Shared-Custodial Password-Authenticated Deterministic Wallets[⋆]

Poulami Das[1], Andreas Erwig[2], and Sebastian Faust[2]

[1] CISPA Helmholtz Center for Information Security
[2] TU Darmstadt
`poulami.das@cispa.de`
`{firstname.lastname}@tu-darmstadt.de`

Abstract. Cryptographic wallets are an essential tool in Blockchain networks to ensure the secure storage and maintenance of an user's cryptographic keys. Broadly, wallets can be divided into three categories, namely custodial, non-custodial, and shared-custodial wallets. The first two are centralized solutions, i.e., the wallet is operated by a single entity, which inherently introduces a single point of failure. Shared-custodial wallets, on the other hand, are maintained by two independent parties, e.g., the wallet user and a service provider, and hence avoid the single point of failure centralized solutions. Unfortunately, current shared-custodial wallets suffer from significant privacy issues.

In our work, we introduce *password-authenticated deterministic wallets* (PADW), a novel and efficient shared-custodial wallet solution, which exhibits strong security and privacy guarantees. In a nutshell, in a PADW scheme, the secret key of the user is shared between the user and the server. In order to generate a signature, the user first authenticates itself to the server by providing a password and afterwards engages in an interactive signing protocol with the server. Security is guaranteed as long as at most one of the two parties is corrupted. Privacy, on the other hand, guarantees that a corrupted server cannot link a transaction to a particular user. We formally model the notion of PADW schemes and we give an instantiation from blind Schnorr signatures. Our construction allows for deterministic key derivation, a feature that is widely used in practice by existing wallet schemes, and it does not rely on any heavy cryptographic primitives. We prove our scheme secure against adaptive adversaries in the random oracle model and under standard assumptions. That is, our security proof only relies on the assumption that the Schnorr signature scheme is unforgeable and that a public key encryption scheme is CCA-secure.

1 Introduction

Blockchain technology has gained increasing popularity throughout the past decade. Arguably its most prominent application are cryptocurrencies such as Bitcoin and Ethereum, which allow to realize financial products in a decentralized way without having to rely on a central authority or a financial institution. The main cryptographic building block in virtually all cryptocurrencies is a digital signature scheme which allows users to authenticate transactions. More concretely, if Alice wants to pay c coins to Bob, she first generates a transaction of the form "pk_A pays c coins to pk_B" where pk_A and pk_B denote the signing public keys of Alice and Bob, respectively. Alice then generates a digital signature on the transaction using her signing secret key sk_A and posts the transaction/signature pair to the Blockchain. Essentially, Alice's funds in the network are tied to her signing secret key such that any party that knows sk_A can spend Alice's funds. This crucially requires that users in a Blockchain network securely store their signing keys and protect them from attackers.

In the cryptocurrency space, the storage and maintenance of cryptographic keys is typically handled by so-called *wallets* which can be broadly divided into three categories. *Custodial wallets* are essentially service providers such as Coinbase[3] or BitGo[4] which generate and maintain cryptographic keys on behalf of the users, i.e., the keys are stored and used *only* by the service provider. While this is a convenient solution from a usability perspective, it requires users to fully trust the service provider to store their

[⋆] This is the full version of a paper that appears in the proceedings of the 14th International Conference on Security and Cryptography for Networks 2024 (SCN 2024).
[3] `https://custody.coinbase.com/`
[4] `https://www.bitgo.com/`

keys securely and to prevent any misuse of the users' funds. The recent bankruptcy of the cryptocurrency exchange FTX, which offered custodial wallets to its customers, exemplifies this risk as it left many users unable to withdraw their money. The subsequent financial damages are likely to be in the billions of USD [5, 17, 23]. Furthermore, since such service providers hold the funds of many users in one centralized place, they naturally pose an attractive target for attackers as illustrated by many striking attacks on various custodial wallets (e.g., [39, 11]). Besides the risk of attacks and fund misuse, custodial wallets typically also do not guarantee any privacy for their users. That is, the service provider typically learns every single transaction that a user issues, thus allowing for censorship or financial attacks such as front running. Several companies such as Fireblocks[5] or Sepior[6] provide distributed custodial wallets where the keys of users are distributed across several devices, all operated by the service provider. While this reduces the risk of full key compromise, it still suffers from the above trust and privacy issues.

Non-custodial wallets, on the other hand, are operated solely by the user, i.e., the user is responsible for the secure generation and storage of its keys. This naturally avoids the trust and privacy issues of custodial wallets by giving the user full control over its own keys. Prominent examples of non-custodial wallets include so-called hardware wallets (e.g., [43, 42]), where the keys are stored on a special purpose device that stays offline most of the time, or software wallets, which are executed on the user's mobile device or computer. In order to operate such wallets securely, the user must ensure that the respective device on which the wallet runs is well protected from attacks. Unfortunately, however, most users may not have the necessary expertise or infrastructure to shield their funds from attacks. Indeed, in the past there have been attacks on both, hardware and software wallets (e.g. [12, 40]).

A compelling middle ground between custodial and non-custodial wallets are *shared-custodial wallets*, which are jointly maintained by the user and a service provider. That is, the user secret-shares its key with a service provider such that no single party knows the entire secret key. The idea is then that the wallet should remain secure as long as at most one of the user or service provider is corrupted, which essentially removes the single point of failure of (non-)custodial wallets. At the same time, however, the user should be in control of its funds at all times, i.e., only the user should be able to decide which transactions to sign at what time. The service provider mainly serves as a safeguard to ensure that the user's funds remain secure even in case the user device gets compromised. Naturally, this setting requires some sort of authentication mechanism through which the user can authenticate itself to the service provider in order to prevent impersonation attacks, where an attacker tries to obtain access to the service provider by impersonating the user. In practice, this is typically achieved via password-based authentication, which is also prominently used by custodial wallets. Lastly, it is important that a shared-custodial wallet offers strong privacy guarantees for the user. At a minimum, we need to guarantee that the service provider does not learn the transactions a user wishes to sign as otherwise the service provider may censor or front-run transactions. Ideally, we would like to additionally guarantee that a service provider does not even learn the signature that is being generated or the public key under which the signature can be verified. These additional properties would allow that the service provider, upon observing a message/signature/public key tuple on the blockchain, could not tell whether it was involved in the signature generation or not. Unfortunately, while several companies are currently exploring shared-custodial wallets (e.g., [47]), but none of them offers privacy guarantees for the user.

Deterministic Wallets. Regardless of whether a wallet is custodial, non-custodial, or shared-custodial, a popular feature to achieve some level of privacy for cryptographic wallets is the concept of deterministic wallets, which is indeed widely followed in practice today (e.g., by Electrum[7], Ledger[8], Trezor[9]). In many cryptocurrency networks, payments of a user can easily be linked if the user uses the same key pair for all of its payments. It is therefore generally recommended to use different key pairs for each transaction. Trivially, users could simply generate a fresh key pair for each payment, which however would result in a large number of keys that a user must store and maintain.

A deterministic wallet, however, addresses this issue of linkable transactions in a more storage efficient manner. At a high level, a deterministic wallet is initialized with a signing key pair $(\mathsf{pk}, \mathsf{sk})$ and a random

[5] `https://www.fireblocks.com/`
[6] `https://www.sepior.com/`
[7] `https://electrum.org/`
[8] `https://www.ledger.com/`
[9] `https://trezor.io/`

string seed and uses two deterministic key derivation algorithms. These algorithms allow to derive so-called (one-time) session key pairs from $(\mathsf{pk}, \mathsf{sk})$ and the seed such that the derived session public keys are unlinkable. This allows the wallet to only store the initial key pair $(\mathsf{pk}, \mathsf{sk})$ and seed while being able to derive arbitrary session keys on the fly. We note that it is typically easy to operate (non-)custodial wallets in the deterministic setting, as the public and secret keys are stored on a single device. For shared-custodial wallets, however, implementing the secret key derivation algorithm is typically more challenging because the secret key is distributed among several devices.

