{0,1\}^\lambda$, it does not set any entries in $\mathcal{D}_{\mathsf{RO}}$. Then, in response to the Obfuscator Oracle query, if the response is "CORRECT GUESS", Sim picks a random $r \leftarrow_{\mathsf{R}} \{0,1\}^\lambda$ and sets $s \leftarrow \mathcal{H}(\mathsf{uid}, r)$. Finally, on responses to the random oracle query, Sim simply returns $\mathcal{H}(\mathsf{uid}, r)$. All other steps remain unchanged.

- **Game 2:** $\mathsf{Game}_2^{(i)}$ differs from Game_1 in responding to the Steal Credential Verifier Oracle query: if it receives a credential y from $\mathcal{F}_{\mathsf{CHL}}$, then Sim computes $(r^*, \sigma) \leftarrow \mathsf{Gen}(\mathsf{y})$ and sets $s \leftarrow \mathcal{H}(\mathsf{uid}, r^*)$. Then, in response to the Obfuscator Oracle query, if the response is "Correct Guess", then Sim computes $(r^*, \sigma) \leftarrow \mathsf{Gen}(\mathsf{x})$ and sets $s \leftarrow \mathcal{H}(\mathsf{uid}, r^*)$. All other steps remain unchanged.

- **Game 3:** Let Q be the total number of Client Login Oracle queries made by $\mathcal{A}$. For each $i \in [Q]$, we define $\mathsf{Game}_3^{(i)}$ to be the following: $\mathsf{Game}_3^{(i)}$ differs from Game_2 in the Client Login Oracle's construction of the value α. Recall that in Game_2, $\alpha \leftarrow (\alpha_1, \alpha_2) = (\mathsf{PKE.Encrypt}(\mathsf{pp}_{\mathsf{pke}}, \mathsf{x}), \mathsf{IBE.Encrypt}(\mathsf{pp}_{\mathsf{ibe}}, (\mathsf{ssid}, \mathsf{tok}, \alpha_1), \mathsf{uid})$. In $\mathsf{Game}_3^{(i)}$, we instead have, for the first i queries to the Client Login Oracle, $\alpha_1^* \leftarrow \mathsf{PKE.Encrypt}(\mathsf{pp}_{\mathsf{pke}}, \mathsf{x}^*)$, where x^* is the credential part of the query sent to $\mathcal{F}_{\mathsf{CHL}}$, and the oracle returns $\alpha = (\alpha_1^*, \alpha_2)$ and sets $\mathcal{D}_{\mathsf{PKE}}[\alpha_1^*] \leftarrow \mathsf{x}^*$ (instead of setting $\mathcal{D}_{\mathsf{PKE}}[\alpha_1] \leftarrow \mathsf{x}$). We define $\mathsf{Game}_3^{(Q)} = \mathsf{Game}_3$.

- **Game 4:** Let Q be the total number of Client Login Oracle queries made by $\mathcal{A}$. For each $i \in [Q]$, we define $\mathsf{Game}_4^{(i)}$ to be the following: $\mathsf{Game}_4^{(i)}$ differs from Game_3 in the Client Login Oracle's construction of α: for the first i queries to the Client Login Oracle, $\alpha^* \leftarrow (\alpha_1^*, \alpha_2^*) = (\mathsf{PKE.Encrypt}(\mathsf{pp}_{\mathsf{pke}}, \mathsf{x}^*), \mathsf{IBE.Encrypt}(\mathsf{pp}_{\mathsf{ibe}}, (\mathsf{ssid}, \mathsf{tok}, \alpha_1^*), \mathsf{uid})$. In other words, we have that $\alpha^* \leftarrow \Pi_{\mathsf{fz}}.\mathsf{ClientLogin}(\mathsf{pp}, \mathsf{ssid}, \mathsf{tok}, \mathsf{uid}, \mathsf{x})$. We define $\mathsf{Game}_4^{(Q)} = \mathsf{Game}_4$.

Note that, by construction, Game_4 is identical to $\mathsf{Expt}^*_{\mathsf{real}}(\mathcal{A})$. In the following lemmas, for two experiments $E_1(\mathcal{A})$ and $E_2(\mathcal{A})$, we use the notation $E_1(\mathcal{A}) \approx E_2(\mathcal{A})$ as shorthand to represent that $|\Pr[E_1(\mathcal{A}) = 1] - \Pr[E_2(\mathcal{A}) = 1]|$ is negligible in λ.