1.1 Our Contribution

In this work, we develop a novel shared-custodial wallet that achieves the following three goals: (1) it offers strong security guarantees, where as long as only one of the parties is malicious the funds of the user remain secure; (2) it guarantees privacy against a malicious service provider; and (3) it supports deterministic key derivation and is therefore compatible with state of the art deterministic wallets that are widely used in practice. To achieve these goals, we introduce the notion of *password-authenticated deterministic wallets* (PADW), which at a high level works as follows. We consider two parties, a user and a server, where each stores a share of the signing secret key, s.t. none can generate a signature without the other. When the user wishes to sign a transaction, it can send a signing request to the server, upon which both parties engage in an interactive signing protocol. The scheme is password-authenticated, i.e., the user initially registers a low-entropy password with the server and uses this password later to authenticate itself during each signing request. The server executes the signing protocol with the user only if the authentication is successful.

A PADW scheme exhibits strong security and privacy guarantees. That is, it guarantees security against a malicious server or user respectively, i.e., the funds of the user remain secure as long as at most one of the two parties is corrupted. In addition, a PADW scheme offers privacy for the user such that a malicious server cannot link a signed transaction to a particular user. This means that a malicious server does not obtain any information about the transaction, signature, or session public key during and after the execution of a signing protocol execution. In the following, we explain our PADW primitive in more detail.

Password-Authentication. As mentioned above, a PADW scheme must be secure against a malicious user or server respectively. This means that upon an adversary corrupting either of the two parties, the adversary should not be able to forge a signature. If the PADW scheme was not password-authenticated, this security notion can trivially be attacked in the following way: recall that the user should be in full control of its funds, i.e., it can decide when a signature must be generated. Therefore, when an adversary corrupts the device of the user, the adversary could simply send a signing request to the server and thereby receive valid signatures for arbitrary messages. The password-authentication prevents this attack since the adversary must now not only corrupt the user's device but also know the correct password to send a valid signing request. Naturally, we must prevent the adversary from learning the password with higher probability than simply guessing it. Therefore, it is crucial to prevent so-called offline attacks, where the adversary can essentially brute-force the (low-entropy) password locally and therefore break the security of the scheme. Indeed, the reason why we require the user to have a secret key share *and* a password (as opposed to just a password), is that it is inherently difficult to prevent offline attacks if the user derives its secret key share from the password. The reason for this is that an adversary corrupting the server could simply locally brute-force all potential user secret key shares and check for each candidate if the reconstruction with the server secret key share yields the scheme's correct secret key. Instead, we want to limit the adversary to online attacks, where it can only guess the password by sending a signing request to the server. Such online attacks are significantly weaker since the (honest) server can limit the amount of password guesses.

Adaptive Security. We prove our construction secure with respect to an adaptive adversary, i.e., an adversary that can corrupt parties during the execution of a protocol run. This is a significantly stronger adversarial model than assuming a static adversary that may corrupt a party only before the execution of a protocol. Indeed, adaptive security is particularly relevant in the password-authenticated setting due to the following scenario that can occur in the adaptive but not in the static setting: recall that in our PADW scheme the user first authenticates to the server and, if the authentication succeeds, the user and server jointly generate a signature. An adaptive adversary may corrupt the user right after the authentication step, but before the actual signature generation. In that case, the adversary circumvents the authentication and can send a signing request for an arbitrary message.

In our construction, we must therefore ensure that even in the above scenario the adversary cannot forge a signature for an arbitrary message. We do so by (1) letting the (honest) user fix the message that it wants to sign even before running the authentication process with the server, and (2) making sure that the adversary does not learn any information about the password even if it corrupts the user right before or right after the authentication.

Instantiation. We provide an instantiation of a PADW scheme from the Schnorr signature scheme [36] which is used by many major Blockchain networks such as Bitcoin [45], Bitcoin Cash, Litecoin and Polkadot. In order to achieve the strong privacy guarantees of a PADW scheme (goal (2)), we rely on the blind signature version of Schnorr signatures [10]. A blind signature scheme allows two parties, a user and a server, to engage in a protocol where the server's input is a signing secret key sk and the user's input is a message m. The protocol outputs a signature on m under sk to the user and guarantees that the server neither learns the signature nor the message m. We adjust the original blind Schnorr scheme in the following way. First, we include a password-authentication mechanism to ensure that only authenticated users can receive a valid signature from the server. Second, we extend it to a setting where the signing secret key is shared between the user and server. Both of these adaptations are crucial to achieve the strong security guarantees of a PADW scheme (goal (1)). Finally, we introduce key derivation algorithms that allow the user to deterministically derive session keys (goal (3)). Finally, we summarize our contributions as follows:

1. As a first step, we formally model the notion of *password-authenticated deterministic wallets* (PADW). In particular, our model considers an adaptive adversary which may corrupt parties at any point during a protocol execution which is particularly relevant for the password-authenticated setting.
2. We then provide a construction of a PADW scheme, which essentially extends the blind Schnorr signature scheme by (1) a password authentication mechanism, (2) a shared signing protocol, where server and user each have a share of the secret key, and (3) a deterministic key derivation algorithm, which allows to provide strong privacy guarantees.
3. Finally, we provide a formal security proof of our construction and we discuss potential extensions of our scheme.

1.2 Related Work

Several previous works (e.g., [20, 25, 32, 46]) considered the setting, where a user and a server jointly generate a signature if the user knows a correct password. We focus here only on works that achieve some form of blindness. Gjøsteen and Thuen [22, 21] present the notion of (partially blind) password-based signatures where the user secret key share is derived directly from the user's password. This allows the server to launch an offline attack on the password making their scheme insecure against a malicious server. Camenisch et al. [8] propose the notion of a password-authenticated server-aided signature scheme which, similarly to our scheme, is secure against corruption of the server and user respectively. However, their construction is based on the RSA signature scheme, which is not compatible with any major Blockchain network. In addition, their construction achieves only a weak notion of blindness, which is not sufficient for our deterministic wallet setting. None of the above works considered the setting of shared-custodial wallets with deterministic key derivation, which is our main focus.

Another line of work considers the primitive of password-protected secret sharing (e.g.,[3, 26, 7]), where a user shares its secret to several servers in such a way that the secret can only be reconstructed given a specific password. However, this primitive does not allow for any distributed computation on the secret shares, but rather requires to first reconstruct the secret before it can be used for any computation. This is in contrast to our setting, where we require that the signing secret key remains shared during the signing protocol and the user only obtains signatures instead of the entire key. Similarly to password-protected secret sharing, Chase et al. [9] recently proposed a custodial secret storage solution, where a user can share a secret to several servers and choose a policy for the reconstruction. In contrast to our setting, however, the secret reconstruction happens again on the user's device.

Deterministic wallets have been extensively studied in the past years (e.g., [14, 15, 24, 31, 2, 16]), but no prior work considered deterministic wallets in the password-authenticated setting. Kondi et al. [29]

consider a threshold wallet solution where the signing key can be proactively refreshed even with some protocol participants being offline. However, they do not consider the password-authenticated or deterministic wallet setting. The work of Marcedone et al. [34] formally analyzes hardware wallets in the following setting: A user knows only a low-entropy password and uses a hardware wallet, which stores a high entropy secret key, to jointly generate signatures. Unfortunately, their construction is prone to offline attacks, i.e., a corrupted hardware wallet can brute-force the user's password and thereby forge signatures. In addition, Marcedone et al. do not consider the deterministic wallet setting.