Lemma 4. *For an adversary $\mathcal{A}$, $\mathsf{Game}_0(\mathcal{A}) \approx \mathsf{Game}_1(\mathcal{A})$ when $\mathcal{H}$ is modeled after a random oracle.*

Proof. Since the random oracle outputs of $\mathcal{H}$ are distributed uniformly at random, the only way in which an adversary can distinguish Game_0 from Game_1 is if he happens to query the Random Oracle in between a Steal Credential Verifier Oracle and Obfuscator Oracle query on an input uid and $r' \in \{0,1\}^\lambda$ for which r' is also selected in the Obfuscator Oracle query response. However, this probability is negligible in λ given that the r input is selected uniformly at random in the Obfuscator Oracle query response. □

Lemma 5. *For an adversary $\mathcal{A}$, $\mathsf{Game}_1(\mathcal{A}) \approx \mathsf{Game}_2(\mathcal{A})$, based on the security of the reusable fuzzy extractor.*

Proof. For an adversary $\mathcal{A}$, we construct a simulator $\mathcal{B}$ which simulates Game_1 and Game_2 by interacting with the challenger of $\mathsf{Expt}_b^{\mathsf{FE}}$ as follows. On each Steal Credential Verifier Oracle and Obfuscator Oracle query from $\mathcal{A}$, $\mathcal{B}$ simulates the query normally, except that when producing (r, σ), obtains the corresponding pair from the challenger of $\mathsf{Expt}_b^{\mathsf{FE}}$ instead. Note that $\mathsf{Expt}_0^{\mathsf{FE}}$ corresponds to Game_1, and $\mathsf{Expt}_1^{\mathsf{FE}}$ corresponds to Game_2. Hence, the claim follows. □

Recall by definition that $\mathsf{Game}_2 = \mathsf{Game}_3^{(0)}$.

Lemma 6. *For an adversary $\mathcal{A}$, with $i \in [Q]$, $\mathsf{Game}_3^{(i-1)}(\mathcal{A}) \approx \mathsf{Game}_3^{(i)}(\mathcal{A})$, based on the CCA security of* PKE.

Proof. For a bit b, let $\mathcal{B}$ be an adversary that participates in $\mathsf{Expt}_b^{\mathsf{CCA}}$ against a challenger. The challenger begins by computing $(\mathsf{pp}_{\mathsf{pke}}, \mathsf{sk}_{\mathsf{pke}}) \leftarrow \mathsf{PKE.Setup}(1^\lambda)$ and returns $\mathsf{pp}_{\mathsf{pke}}$ to $\mathcal{B}$. Note that for each Setup Oracle query, if $\mathsf{type} = \textsc{RevealRegKey}$ for the i^{th} Setup Oracle query, then Game $\mathsf{Game}_3^{(i-1)}$ and $\mathsf{Game}_3^{(i)}$ are identical, concluding the proof. If $\mathsf{type} = \textsc{HideRegKey}$, $\mathcal{B}$ returns $\mathsf{pp}_{\mathsf{pke}}$ to $\mathcal{A}$ as the response to the Setup Oracle query.

Then, for the i^{th} Client Login Oracle query, $\mathcal{B}$ submits two challenge messages x and x^* to the challenger, who then returns a ciphertext c. For each Server Login Oracle query, $\mathcal{B}$ makes a decryption oracle query to the challenger in order to obtain an output x' (in place of calling $\mathsf{PKE.Decrypt}$ by itself). Note that by construction, Sim only invokes $\mathsf{PKE.Decrypt}$ if the input to the Server Login oracle differs from every ciphertext produced in response to the Client Login Oracle query. Similarly, during a response to the Obfuscation Oracle, Sim only calls $\mathsf{PKE.Decrypt}$ on a ciphertext that was not produced by the Client Login Oracle query. Hence, in both types of queries, the admissibility condition of $\mathsf{Expt}_b^{\mathsf{CCA}}$ is satisfied. Since $\mathsf{Expt}_0^{\mathsf{CCA}}$ corresponds to $\mathsf{Game}_3^{(i-1)}$ and $\mathsf{Expt}_1^{\mathsf{CCA}}$ corresponds to $\mathsf{Game}_3^{(i)}$, the lemma follows. □

Recall by definition that $\mathsf{Game}_3^{(Q)} = \mathsf{Game}_3 = \mathsf{Game}_4^{(0)}$.

Lemma 7. *For an adversary $\mathcal{A}$, with $i \in [Q]$, $\mathsf{Game}_4^{(i-1)}(\mathcal{A}) \approx \mathsf{Game}_4^{(i)}(\mathcal{A})$, based on the ID-CCA security of* IBE*.*

Proof. For a bit b, let $\mathcal{B}$ be an adversary that participates in $\mathsf{Expt}_b^{\mathsf{IDCCA}}$ against a challenger. The challenger begins by computing $(\mathsf{pp}_{\mathsf{ibe}}, \mathsf{sk}_{\mathsf{ibe}}) \leftarrow \mathsf{IBE.Setup}(1^\lambda)$ and returns $\mathsf{pp}_{\mathsf{ibe}}$ to $\mathcal{B}$. Note that for each Setup Oracle query, if $\mathsf{type} = \textsc{RevealRegKey}$ for the i^{th} Setup Oracle query, then $\mathsf{Game}_4^{(i-1)}$ and $\mathsf{Game}_4^{(i)}$ are identical, concluding the proof. If $\mathsf{type} = \textsc{HideRegKey}$, $\mathcal{B}$ returns $\mathsf{pp}_{\mathsf{pke}}$ to $\mathcal{A}$ as the response to the Setup Oracle query.