Finally, the blind Schnorr signature scheme has been extensively studied in the past (e.g. [19, 38, 27, 18]), in particular with respect to its concurrent security. Schnorr [37] introduced the ROS problem and showed that solving it leads to an attack against the concurrent security of blind Schnorr. Later, Wagner [41] showed that the ROS problem can be solved in subexponential time and recently, Benhamouda et al. [4] presented the first ROS attack running in polynomial time. However, since we do not consider concurrent security in our work, our solution is not affected by the ROS attack.

2 Preliminaries

2.1 Notation

For an integer $l > 0$, we use $[l]$ to denote the set of integers $\{1, \cdots, l\}$ and we use the notation $s \xleftarrow{\$} H$ to denote the uniform sampling of a variable s from a set H. For two strings $a, b \in \{0,1\}^*$, we write $a = (b, \cdot)$ if a is prefixed by b. For an algorithm A, we use the notation $y \leftarrow A(x)$ to denote the execution of A on input x that outputs y. We denote by $y \in A(x)$ that y is an element in the set of all possible outputs of an execution of A on input x. If A and B are two interactive algorithms, we use the notation $(a, b) \leftarrow \langle A(x), B(y) \rangle$ to denote the joint execution of A and B on inputs x and y respectively and with outputs a for A and b for B. Throughout this paper, we denote by κ the security parameter and we abbreviate the terms *probabilistic polynomial time* and *deterministic polynomial time* by PPT and DPT respectively.

2.2 Adversary and Communication Model

In our work, we assume an adaptive adversary that, upon corruption of a party, obtains full control over the party's device and learns its entire internal state. Corruption can occur before, after, or even during the execution of a protocol, with the only restriction that corruption may not occur while a party is executing a local computation. More specifically, we assume that corruption during a protocol execution can occur at any point of interaction, but not during local computation. This is a common and reasonable model since local computation typically takes little time in comparison to interaction, where messages have to be sent over the network to another party. We assume reliable erasures, i.e., we assume that parties can reliably erase their internal state. This is a common assumption to prove adaptive security of protocols and in particular, it is a necessary assumption in our password-authenticated setting since the user device must be able to reliably erase the password again after successful authentication. Otherwise, an adversary corrupting the user's device can simply learn the user's secret key share and password, which would essentially allow to forge signatures for arbitrary messages. In our work, we assume the existence of authenticated channels, i.e., when an honest server and an honest user communicate, the adversary cannot read or tamper with the messages.

2.3 Digital Signatures and Public Key Encryption

Definition 1 (Digital signatures). *A digital signature scheme* Sig *is defined w.r.t. a message space* $\mathcal{M}$ *and consists of a triple of algorithms* $\mathsf{Sig} = (\mathsf{KGen}, \mathsf{Sign}, \mathsf{Verify})$ *which are defined as follows: The probabilistic key generation algorithm* KGen *takes as input a security parameter* κ *and outputs a key pair* $(\mathsf{pk}, \mathsf{sk})$*. The probabilistic signing algorithm* Sign *takes as input a secret key* sk *and message* $m \in \mathcal{M}$ *and outputs a signature* σ*. The deterministic verification algorithm* Verify *takes as input a public key* pk*, a message* $m \in \mathcal{M}$*, and a signature* σ *and outputs a bit* $b \in \{0,1\}$*. If the output is* 1*,* σ *is called a valid signature.*

Correctness. *For all* $\kappa \in \mathbb{N}$, *all* $(\mathsf{pk}, \mathsf{sk}) \leftarrow \mathsf{KGen}(1^\kappa)$ *and all* $m \in \mathcal{M}$ *it must hold that*

$$\Pr\left[\mathsf{Verify}(\mathsf{pk}, m, \mathsf{Sign}(\mathsf{sk}, m)) = 1\right] = 1.$$

Definition 2 (Unforgeability of digital signature schemes). *A digital signature scheme* $\mathsf{Sig} = (\mathsf{KGen}, \mathsf{Sign}, \mathsf{Verify})$ *is unforgeable if for any PPT adversary* $\mathcal{A}$ *the following holds* $\Pr[\mathbf{uf\text{-}cma}^{\mathcal{A}}_{\mathsf{Sig}}(1^\kappa) = 1] \leq \mathsf{negl}(\kappa)$, *where* negl *is a negligible function in the security parameter* κ *and the game* $\mathbf{uf\text{-}cma}_{\mathsf{Sig}}$ *is defined below.*

Game $\mathbf{uf\text{-}cma}^{\mathcal{A}}_{\mathsf{Sig}}(1^\kappa)$

- The game generates $(\mathsf{pk}, \mathsf{sk}) \leftarrow \mathsf{Sig.KGen}(1^\kappa)$ and initiates a list $\mathsf{SigList} := \emptyset$. It forwards pk to $\mathcal{A}$.
- $\mathcal{A}$ receives access to a signing oracle, which on input a message m first adds m to $\mathsf{SigList}$, i.e., $\mathsf{SigList} \leftarrow \mathsf{SigList} \cup \{m\}$ and then outputs $\sigma \xleftarrow{\$} \mathsf{Sig.Sign}(\mathsf{sk}, m)$.
- Eventually, the adversary outputs a forgery (σ^*, m^*) and wins the game if (1) it holds that $\mathsf{Sig.Verify}(\mathsf{pk}, m^*, \sigma^*) = 1$, and (2) $m^* \notin \mathsf{SigList}$.

Since our solution relies on the (blind) Schnorr signature scheme, we recall in Appendix A the Schnorr signature scheme and the blind Schnorr signature scheme.

Definition 3 (Public Key Encryption). *A public key encryption scheme* PKE *is defined w.r.t. a message space* $\mathcal{M}$ *and consists of a triple of algorithms* $\mathsf{PKE} = (\mathsf{KGen}, \mathsf{Enc}, \mathsf{Dec})$ *which are defined as follows: The probabilistic key generation algorithm* KGen *takes as input a security parameter* κ *and outputs a key pair* $(\mathsf{pk}, \mathsf{sk})$. *The probabilistic encryption algorithm* Enc *takes as input a public key* pk *and a message* $m \in \mathcal{M}$ *and outputs a ciphertext* ct. *The deterministic decryption algorithm* Dec *takes as input a secret key* sk *and a ciphertext* ct *and outputs either* $\perp$ *or a message* $m \in \mathcal{M}$.

Correctness. *For all* $\kappa \in \mathbb{N}$, *all* $(\mathsf{pk}, \mathsf{sk}) \leftarrow \mathsf{KGen}(1^\kappa)$ *and all* $m \in \mathcal{M}$ *it must hold that*

$$\Pr\left[\mathsf{Dec}(\mathsf{sk}, \mathsf{Enc}(\mathsf{pk}, m)) = 1\right] = 1.$$

Definition 4 (IND-CCA-security of public key encryption schemes). *A public key encryption scheme* $\mathsf{PKE} = (\mathsf{KGen}, \mathsf{Enc}, \mathsf{Dec})$ *is* **IND-CCA**-*secure if for any PPT adversary* $\mathcal{A}$ *the following holds:*

$$\Pr[\mathbf{IND\text{-}CCA}^{\mathcal{A}}_{\mathsf{PKE}}(1^\kappa) = 1] \leq \frac{1}{2} + \mathsf{negl}(\kappa),$$

where negl *is a negligible function in the security parameter* κ *and the game* $\mathbf{IND\text{-}CCA}_{\mathsf{PKE}}$ *is defined below. We define* $\mathcal{A}$*'s advantage in game* $\mathbf{IND\text{-}CCA}^{\mathcal{A}}_{\mathsf{PKE}}$ *as* $\mathsf{Adv}^{\mathcal{A}}_{\mathbf{IND\text{-}CCA}_{\mathsf{PKE}}} := \Pr[\mathbf{IND\text{-}CCA}^{\mathcal{A}}_{\mathsf{PKE}}(1^\kappa) = 1] - \frac{1}{2}$.

Game $\mathbf{IND\text{-}CCA}^{\mathcal{A}}_{\mathsf{PKE}}(1^\kappa)$

- The game generates $(\mathsf{pk}, \mathsf{sk}) \leftarrow \mathsf{PKE.KGen}(1^\kappa)$ and forwards the public key pk to $\mathcal{A}$.
- $\mathcal{A}$ receives access to a decryption oracle, which on input a ciphertext ct, outputs $\mathsf{PKE.Dec}(\mathsf{sk}, \mathsf{ct})$.
- Eventually, the adversary outputs two messages (m_0, m_1), upon which the game checks if $m_0 \in \mathcal{M}$ and $m_1 \in \mathcal{M}$. If any of the checks fails, the game returns $\perp$.
- The game samples a random bit $b \xleftarrow{\$} \{0, 1\}$, computes $\mathsf{ct}^* \leftarrow \mathsf{PKE.Enc}(\mathsf{pk}, m_b)$, and forwards ct^* to $\mathcal{A}$.
- $\mathcal{A}$ again receives access to a decryption oracle, which on input a ciphertext ct first checks if $\mathsf{ct} = \mathsf{ct}^*$. If so, the oracle returns $\perp$ and otherwise it outputs $\mathsf{PKE.Dec}(\mathsf{sk}, \mathsf{ct})$.
- Eventually, the adversary outputs a bit b^* and wins the game if it holds that $b^* = b$.

3 Password-Authenticated Deterministic Wallets

In this section, we introduce the notion of password-authenticated deterministic wallets (PADW) in the shared-custodial setting by first providing a formal model and then showing an efficient construction. Finally, we formally prove the security of the construction within our model.

3.1 Model

The general setting for PADW schemes resembles the one of blind signature schemes: A blind signature scheme is a two-party primitive that allows a user and a server to engage in an interactive protocol where the server's input is a signing secret key sk and the user's input is a message m. At the end of the protocol the user receives a signature on m under sk. A blind signature must guarantee two security properties, namely *blindness* and *one-more unforgeability.* The former guarantees that the server cannot link a signature to the protocol execution during which it was generated, while the latter guarantees that the user can only obtain n signatures after n signing protocol executions with the server. A password-authenticated deterministic wallet (PADW) scheme extends the notion of blind signatures in three ways: (1) it adds a password-authentication to the signing protocol such that the server only accepts signing requests if the user manages to successfully authenticate itself to the server; (2) it shares the secret key sk between the server and the user such that none of the two parties knows the entire signing secret key; and (3) it allows for deterministic key derivation.

In more detail, a PADW scheme proceeds in three phases, namely an initial **setup phase**, a **key derivation phase**, and an **interactive signing phase**. In the following, we highlight the functionality of each of these three phases.

Setup Phase. During the setup phase, all initial keys and setup values are being generated and exchanged. More concretely, the user generates the scheme's public key pk and the corresponding signing secret key shares for the user and server $(\mathsf{sk}_{\mathcal{U}}, \mathsf{sk}_{\mathcal{S}})$ as well as a seed seed. On the other hand, the server generates an independent server public/secret key pair $(\mathsf{pk}'_{\mathcal{S}}, \mathsf{sk}'_{\mathcal{S}})$, which is required only for the password-authentication during the signing phase. The user then sends pk and $\mathsf{sk}_{\mathcal{S}}$ to the server and registers a password with the server. Finally, the server sends $\mathsf{pk}'_{\mathcal{S}}$ to the user.

Key Derivation Phase. During the key derivation phase, the user can deterministically derive a session user secret key share and the corresponding session public key using seed, $\mathsf{sk}_{\mathcal{U}}$, and pk. To allow for this key derivation, a PADW scheme defines a secret and a public key derivation algorithm. We note that the secret key derivation algorithm allows to only derive session *user* secret key shares, while the server uses its initial secret key share for all sessions. The intuitive reason for this is that the user should be in control of the derivation of session keys, while the server remains unaware of it. In a bit more detail, we would like that the user can locally derive a session public key and a session user secret key share, before initiating the signing process with the server. Note here that the signature output from the signing process is always under a fresh session public key, which is oblivious to the server.

Signing Phase. Finally, during the signing phase, the user and server can engage in a joint signing protocol, where the user can decide which message should be signed for which session key. The server, on the other hand, always executes the signing protocol using its initial secret key share without knowing which message is being signed for which session key. This allows for a strong privacy guarantee, where the server not only does not learn the message and final signature of a signing process (as is the case for standard blind signatures), but also does not learn under which session public key the final signature will be valid. This last property is essential to achieve any meaningful privacy guarantees in the Blockchain setting, since a user will eventually publish the message, signature, and public key to the Blockchain. If the server would learn the user's session public keys, it could trivially link messages and signatures to the user as well.

We present a pictorial presentation of a PADW scheme in Figure 1, and we now present the formal definition of a PADW scheme.

Definition 5 (Password-Authenticated Deterministic Wallet). *A* password-authenticated deterministic wallet scheme PADW *is defined w.r.t. a message space* $\mathcal{M}$ *and a password space* $\mathcal{PW}$ *and is executed between a user* $\mathcal{U}$ *and a server* $\mathcal{S}$*. A* PADW *scheme consists of procedures* $\mathsf{PADW} = (\mathsf{Gen}, \mathsf{Register}, \mathsf{SKDer}, \mathsf{PKDer}, \mathsf{Sign}_{\mathcal{S}}, \mathsf{Sign}_{\mathcal{U}}, \mathsf{Verify})$*, which are defined as as follows:*

– *The probabilistic key generation algorithm* Gen *takes as input a security parameter* κ *and outputs a public key* pk*, a user secret key* $\mathsf{sk}_{\mathcal{U}}$*, a server secret key* $\mathsf{sk}_{\mathcal{S}}$*, and a seed* seed*.*
– *The probabilistic registration algorithm* $\mathsf{Register}$ *takes as input a security parameter* κ *and a password* $\mathsf{pw} \in \mathcal{PW}$ *and outputs a server registration token* $\tau_{\mathcal{S}}^{\mathsf{reg}}$ *and a user registration token* $\tau_{\mathcal{U}}^{\mathsf{reg}}$*.*

– *The deterministic secret key derivation algorithm* SKDer *takes as input a user secret key share* $\mathsf{sk}_{\mathcal{U}}$*, a* seed *and an identifier* ID*. It outputs a user secret key share* $\mathsf{sk}_{\mathcal{U}}^{\mathsf{ID}}$*.*
– *The deterministic public key derivation algorithm* PKDer *takes as input a public key* pk*, a* seed *and an identifier* ID*. It outputs a public key* $\mathsf{pk}^{\mathsf{ID}}$*.*
– *The server signing protocol* $\mathsf{Sign}_{\mathcal{S}}$ *takes as input a secret key share* $\mathsf{sk}_{\mathcal{S}}$ *and a server registration token* $\tau_{\mathcal{S}}^{\mathsf{reg}}$*. It outputs a bit* $b \in \{0,1\}$*.*
– *The user signing protocol* $\mathsf{Sign}_{\mathcal{U}}$ *takes as input a public key* pk*, a user secret key* $\mathsf{sk}_{\mathcal{U}}$*, a password* $\mathsf{pw} \in \mathcal{PW}$*, a user registration token* $\tau_{\mathcal{U}}^{\mathsf{reg}}$*, and a message* $m \in \mathcal{M}$*. It outputs a signature* σ*. Note that it may be that* $\sigma = \bot$*.*
– *The deterministic verification algorithm* Verify *takes as input a public key* pk*, a message* $m \in \mathcal{M}$ *and a signature* σ *and it outputs a bit* $b \in \{0,1\}$*.*