Then, for the i^{th} Client Login Oracle query, $\mathcal{B}$ submits the challenge id uid^* and two challenge messages tok and tok^* to the challenger, who then returns a ciphertext c. For each Server Login Oracle query, after checking its preconditions, $\mathcal{B}$ makes a decryption oracle query to the challenger in order to obtain an output tok' (in place of calling $\mathsf{IBE.KeyGen}$ followed by $\mathsf{IBE.Decrypt}$ by itself). In order to assert the admissibility of these queries, there are two properties to check:

- $\mathsf{IBE.KeyGen}$ is called on the challenge id uid^*. Note that the only opportunity for $\mathsf{IBE.KeyGen}$ to be called is during a Steal Credential Verifier Oracle query. This must also happen *before* calling the Client Login Oracle (when the challenge messages are submitted to the challenger). In this case, note that $\mathcal{D}_{\mathsf{status}}[\mathsf{sid}, \mathsf{uid}^*]$ is set to $\textsc{Restricted}$, and there are two cases to consider based on the value of $\mathcal{C}(\mathsf{x}, \mathsf{y})$ in the Client Login Oracle response. If $\mathcal{C}(\mathsf{x}, \mathsf{y}) = 1$, then the Client Login Oracle simulation aborts, and by construction $\mathsf{Game}_4^{(i-1)}$ and $\mathsf{Game}_4^{(i)}$ are identical. If $\mathcal{C}(\mathsf{x}, \mathsf{y}) \neq 1$, then in response to the Server Login Oracle, $\mathcal{B}$ can simply set the appropriate parameters to $\perp$ instead of submitting the decryption oracle query to the challenger. Hence, in either case, we have shown that the admissibility conditions for the IBE CCA game have not been violated.

- $\mathsf{IBE.KeyGen}$ is not called on the challenge id uid^*. There are two subcases to consider based on the α component of the query sent to the Server Login Oracle. If there is already an entry of the form $(\star, \star, \star, \alpha)$ in $\mathcal{D}_{\mathsf{sent}}$, then $\mathcal{B}$ can complete the simulation by setting the appropriate parameters to $\perp$. If there is no such entry in $\mathcal{D}_{\mathsf{sent}}$, then, for $(\alpha_1, \alpha_2) = \alpha$, we need only

consider cases where α_2 previously appeared in $\mathcal{D}_{\mathsf{sent}}$. For each of these cases, the α_1 value must be such that when decrypting α_2, the output of the decryption will not match either ssid or α_1. Hence, it suffices for the simulator in these cases to set the appropriate parameters to $\perp$.

We have shown that the simulator obeys the admissibility conditions for $\mathsf{Expt}_b^{\mathsf{IDCCA}}$. Since $\mathsf{Expt}_0^{\mathsf{IDCCA}}$ corresponds to $\mathsf{Game}_4^{(i-1)}$ and $\mathsf{Expt}_1^{\mathsf{IDCCA}}$ corresponds to $\mathsf{Game}_4^{(i)}$, the lemma follows. □

Now, by tying together the above lemmas, we have shown that $\mathsf{Expt}^*_{\mathsf{ideal}}(\mathcal{A}, \mathsf{Sim})$ is indistinguishable from $\mathsf{Expt}^*_{\mathsf{real}}(\mathcal{A})$, as formalized in the below lemma.

Lemma 8. *For any adversary $\mathcal{A}$, with Sim defined as the simulator from Section 5.3, we have that $\mathsf{Expt}^*_{\mathsf{real}}(\mathcal{A}) \approx \mathsf{Expt}^*_{\mathsf{ideal}}(\mathcal{A}, \mathsf{Sim})$.*

Proof. Follows directly from applying Lemmas 4, 5, 6, and 7. □

B.1 Applying the Obfuscator

Recall from Definition 5 that in order to apply a distributional VBB obfuscator to a program $\mathbf{MBCC}[f, y, z]$, it must be the case that y is computationally unpredictable, even in the presence of f and z.

Lemma 9. *For every adversary $\mathcal{A}$, there exists a simulator $\mathcal{A}^*$ such that $\mathsf{Expt}_{\mathsf{real}}(\mathcal{A}) \approx \mathsf{Expt}^*_{\mathsf{real}}(\mathcal{A}^*)$, based on the security of the distributional VBB obfuscator and the reusable fuzzy extractor and the assumption that $\mathcal{X}$ has min-entropy at least λ.*

Proof. Defining f, y, and z to be of the form in which each of the programs obfuscated $\mathsf{Expt}_{\mathsf{ideal}}$ are of the form $\mathbf{MBCC}[f, y, z]$, note that the only term that is not truly independent of $y = \mathcal{H}(\mathsf{uid}, z)$ is σ (contained in the description of f).