Server $\mathcal{S}$	**Setup Phase**	User $\mathcal{U}$
		$(\mathsf{pk}, \mathsf{sk}_{\mathcal{U}}, \mathsf{sk}_{\mathcal{S}}, \mathsf{seed}) \leftarrow \mathsf{Gen}(1^{\kappa})$
		$(\tau_{\mathcal{S}}^{\mathsf{reg}}, \tau_{\mathcal{U}}^{\mathsf{reg}}) \leftarrow \mathsf{Register}(1^{\kappa}, \mathsf{pw})$
	$\xleftarrow{\mathsf{pk}, \mathsf{sk}_{\mathcal{S}}, \tau_{\mathcal{S}}^{\mathsf{reg}}}$	
		Erase $\mathsf{sk}_{\mathcal{S}}$ and $\tau_{\mathcal{S}}^{\mathsf{reg}}$
Output $(\mathsf{pk}, \mathsf{sk}_{\mathcal{S}}, \tau_{\mathcal{S}}^{\mathsf{reg}})$		Output $(\mathsf{pk}, \mathsf{sk}_{\mathcal{U}}, \mathsf{seed}, \tau_{\mathcal{U}}^{\mathsf{reg}})$
	Key Derivation Phase (offline)	User $\mathcal{U}(\mathsf{pk}, \mathsf{sk}_{\mathcal{U}}, \mathsf{seed}, \mathsf{ID})$
		$\mathsf{pk}^{\mathsf{ID}} \leftarrow \mathsf{PKDer}(\mathsf{pk}, \mathsf{seed}, \mathsf{ID})$
		$\mathsf{sk}_{\mathcal{U}}^{\mathsf{ID}} \leftarrow \mathsf{SKDer}(\mathsf{sk}_{\mathcal{U}}, \mathsf{seed}, \mathsf{ID})$
Server $\mathcal{S}(\mathsf{sk}_{\mathcal{S}}, \tau_{\mathcal{S}}^{\mathsf{reg}})$	**Signing Phase**	User $\mathcal{U}(\mathsf{pk}^{\mathsf{ID}}, \mathsf{sk}_{\mathcal{U}}^{\mathsf{ID}}, \mathsf{pw}, \tau_{\mathcal{U}}^{\mathsf{reg}}, m)$
	$(b, \sigma) \leftarrow \langle \mathsf{Sign}_{\mathcal{S}}(\mathsf{sk}_{\mathcal{S}}, \tau_{\mathcal{S}}^{\mathsf{reg}}), \mathsf{Sign}_{\mathcal{U}}(\mathsf{pk}^{\mathsf{ID}}, \mathsf{sk}_{\mathcal{U}}^{\mathsf{ID}}, \mathsf{pw}, \tau_{\mathcal{U}}^{\mathsf{reg}}, m) \rangle$	
Output b		Output σ

Fig. 1. Example run of a password-authenticated deterministic wallet scheme. During the (one-time) **Setup Phase**, the user generates the public key, (server/user) secret key shares, and seed as well as the necessary tokens for the initial password registration. It then sends the public key, the server secret key share, and the server registration token to the server and erases all server secret values. During the **Key Derivation Phase** the user can derive a session public key and user secret key share. Note that this derivation happens offline, i.e., without interaction with the server. In the **Signing Phase**, the server and user can then jointly generate a signature.

Definition 6 (Correctness). *For any* $\kappa \in \mathbb{N}$*, any* $(\mathsf{pk}, \mathsf{sk}_{\mathcal{U}}, \mathsf{sk}_{\mathcal{S}}, \mathsf{seed}) \leftarrow \mathsf{Gen}(1^{\kappa})$*, and any* $\mathsf{ID} \in \{0,1\}^*$*, we define* $(\mathsf{sk}_{\mathcal{U}}^{\mathsf{ID}}, \mathsf{pk}^{\mathsf{ID}})$ *as* $\mathsf{sk}_{\mathcal{U}}^{\mathsf{ID}} := \mathsf{SKDer}(\mathsf{sk}_{\mathcal{U}}, \mathsf{seed}, \mathsf{ID})$ *and* $\mathsf{pk}^{\mathsf{ID}} := \mathsf{PKDer}(\mathsf{pk}, \mathsf{seed}, \mathsf{ID})$*.*

PADW *is* correct *if for all* $\kappa \in \mathbb{N}$*, all* $(\mathsf{pk}, \mathsf{sk}_{\mathcal{U}}, \mathsf{sk}_{\mathcal{S}}, \mathsf{seed}) \leftarrow \mathsf{Gen}(1^{\kappa})$*, all* $\mathsf{pw} \in \mathcal{PW}$*, all* $m \in \mathcal{M}$*, all* $\mathsf{ID} \in \{0,1\}^*$*, and all* $(\tau_{\mathcal{S}}^{\mathsf{reg}}, \tau_{\mathcal{U}}^{\mathsf{reg}}) \leftarrow \mathsf{Register}(1^{\kappa}, \mathsf{pw})$ *it holds that:*

$$\Pr\left[\mathsf{Verify}(\mathsf{pk}^{\mathsf{ID}}, m, \sigma) = 1\right] = 1 \text{ and } b = 1,$$

where $(b, \sigma) \leftarrow \left\langle \mathsf{Sign}_{\mathcal{S}}(\mathsf{sk}_{\mathcal{S}}, \tau_{\mathcal{S}}^{\mathsf{reg}}), \mathsf{Sign}_{\mathcal{U}}(\mathsf{pk}^{\mathsf{ID}}, \mathsf{sk}_{\mathcal{U}}^{\mathsf{ID}}, \mathsf{pw}, \tau_{\mathcal{U}}^{\mathsf{reg}}, m) \right\rangle$.

We generally assume that the setup phase (cf. Figure 1), i.e., the key generation and password registration is executed by an honest user. The rationale behind this assumption is that a corrupted user during the setup phase can choose the password and use it during the signing phase to successfully authenticate itself to the server. This would trivially allow the malicious user to obtain signatures on arbitrary messages without having to know the server's secret key share. In addition, the assumption of an honest setup phase is crucial to ensure availability (see Section 4.2 for more details). For simplicity and in line with previous works [1, 6], we assume in our models that the user chooses the password uniformly at random from the password space. We note that we can easily integrate arbitrary password distributions in our model. Moreover, our constructions remain secure in such a more general model (albeit with a loss depending on the predictability of the distribution). Finally, we generally assume that the user's password is not stored on the user's device. That is, when the adversary corrupts the device of the user, it learns all high-entropy secrets stored on the device (including the user's secret key share), but it does not learn the password. A PADW scheme must satisfy the security properties of *user unforgeability*, *server unforgeability*, and *blindness*.

User unforgeability. Intuitively, user unforgeability of a PADW scheme guarantees that it is infeasible for an adaptive adversary, which corrupts the user, to generate *any* valid signature unless it guesses the password of the user or breaks the unforgeability of Schnorr signatures. The adversary is, however, restricted to a fixed amount of (online) password guesses, which models the situation of most online services, which block the account of a user after a certain amount of failed login attempts.

More concretely, the user unforgeability game **u-wunf** is parameterized by an integer k, which represents the maximum number of failed password guesses of an adversary[10]. The game then proceeds as follows. It first executes the setup phase for server and user, i.e., it generates all values that the server and user generate during the initial setup. The adversary then receives the public key pk as input and obtains access to a corruption, a signing, and a key derivation oracle. The corruption oracle can be queried at *any* time throughout the game (after the setup phase) and essentially returns the entire internal state of the user to the adversary (including the user secret key share $\mathsf{sk}_{\mathcal{U}}$ and the user registration token $\tau_{\mathcal{U}}^{\mathsf{reg}}$). The signing oracle behaves differently depending on whether the adversary has already corrupted the user or not. If the adversary queries the signing oracle *before* the corruption oracle, the signing oracle takes as input a message m and an identity ID and computes a valid signature for m under public key $\mathsf{pk}^{\mathsf{ID}}$. This essentially models an adversary that observes honestly generated signatures for specific keys. If the adversary queries the signing oracle *after* the corruption oracle, the oracle and the adversary jointly execute the signing procedure, where the oracle executes the server signing procedure. Additionaly, the adversary may also query the corruption oracle *during* a signing oracle query. In this case, the adversary may continue the signing execution of the current oracle query. If an execution of the signing oracle fails, i.e., if the server signing procedure outputs 0, then the game registers this oracle call as a failed password guess and increases a counter variable ctr. Finally, the adversary obtains access to a key derivation oracle, which outputs a session public key for an adversarially chosen identity.

Eventually, the adversary outputs a forgery $(\sigma^*, m^*, \mathsf{ID}^*)$ and wins the game if (1) the signature σ^* is valid w.r.t. to message m^* and public key $\mathsf{pk}^{\mathsf{ID}^*}$, (2) the failed password attempts are smaller than k, i.e., if $\mathsf{ctr} \leq k$, and (3) the signing oracle *before* corruption has not been queried on m^* and ID^* previously. Note that we make no restrictions on the forgery w.r.t. signing oracle queries *after* corruption. That is, the forgery may be a signature that was output by the signing oracle after corruption. This captures that user unforgeability crucially relies on the adversary not being able to correctly guess the password: after corruption, the adversary can only generate valid signatures via the signing oracle if it knows the correct password. However, if the adversary manages to do so, the signature as output by the signing oracle must be considered a valid forgery.

Definition 7 (User unforgeability). *A password-authenticated deterministic wallet* $\mathsf{PADW} = (\mathsf{Gen}, \mathsf{Register}, \mathsf{SKDer}, \mathsf{PKDer}, \mathsf{Sign}_{\mathcal{S}}, \mathsf{Sign}_{\mathcal{U}}, \mathsf{Verify})$ *is user unforgeable if for any PPT adversary* $\mathcal{A}$*, any password space* $\mathcal{PW}$ *with* $0 < |\mathcal{PW}| = poly(\kappa)$*, and any* $k \in \mathbb{N}$ *with* $k < |\mathcal{PW}|$ *it holds*

$$\Pr[\mathbf{u\text{-}wunf}^{\mathcal{A},k}_{\mathsf{PADW}}(1^\kappa) = 1] \leq \frac{k}{|\mathcal{PW}|} + \mathsf{negl}(\kappa),$$

where negl *is a negligible function in the security parameter* κ*.*

Game $\mathbf{u\text{-}wunf}^{\mathcal{A},k}_{\mathsf{PADW}}(1^\kappa)$

- The game generates $(\mathsf{pk}, \mathsf{sk}_{\mathcal{U}}, \mathsf{sk}_{\mathcal{S}}, \mathsf{seed}) \leftarrow \mathsf{Gen}(1^\kappa)$ and samples a password uniformly at random $\mathsf{pw} \xleftarrow{\$} \mathcal{PW}$. The game additionally executes $(\tau_{\mathcal{S}}^{\mathsf{reg}}, \tau_{\mathcal{U}}^{\mathsf{reg}}) \leftarrow \mathsf{Register}(1^\kappa, \mathsf{pw})$ and initiates a counter $\mathsf{ctr} := 0$ as well as two lists $\mathsf{SigList} := \emptyset$ and $\mathsf{KeyList} := \emptyset$.
- The game runs adversary $\mathcal{A}(\mathsf{pk})$ and grants $\mathcal{A}$ access to the following oracles:

[10] In practice, k is typically set to a low single digit integer, e.g. $k = 3$.

- **Corruption oracle:** Upon a query by $\mathcal{A}$, the oracle returns the user secret key $\mathsf{sk}_{\mathcal{U}}$, the seed seed, the user registration token $\tau_{\mathcal{U}}^{\mathsf{reg}}$, and the entire internal state of the user. The adversary may query this oracle at any point, in particular before, after or during a signing oracle query.
- **Signing oracle:**
  - ∗ **Before corruption:** Upon a query by $\mathcal{A}$ on message m and identity ID, the oracle first checks if $\mathsf{KeyList}[\mathsf{ID}] = \bot$. If so, the oracle outputs $\bot$. Otherwise, it fetches $(\mathsf{pk}^{\mathsf{ID}}, \mathsf{sk}_{\mathcal{U}}^{\mathsf{ID}}) \leftarrow \mathsf{KeyList}[\mathsf{ID}]$ and adds $(\mathsf{pk}^{\mathsf{ID}}, m)$ to $\mathsf{SigList}$. The oracle then executes $\left\langle \mathsf{Sign}_{\mathcal{S}}(\mathsf{sk}_{\mathcal{S}}, \tau_{\mathcal{S}}^{\mathsf{reg}}), \mathsf{Sign}_{\mathcal{U}}(\mathsf{pk}^{\mathsf{ID}}, \mathsf{sk}_{\mathcal{U}}^{\mathsf{ID}}, \mathsf{pw}, \tau_{\mathcal{U}}^{\mathsf{reg}}, m) \right\rangle$ and outputs the resulting signature σ. If $\mathcal{A}$ queries the corruption oracle during the execution of the signing procedure, the oracle and the adversary jointly finish the current execution, where the oracle executes the remaining instructions of the server signing procedure $\mathsf{Sign}_{\mathcal{S}}$. If the server signing procedure outputs 0, then the oracle sets $\mathsf{ctr} \leftarrow \mathsf{ctr} + 1$.
  - ∗ **After corruption:** Upon a query by $\mathcal{A}$, the oracle behaves as follows: If it holds that $\mathsf{ctr} > k$, then the oracle returns 0 and aborts. Otherwise, the oracle and $\mathcal{A}$ jointly execute $\langle \mathsf{Sign}_{\mathcal{S}}(\mathsf{sk}_{\mathcal{S}}, \tau_{\mathcal{S}}^{\mathsf{reg}}), \mathsf{Sign}_{\mathcal{U}}(\cdot) \rangle$, where the oracle executes the server signing procedure $\mathsf{Sign}_{\mathcal{S}}$. If the server signing procedure outputs 0, then the oracle sets $\mathsf{ctr} \leftarrow \mathsf{ctr} + 1$.
- **Key derivation oracle:** On input an identity ID, this oracle computes $\mathsf{pk}^{\mathsf{ID}} \leftarrow \mathsf{PKDer}(\mathsf{pk}, \mathsf{seed}, \mathsf{ID})$, $\mathsf{sk}_{\mathcal{U}}^{\mathsf{ID}} \leftarrow \mathsf{SKDer}(\mathsf{sk}_{\mathcal{U}}, \mathsf{seed}, \mathsf{ID})$ and stores $\mathsf{pk}^{\mathsf{ID}}$ and $\mathsf{sk}_{\mathcal{U}}^{\mathsf{ID}}$ in $\mathsf{KeyList}$, i.e., $\mathsf{KeyList}[\mathsf{ID}] \leftarrow (\mathsf{pk}^{\mathsf{ID}}, \mathsf{sk}_{\mathcal{U}}^{\mathsf{ID}})$. The oracle outputs $\mathsf{pk}^{\mathsf{ID}}$.