However, under the assumption that $(\mathsf{Gen}, \mathsf{Rep})$ is a reusable fuzzy extractor, and that the inputs y are drawn from $\mathcal{X}$ with sufficient min-entropy, since r is produced from $\mathsf{Gen}(\mathsf{y})$, we can conclude that each r produced in this way also has entropy at least λ. This allows us to apply the distributional VBB obfuscator property as defined in Definition 5 on each obfuscation, which yields the existence of such a simulator $\mathcal{A}^*$ that satisfies the claim. □

Lemma 10. *For every adversary $\mathcal{A}$, there exists a simulator $\mathcal{A}^*$ and simulator Sim such that $\mathsf{Expt}_{\mathsf{ideal}}(\mathcal{A}, \mathsf{Sim}) \approx \mathsf{Expt}^*_{\mathsf{ideal}}(\mathcal{A}^*, \mathsf{Sim})$, based on the security of the distributional VBB obfuscator, the reusable fuzzy extractor, and the assumption that $\mathcal{X}$ has min-entropy at least λ.*

Proof. This also follows similarly to the proof of Lemma B.1 except even more straightforwardly, since the inputs s to each obfuscated function are drawn independently and uniformly at random. □

Finally, by observing that for any adversary $\mathcal{A}$, we have that there exists a simulator $\mathcal{A}^*$ and Sim for which $\mathsf{Expt}_{\mathsf{real}}(\mathcal{A}) \approx \mathsf{Expt}^*_{\mathsf{real}}(\mathcal{A}^*)$ (from Lemma B.1), $\mathsf{Expt}^*_{\mathsf{real}}(\mathcal{A}^*) \approx \mathsf{Expt}^*_{\mathsf{ideal}}(\mathcal{A}^*, \mathsf{Sim})$ (from Lemma 8), and $\mathsf{Expt}^*_{\mathsf{ideal}}(\mathcal{A}^*) \approx \mathsf{Expt}_{\mathsf{ideal}}(\mathcal{A}, \mathsf{Sim})$. Putting these together, we have that $\mathsf{Expt}_{\mathsf{real}}(adv) \approx \mathsf{Expt}_{\mathsf{ideal}}(\mathcal{A}, \mathsf{Sim})$, which concludes the proof of Theorem 2.

Session

- On (ClientLogin, sid, ssid, uid, tok) from U, if this the first ClientLogin message for (sid, ssid, uid), record ⟨Client, sid, ssid, uid, tok⟩ and mark it Fresh. send (ClientLogin, sid, ssid, uid) to $\mathcal{A}^*$.
- On (ServerLogin, sid, ssid, uid) from S, if this is the first ServerLogin message for (sid, ssid, uid), record ⟨Server, sid, ssid, uid⟩ and mark it Fresh. Send (ServerLogin, sid, ssid, uid) to $\mathcal{A}^*$.
- On (Interrupt, sid, ssid, uid) from $\mathcal{A}^*$, if this is the first Interrupt message for (sid, ssid, uid), mark ⟨Server, sid, ssid, uid⟩ as Interrupted.

Login and Authentication

- On (Login, sid, ssid, uid, tok′, flag′) from $\mathcal{A}^*$, if there is a record ⟨Server, sid, ssid, uid⟩, then mark it Completed and do the following:
  - If the client uid is corrupted, then set (tok, flag) = (tok′, flag′).
  - Otherwise if there is a record ⟨Client, sid, ssid, uid, tok⟩ marked Fresh, then mark it Completed. If the server record ⟨Server, sid, ssid, uid⟩ is Interrupted, then set (tok, flag) = (⊥, Failure). If the server record ⟨Server, sid, ssid, uid⟩ is Fresh, then don't modify tok and set flag = Success.

  Output (sid, ssid, uid, tok, flag) to S.

Figure 2: The ideal functionality $\mathcal{F}_{\mathsf{AL}}$ for Authenticated Login.

C Authenticated Login

We formally define the UC authenticated login functionality $\mathcal{F}_{\mathsf{AL}}$ in Figure 2. This is essentially a simplified version of $\mathcal{F}_{\mathsf{CHL}}$, with the removal of password/credential verification and the credential stealing interfaces. As such, it is the analogue of key exchange in the same sense as CHL is analogous to aPAKE.

Authenticated Login carries some of the distinctions that CHL has, compared to aPAKE. For example, the environment passes a token to the client and expects the same token to be output by the server, when the protocol is correctly executed. Also, an Interrupt interface is needed for authenticated login, but not in key exchange. This is because, in key exchange, if the server simulation gets a modified message, the simulator does not call NewKey, effectively aborting the session. This is the same as in the real world, where there is no explicit abort. In authenticated login, if the server simulation gets a modified message, then it must signal a login failure to the environment; this is the purpose of the Interrupt functionality. The Interrupt interface is the equivalent of the TestCred interface in CHL, except there is no low entropy credential to test.