– Finally, upon the adversary outputting a forgery $(\sigma^*, m^*, \mathsf{ID}^*)$, the game first computes $\mathsf{pk}^{\mathsf{ID}^*} \leftarrow \mathsf{PKDer}(\mathsf{pk}, \mathsf{seed}, \mathsf{ID}^*)$. The adversary wins the game if the following conditions hold:
(1) $\mathsf{Verify}(\mathsf{pk}^{\mathsf{ID}^*}, m^*, \sigma^*) = 1$, (2) $\mathsf{ctr} \leq k$, and (3) $(\mathsf{pk}^{\mathsf{ID}^*}, m^*) \notin \mathsf{SigList}$.

Server unforgeability. Intuitively, server unforgeability of a PADW scheme guarantees that an adaptive adversary corrupting the server cannot forge a signature. We formalize this property via the game **s-wunf** below. Similarly to the user unforgeability game, the **s-wunf** game first generates the setup values. The adversary then receives as input the public key pk and access to a corruption, signing, and key derivation oracle. The oracles are defined in a very similar way as the respective oracles in game **u-wunf** with the difference that the corruption oracle, upon querying, returns the internal state of the server and the signing oracle after corruption executes the user signing procedure.

Eventually, the adversary outputs a forgery $(\sigma^*, m^*, \mathsf{ID}^*)$ and wins the game if (1) σ^* is a valid signature for m^* and $\mathsf{pk}^{\mathsf{ID}^*}$, and (2) m^* has previously never been queried to the signing oracle (before or after corruption) for identity ID^*.

Definition 8 (Server unforgeability). *A password-authenticated deterministic wallet scheme* $\mathsf{PADW} = (\mathsf{Gen}, \mathsf{Register}, \mathsf{SKDer}, \mathsf{PKDer}, \mathsf{Sign}_{\mathcal{S}}, \mathsf{Sign}_{\mathcal{U}}, \mathsf{Verify})$ *is server unforgeable if for any PPT adversary* $\mathcal{A}$ *it holds that*

$$\Pr[\textbf{s-wunf}_{\mathsf{PADW}}^{\mathcal{A}}(1^{\kappa}) = 1] \leq \mathsf{negl}(\kappa),$$

where negl *is a negligible function in the security parameter* κ.

Game $\textbf{s-wunf}_{\mathsf{PADW}}^{\mathcal{A}}(1^{\kappa})$

– The game generates $(\mathsf{pk}, \mathsf{sk}_{\mathcal{U}}, \mathsf{sk}_{\mathcal{S}}, \mathsf{seed}) \leftarrow \mathsf{Gen}(1^{\kappa})$ and samples a password uniformly at random $\mathsf{pw} \overset{\$}{\leftarrow} \mathcal{PW}$. The game additionally executes $(\tau_{\mathcal{S}}^{\mathsf{reg}}, \tau_{\mathcal{U}}^{\mathsf{reg}}) \leftarrow \mathsf{Register}(1^{\kappa}, \mathsf{pw})$ and initiates lists $\mathsf{SigList} := \emptyset$ and $\mathsf{KeyList} := \emptyset$.

– The game runs adversary $\mathcal{A}(\mathsf{pk})$ and grants $\mathcal{A}$ access to the following oracles:
- **Corruption oracle:** Upon a query by $\mathcal{A}$, the oracle returns the server secret key share $\mathsf{sk}_{\mathcal{S}}$ and the registration token $\tau_{\mathcal{S}}^{\mathsf{reg}}$, and the entire internal state of the server. The adversary may query this oracle during the execution of a signing oracle query.
- **Signing oracle:**
  - ∗ **Before corruption:** Upon a query by $\mathcal{A}$ on message m and identity ID, the oracle first checks if $\mathsf{KeyList}[\mathsf{ID}] = \bot$. If so, the oracle outputs $\bot$. Otherwise, the oracle fetches $(\mathsf{pk}^{\mathsf{ID}}, \mathsf{sk}_{\mathcal{U}}^{\mathsf{ID}}) \leftarrow \mathsf{KeyList}[\mathsf{ID}]$ and adds $(\mathsf{pk}^{\mathsf{ID}}, m)$ to list $\mathsf{SigList}$. The oracle then executes the signing protocol $\left\langle \mathsf{Sign}_{\mathcal{S}}(\mathsf{sk}_{\mathcal{S}}, \tau_{\mathcal{S}}^{\mathsf{reg}}), \mathsf{Sign}_{\mathcal{U}}(\mathsf{pk}^{\mathsf{ID}}, \mathsf{sk}_{\mathcal{U}}^{\mathsf{ID}}, \mathsf{pw}, \tau_{\mathcal{U}}^{\mathsf{reg}}, m) \right\rangle$ and outputs the resulting signature σ. If $\mathcal{A}$ queries the corruption oracle during the execution of the signing procedure, the oracle and the adversary jointly finish the current execution, where the oracle executes the remaining instructions of the user signing procedure $\mathsf{Sign}_{\mathcal{U}}$.

  * **After corruption:** Upon a query by $\mathcal{A}$ on message m and identity ID, the oracle first checks if $\mathsf{KeyList}[\mathsf{ID}] = \perp$. If so, the oracle outputs $\perp$. Otherwise, the oracle fetches $(\mathsf{pk}^{\mathsf{ID}}, \mathsf{sk}_{\mathcal{U}}^{\mathsf{ID}}) \leftarrow \mathsf{KeyList}[\mathsf{ID}]$ and adds $(\mathsf{pk}^{\mathsf{ID}}, m)$ to list $\mathsf{SigList}$.Then the oracle and $\mathcal{A}$ jointly execute the signing procedure $\left\langle \mathsf{Sign}_{\mathcal{S}}(\cdot), \mathsf{Sign}_{\mathcal{U}}(\mathsf{pk}^{\mathsf{ID}}, \mathsf{sk}_{\mathcal{U}}^{\mathsf{ID}}, \mathsf{pw}, \tau_{\mathcal{U}}^{\mathsf{reg}}, m)\right\rangle$, where the oracle executes $\mathsf{Sign}_{\mathcal{U}}$. The oracle finally outputs a signature σ.
- **Key derivation oracle:** On input an identity ID, this oracle computes $\mathsf{pk}^{\mathsf{ID}} \leftarrow \mathsf{PKDer}(\mathsf{pk}, \mathsf{seed}, \mathsf{ID})$, $\mathsf{sk}_{\mathcal{U}}^{\mathsf{ID}} \leftarrow \mathsf{SKDer}(\mathsf{sk}_{\mathcal{U}}, \mathsf{seed}, \mathsf{ID})$, and stores $\mathsf{pk}^{\mathsf{ID}}$ and $\mathsf{sk}_{\mathcal{U}}^{\mathsf{ID}}$ in $\mathsf{KeyList}$, i.e., $\mathsf{KeyList}[\mathsf{ID}] \leftarrow (\mathsf{pk}^{\mathsf{ID}}, \mathsf{sk}_{\mathcal{U}}^{\mathsf{ID}})$. The oracle outputs $\mathsf{pk}^{\mathsf{ID}}$.