C.1 PKI-based Authenticated Login Protocol

We describe a PKI-based protocol Π_{AL} in Figure 3, and show that it securely realizes $\mathcal{F}_{\mathsf{AL}}$. This protocol is a straightforward application of a public-key encryption PKE and signature scheme Sig to ensure confidential and authenticated transport from the client to the server. Given that the client is required to have a signature certified by a certificate authority, this protocol clearly provides redundant security guarantees when integrated within an already-authenticated TLS channel. However, Π_{AL} illustrates the basic concepts around the authenticated login primitive itself.

Setup

Sample $(\mathsf{vk}_{\mathsf{uid}}, \mathsf{sk}_{\mathsf{uid}}) \leftarrow \mathsf{Sig.Setup}(1^\lambda)$ for all the client uid's. Sample $(\mathsf{ek}, \mathsf{dk}) \leftarrow \mathsf{PKE.Setup}(1^\lambda)$ for the Server.

Session

- On $(\textsc{ClientLogin}, \mathsf{sid}, \mathsf{ssid}, \mathsf{uid}, \mathsf{tok})$ from U, if this the first $\textsc{ClientLogin}$ message for $(\mathsf{sid}, \mathsf{ssid}, \mathsf{uid})$, then sample $\mathsf{enc} \leftarrow \mathsf{PKE.Encrypt}(\mathsf{ek}, \mathsf{tok})$ and $\mathsf{sig} \leftarrow \mathsf{Sig.Sign}(\mathsf{sk}_{\mathsf{uid}}, (\mathsf{sid}, \mathsf{ssid}, enc))$, and send $(\mathsf{sid}, \mathsf{ssid}, \mathsf{uid}, \mathsf{enc}, \mathsf{sig})$ to $\mathcal{A}$.
- On $(\textsc{ServerLogin}, \mathsf{sid}, \mathsf{ssid}, \mathsf{uid})$ from S, if this is the first $\textsc{ServerLogin}$ message for $(\mathsf{sid}, \mathsf{ssid}, \mathsf{uid})$, receive $(\mathsf{sid}, \mathsf{ssid}, \mathsf{uid}, \mathsf{enc}', \mathsf{sig}')$ from $\mathcal{A}$. If $\mathsf{Sig.Verify}(\mathsf{vk}_{\mathsf{uid}}, (\mathsf{sid}, \mathsf{ssid}, \mathsf{sig}'))$ succeeds, then compute $\mathsf{tok}' = \mathsf{PKE.Decrypt}(\mathsf{dk}, \mathsf{enc}')$ and set $\mathsf{flag}' = \textsc{Success}$. Otherwise set $(\mathsf{tok}', \mathsf{flag}') = (\perp, \textsc{Failure})$. Output $(\mathsf{sid}, \mathsf{ssid}, \mathsf{uid}, \mathsf{tok}', \mathsf{flag}')$ to S.

Figure 3: PKI-based Authenticated Login protocol Π_{AL}.

Theorem 3. *The protocol Π_{AL} realizes $\mathcal{F}_{\mathsf{AL}}$, provided* PKE *satisfies IND-CPA security and* Sig *is existentially unforgeable.*

Proof. We first describe the simulator $\mathcal{A}^*$ below:

- **Setup:** The simulator generates signature key pairs $(\mathsf{vk}_{\mathsf{uid}}, \mathsf{sk}_{\mathsf{uid}})$ and encryption key pairs $(\mathsf{ek}, \mathsf{dk})$.
- **Client Login:** The simulator receives $(\textsc{ClientLogin}, \mathsf{sid}, \mathsf{ssid}, \mathsf{uid})$ from U. Sample $\mathsf{enc} \leftarrow \mathsf{PKE.Encrypt}(\mathsf{ek}, 0)$ and $\mathsf{sig} \leftarrow \mathsf{Sig.Sign}(\mathsf{sk}_{\mathsf{uid}}, (\mathsf{sid}, \mathsf{ssid}, \mathsf{enc}))$. Send $(\mathsf{sid}, \mathsf{ssid}, \mathsf{uid}, \mathsf{enc}, \mathsf{sig})$ to $\mathcal{A}$.
- **Server Login:** The simulator receives $(\textsc{ServerLogin}, \mathsf{sid}, \mathsf{ssid}, \mathsf{uid})$ from S. Then it waits to accept a message $(\mathsf{sid}, \mathsf{ssid}, \mathsf{uid}, \mathsf{enc}', \mathsf{sig}')$. If this is same as the corresponding sent message, then just call $(\textsc{Login}, \mathsf{sid}, \mathsf{ssid}, \mathsf{uid}, \perp, \perp)$. If it is not the same message that was sent, then the simulator calls $(\textsc{Interrupt}, \mathsf{sid}, \mathsf{ssid}, \mathsf{uid})$ first and then calls $(\textsc{Login}, \mathsf{sid}, \mathsf{ssid}, \mathsf{uid}, \perp, \textsc{Failure})$.

In order to prove that this simulation is indistinguishable to the adversary and the environment from the real protocol execution, note that the adversary only sees encryptions of 0, which are indistinguishable from the hybrid execution due to the semantic security of PKE. Also, the adversary cannot construct fake messages purporting to come from honest parties which pass signature verification, due to the existential unforgeability of the Sig scheme. □

C.2 Secure Channel from Authenticated Login

In this section, we provide the UC secure channel functionality $\mathcal{F}_{\mathsf{SC}}$ in Figure 4, which is adapted from [CK02]. We then present in Figure 5 the protocol Π_{SC} in the $\mathcal{F}_{\mathsf{AL}}$-hybrid model and show that it securely realizes $\mathcal{F}_{\mathsf{SC}}$.

Theorem 4. *The protocol Π_{SC} realizes $\mathcal{F}_{\mathsf{SC}}$ in the $\mathcal{F}_{\mathsf{AL}}$-hybrid model, provided that* PKE *satisfies IND-CCA security and* MAC *is unforgeable.*

Proof. We describe the simulator steps as follows:

- Upon receiving $\mathcal{F}_{\mathsf{SC}}.(\textsc{NewSession}, \mathsf{sid}, \mathsf{U}, \mathsf{S}, \textsc{initiator})$ from U, the simulator samples encryption and mac keys $\mathsf{tok} = (\mathsf{k_e}, \mathsf{k_a})$ and calls $\mathcal{F}_{\mathsf{AL}}$ with $(\textsc{ClientLogin}, \mathsf{sid}, \mathsf{U}, \mathsf{tok})$.

$\mathcal{F}_{\mathsf{SC}}$ proceeds as follows, running with parties U and S and $\mathcal{A}^*$:

- On (NEWSESSION, $\mathsf{sid}, \mathsf{U}, \mathsf{S}$, INITIATOR) from S, send (NEWSESSION, $\mathsf{sid}, \mathsf{U}, \mathsf{S}$, INITIATOR) to $\mathcal{A}^*$, and wait to receive a value (NEWSESSION, $\mathsf{sid}, \mathsf{S}, \mathsf{U}$, RESPONDER) from U. Once this value is received, set a boolean variable active, Say that U and S are the partners of this session. Send (NEWSESSION, $\mathsf{sid}, \mathsf{S}, \mathsf{U}$, RESPONDER) to $\mathcal{A}^*$.
- On (SEND, $\mathsf{sid}, \mathsf{U}, \mathsf{S}, m$) from U, and if active is set, send (SEND, $\mathsf{sid}, \mathsf{U}, \mathsf{S}, |m|, t$) to the adversary, where t is a counter initialized to 0 and incremented by 1 at every send. Record ($\mathsf{sid}, \mathsf{U}, \mathsf{S}, m, t$).
- On (DELIVER, $\mathsf{sid}, \mathsf{U}, \mathsf{S}, m, t$) from $\mathcal{A}^*$: if S is corrupted, then output (RECEIVED, $\mathsf{sid}, \mathsf{U}, \mathsf{S}, m, t$). Otherwise, lookup record ($\mathsf{sid}, \mathsf{U}, \mathsf{S}, m', t$) and output (RECEIVED, $\mathsf{sid}, \mathsf{U}, \mathsf{S}, m', t$).
- The sends from S to U are handled analogously.

Figure 4: The ideal functionality $\mathcal{F}_{\mathsf{SC}}$ for Secure-Channel

- On $\mathcal{F}_{\mathsf{SC}}$.(NEWSESSION, $\mathsf{sid}, \mathsf{U}, \mathsf{S}$, INITIATOR) from U, sample encryption and mac keys $\mathsf{tok} = (\mathsf{k_e}, \mathsf{k_a})$ and call $\mathcal{F}_{\mathsf{AL}}$ with (CLIENTLOGIN, $\mathsf{sid}, \mathsf{U}, \mathsf{tok}$).
- On $\mathcal{F}_{\mathsf{SC}}$.(NEWSESSION, $\mathsf{sid}, \mathsf{S}, \mathsf{U}$, RESPONDER) from S, call $\mathcal{F}_{\mathsf{AL}}$ with (SERVERLOGIN, sid, U).
- On ($\mathsf{sid}, \mathsf{uid}, \mathsf{tok}', \mathsf{flag}$) from $\mathcal{F}_{\mathsf{AL}}$.LOGIN: If $\mathsf{flag} =$ SUCCESS, then parse tok' as $(\mathsf{k}'_{\mathsf{e}}, \mathsf{k}'_{\mathsf{a}})$.
- On $\mathcal{F}_{\mathsf{SC}}$.(SEND, $\mathsf{sid}, \mathsf{U}, \mathsf{S}, m$) from U, encrypt and mac (t, m) with counter t which is initialized to 0 and incremented by 1 for each send, using $(\mathsf{k_e}, \mathsf{k_a})$ and sends (DELIVER, $\mathsf{sid}, \mathsf{S}, \mathsf{U}, t, c$) to $\mathcal{A}^*$.
- Upon receiving (DELIVER, $\mathsf{sid}, \mathsf{S}, \mathsf{U}, t, c'$) from $\mathcal{A}^*$ use the corresponding key pair $(\mathsf{k_e}, \mathsf{k_a})$ for $(\mathsf{sid}, \mathsf{S}, \mathsf{U})$, checks authenticity and decrypts to m' and outputs $\mathcal{F}_{\mathsf{SC}}$.(RECEIVED, $\mathsf{sid}, \mathsf{S}, \mathsf{U}, t, m'$).
- The sends from S to U are handled analogously.

Figure 5: Protocol Π_{SC} to realize $\mathcal{F}_{\mathsf{SC}}$ in the $\mathcal{F}_{\mathsf{AL}}$-hybrid model.

- Upon receiving $\mathcal{F}_{\mathsf{SC}}.(\textsc{NewSession}, \mathsf{sid}, \mathsf{S}, \mathsf{U}, \textsc{responder})$ from S, the simulator calls $\mathcal{F}_{\mathsf{AL}}$ with $(\textsc{ServerLogin}, \mathsf{sid}, \mathsf{U})$.
- Upon getting output $(\mathsf{sid}, \mathsf{uid}, \mathsf{tok}', \mathsf{flag})$ from $\mathcal{F}_{\mathsf{AL}}.\textsc{Login}$, if $\mathsf{flag} = \textsc{Success}$, then parse tok' as $(\mathsf{k}'_{\mathsf{e}}, \mathsf{k}'_{\mathsf{a}})$.
- Upon receiving $\mathcal{F}_{\mathsf{SC}}.(\textsc{Send}, \mathsf{sid}, \mathsf{U}, \mathsf{S}, t, |m|)$ from U, the simulator encrypts and macs $(t, 0^{|m|})$ with counter t which is initialized to 0 and incremented by 1 for each send, using $(\mathsf{k}_{\mathsf{e}}, \mathsf{k}_{\mathsf{a}})$ and sends to $\mathcal{A}$.
- Upon receiving $\mathcal{F}_{\mathsf{SC}}.(\mathsf{sid}, \mathsf{S}, \mathsf{U}, t, c')$ from $\mathcal{A}$, the simulator checks if it sent c' for $(\mathsf{sid}, \mathsf{S}, \mathsf{U}, t)$. If so, then it calls $\mathcal{F}_{\mathsf{SC}}$ with $(\textsc{Received}, \mathsf{sid}, \mathsf{S}, \mathsf{U}, t, \perp)$. This has the effect of allowing $\mathcal{F}_{\mathsf{SC}}$ to output the recorded message and completely ignore c'. Otherwise, the simulator looks up for a recorded key pair $(\mathsf{k}_{\mathsf{e}}, \mathsf{k}_{\mathsf{a}})$ for $(\mathsf{sid}, \mathsf{S}, \mathsf{U})$, checks authenticity and decrypts to m' and sends $\mathcal{F}_{\mathsf{SC}}.(\textsc{Received}, \mathsf{sid}, \mathsf{S}, \mathsf{U}, t, m')$. Observe that the adversary can modify the message while passing authenticity only if it has corrupted the party or compromised the session in some way.

The sends from S to U are handled analogously.

Indistinguishability follows from the fact that the adversary only sees encryptions of 0, which are indistinguishable from the hybrid execution due to the CCA security of PKE. Also, the adversary is unable to construct fake messages purporting to come from honest parties which pass MAC verification, due to the unforgeability of the MAC scheme. □

D UC aPAKE Definition

We reproduce the UC functionality for Asymmetric PAKEs from [JKX18], split between Figures 6 and 7. More details can be found in the original paper [GMR06].

In the description below, we assume $P \in \{\mathsf{U}, \mathsf{S}\}$.

Password Registration

- On $(\textsc{StorePwdFile}, \mathit{sid}, \mathsf{U}, \textsc{pw})$ from S, if this is the first StorePwdFile message, record $\langle \textsc{File}, \mathsf{U}, \mathsf{S}, \textsc{pw} \rangle$ and mark it Uncompromised.

Stealing Password Data

- On $(\textsc{StealPwdFile}, \mathsf{sid})$ from $\mathcal{A}^*$, if there is no record $\langle \textsc{File}, \mathsf{U}, \mathsf{S}, \textsc{pw} \rangle$, return "no password file" to $\mathcal{A}^*$. Otherwise, if the record is marked Uncompromised, mark it Compromised; regardless,
  - If there is a record $\langle \textsc{Offline}, \textsc{pw} \rangle$, send $\textsc{pw}$ to $\mathcal{A}^*$.
  - Else, return "password file stolen" to $\mathcal{A}^*$.
- On $(\textsc{OfflineTestPwd}, \mathsf{sid}, \textsc{pw}^*)$ from $\mathcal{A}^*$, do:
  - If there is a record $\langle \textsc{File}, \mathsf{U}, \mathsf{S}, \textsc{pw} \rangle$ marked Compromised, do: if $\textsc{pw}^* = \textsc{pw}$, return "correct guess" to $\mathcal{A}^*$; else return "wrong guess".
  - Else record $\langle \textsc{Offline}, \textsc{pw} \rangle$.

Password Authentication

- On $(\textsc{UsrSession}, \mathsf{sid}, \mathsf{ssid}, \mathsf{S}, \textsc{pw}')$ from U, send $(\textsc{UsrSession}, \mathsf{sid}, \mathsf{ssid}, \mathsf{U}, \mathsf{S})$ to $\mathcal{A}^*$. Also, if this is the first UsrSession message for ssid, record $\langle \mathsf{ssid}, \mathsf{U}, \mathsf{S}, \textsc{pw}' \rangle$ and mark it Fresh.
- On $(\textsc{SvrSession}, \mathsf{sid}, \mathsf{ssid})$ from S, retrieve $\langle \textsc{File}, \mathsf{U}, \mathsf{S}, \textsc{pw} \rangle$, and send $(\textsc{SvrSession}, \mathsf{sid}, \mathsf{ssid}, \mathsf{U}, \mathsf{S})$ to $\mathcal{A}^*$. Also, if this is the first SvrSession message for ssid, record $\langle \mathsf{ssid}, \mathsf{S}, \mathsf{U}, \textsc{pw} \rangle$ and mark it Fresh.

Figure 6: The ideal functionality for aPAKE (part 1 of 2).

Active Session Attacks

- On $(\textsc{TestPwd}, \mathsf{sid}, \mathsf{ssid}, \mathsf{P}, \textsc{pw}^*)$ from $\mathcal{A}^*$, if there is a record $\langle \mathsf{ssid}, \mathsf{P}, \mathsf{P}', \textsc{pw}' \rangle$ marked Fresh, do: if $\textsc{pw}^* = \textsc{pw}'$, mark it Compromised and return "correct guess" to $\mathcal{A}^*$; else mark it Interrupted and return "wrong guess".
- On $(\textsc{Impersonate}, \mathsf{sid}, \mathsf{ssid})$ from $\mathcal{A}^*$, if there is a record $\langle \mathsf{ssid}, \mathsf{U}, \mathsf{S}, \textsc{pw}' \rangle$ marked Fresh, do: if there is a record $\langle \textsc{File}, \mathsf{U}, \mathsf{S}, \textsc{pw} \rangle$ marked Compromised and $\textsc{pw}' = \textsc{pw}$, mark $\langle \mathsf{ssid}, \mathsf{U}, \mathsf{S}, \textsc{pw}' \rangle$ Compromised and return "correct guess" to $\mathcal{A}^*$; else mark it Interrupted and return "wrong guess".

Key Generation and Authentication

- On $(\textsc{NewKey}, \mathsf{sid}, \mathsf{ssid}, \mathsf{P}, SK^*)$ from $\mathcal{A}^*$ where $|SK^*| = \ell$, if there is a record $\langle \mathsf{ssid}, \mathsf{P}, \mathsf{P}', \textsc{pw}' \rangle$ not marked Completed, do:
  - If the record is Compromised, or P or P' is corrupted, set $SK := SK^*$.
  - Else if the record is Fresh, a $(\mathsf{sid}, \mathsf{ssid}, SK')$ tuple was sent to P', and at that time there was a record $(\mathsf{ssid}, \mathsf{P}', \mathsf{P})$ marked Fresh, set $SK := SK'$.
  - Else pick $SK \leftarrow_{\mathsf{R}} \{0,1\}^\ell$.

  Finally, mark $\langle \mathsf{ssid}, \mathsf{P}, \mathsf{P}', \textsc{pw}' \rangle$ Completed and send $(\mathsf{sid}, \mathsf{ssid}, SK)$ to P.
- On $(\textsc{TestAbort}, \mathsf{sid}, \mathsf{ssid}, \mathsf{P})$ from $\mathcal{A}^*$, if there is a record $\langle \mathsf{ssid}, \mathsf{P}, \mathsf{P}', \textsc{pw}' \rangle$ not marked Completed, do:
  - If it is Fresh and there is a record $\langle \mathsf{ssid}, \mathsf{P}', \mathsf{P}, \textsc{pw}' \rangle$, send Succ to $\mathcal{A}^*$.
  - Else send Fail to $\mathcal{A}^*$ and $(\textsc{Abort}, \mathsf{sid}, \mathsf{ssid})$ to P, and mark $\langle \mathsf{ssid}, \mathsf{P}, \mathsf{P}', \textsc{pw}' \rangle$ Completed.

Figure 7: The ideal functionality for aPAKE (part 2 of 2).