– Finally, upon the adversary outputting a forgery $(\sigma^*, m^*, \mathsf{ID}^*)$, the game computes $\mathsf{pk}^{\mathsf{ID}^*} \leftarrow \mathsf{PKDer}(\mathsf{pk}, \mathsf{seed}, \mathsf{ID}^*)$. The adversary wins the game if the following conditions hold: (1) $\mathsf{Verify}(\mathsf{pk}^{\mathsf{ID}^*}, m^*, \sigma^*) = 1$ and (2) $(\mathsf{pk}^{\mathsf{ID}^*}, m^*) \notin \mathsf{SigList}$.

***Blindness*.** At a high level, the blindness property of a PADW scheme guarantees that a malicious server, upon seeing a signature (along with the corresponding message and public key), cannot distinguish whether it was involved in the signature generation, or whether the signature was generated independently.

We formalize this property via the game **wblind** below, in which the adversary obtains access to a corruption oracle, a signing oracle, a key derivation oracle, and a challenge oracle. While the corruption, signing, and key derivation oracles are defined in the same way as the respective oracles in game **s-wunf**, the challenge oracle does the following: it takes as input two messages m_0 and m_1 as well as an identity ID. It then generates a public key/signature pair for each message, namely $(\mathsf{pk}_0, \sigma_0)$ for message m_δ and $(\mathsf{pk}_1, \sigma_1)$ for message $m_{1-\delta}$, where δ is a uniform random bit. For the first public key/signature pair $(\mathsf{pk}_0, \sigma_0)$, pk_0 is derived as $\mathsf{pk}_0 \leftarrow \mathsf{PKDer}(\mathsf{pk}, \mathsf{seed}, \mathsf{ID})$ and σ_0 is generated in interaction with the adversary (assuming the adversary has already queried the corruption oracle), i.e., σ_0 is computed via a joint execution of the signing protocol. The second pair is generated by the game only (i.e., without involvement of the adversary) in the following way: the game samples fresh setup values and executes the signing protocol on these fresh values. That is, to generate σ_1, the game plays the role of both, the user and the server. Finally, the game samples a random bit b and outputs the public key/signature pair $(\mathsf{pk}_b, \sigma_b)$ to the adversary, which must decide whether or not the pair was generated during the joint signing process between the adversary and the game.

We note that this blindness property already captures the unlinkability of keys property that is typically required by deterministic wallets [14]. The intuitive reason for this is that if session public keys were linkable (and therefore distinguishable from freshly chosen public keys), then the adversary in game **wblind** could trivially win by simply distinguishing the public keys pk_0 and pk_1.

Definition 9 (Blindness). *A password-authenticated deterministic wallet scheme* $\mathsf{PADW} = (\mathsf{Gen}, \mathsf{Register}, \mathsf{SKDer}, \mathsf{PKDer}, \mathsf{Sign}_{\mathcal{S}}, \mathsf{Sign}_{\mathcal{U}}, \mathsf{Verify})$ *satisfies blindness if for any PPT adversary* $\mathcal{A}$ *it holds that*

$$\Pr[\mathbf{wblind}_{\mathsf{PADW}}^{\mathcal{A}}(1^\kappa) = 1] \leq \frac{1}{2} + \mathsf{negl}(\kappa),$$

where negl *is a negligible function in the security parameter* κ.

Game $\mathbf{wblind}_{\mathsf{PADW}}^{\mathcal{A}}(1^\kappa)$

– The game generates keys $(\mathsf{pk}, \mathsf{sk}_{\mathcal{U}}, \mathsf{sk}_{\mathcal{S}}, \mathsf{seed}) \leftarrow \mathsf{Gen}(1^\kappa)$ and samples a password uniformly at random $\mathsf{pw} \xleftarrow{\$} \mathcal{PW}$. The game additionally initiates a list $\mathsf{KeyList} := \emptyset$ and executes $(\tau_{\mathcal{S}}^{\mathsf{reg}}, \tau_{\mathcal{U}}^{\mathsf{reg}}) \leftarrow \mathsf{Register}(1^\kappa, \mathsf{pw})$.

– The game runs adversary $\mathcal{A}(\mathsf{pk})$ and grants $\mathcal{A}$ access to the following oracles:
- **Corruption oracle:** This oracle is defined identically to the corruption oracle in game **s-wunf** (cf. Definition 8).
- **Signing oracle:** This oracle is defined identically to the signing oracle in game **s-wunf** (cf. Definition 8) except that it does not maintain list $\mathsf{SigList}$.
- **Key derivation oracle:** This oracle is defined identically to the key derivation oracle in game **s-wunf** (cf. Definition 8).

- **Challenge oracle:** This oracle must be queried exactly once. On input two messages m_0 and m_1 and an identity ID, the oracle first checks if $\mathsf{KeyList}[\mathsf{ID}] \neq \perp$. If so, the oracle outputs $\perp$. Otherwise, the oracle samples two bits δ and b uniformly at random and computes $\mathsf{pk}_0 \leftarrow \mathsf{PKDer}(\mathsf{pk}, \mathsf{seed}, \mathsf{ID})$, $\mathsf{sk}_{\mathcal{U},0} \leftarrow \mathsf{SKDer}(\mathsf{sk}_{\mathcal{U}}, \mathsf{seed}, \mathsf{ID})$. The oracle then executes the signing protocol

$$\langle \mathsf{Sign}_{\mathcal{S}}(\mathsf{sk}_{\mathcal{S}}, \tau_{\mathcal{S}}^{\mathsf{reg}}), \mathsf{Sign}_{\mathcal{U}}(\mathsf{pk}_0, \mathsf{sk}_{\mathcal{U},0}, \mathsf{pw}, \tau_{\mathcal{U}}^{\mathsf{reg}}, m_\delta)\rangle$$

  to generate a signature σ_0. If the adversary previously or during this execution queried the corruption oracle, then the oracle only executes instructions of the user signing procedure $\mathsf{Sign}_{\mathcal{U}}$. If for the resulting signature σ_0 it holds that $\sigma_0 = \perp$, then the oracle returns $\perp$. The oracle then generates $(\mathsf{pk}_1, \mathsf{sk}_{\mathcal{U},1}, \mathsf{sk}_{\mathcal{S},1}, \mathsf{seed}_1) \leftarrow \mathsf{Gen}(1^\kappa)$, $\mathsf{pw}_1 \xleftarrow{\$} \mathcal{PW}$, $(\tau_{\mathcal{S},1}^{\mathsf{reg}}, \tau_{\mathcal{U},1}^{\mathsf{reg}}) \leftarrow \mathsf{Register}(1^\kappa, \mathsf{pw})$ and executes the signing protocol $\left\langle \mathsf{Sign}_{\mathcal{S}}(\mathsf{sk}_{\mathcal{S},1}, \tau_{\mathcal{S},1}^{\mathsf{reg}}), \mathsf{Sign}_{\mathcal{U}}(\mathsf{pk}_1, \mathsf{sk}_{\mathcal{U},1}, \mathsf{pw}_1, \tau_{\mathcal{U},1}^{\mathsf{reg}}, m_{1-\delta})\right\rangle$ to generate a signature σ_1. Finally, the oracle sets $\mathsf{KeyList}[\mathsf{ID}] \leftarrow (\mathsf{pk}_b, \mathsf{sk}_{\mathcal{U},b})$ and outputs $(\mathsf{pk}_b, \sigma_b)$.

– Eventually, the adversary outputs a bit b' and wins the game if $b' = b$.

We call a PADW scheme secure if it satisfies *user unforgeability*, *server unforgeability*, and *blindness*.

3.2 Construction

In Figures 2 and 3 we present our construction of a password-authenticated deterministic wallet scheme which internally uses a hash function $\mathsf{H} : \{0,1\}^* \to \mathbb{Z}_q$ and a public key encryption scheme $\mathsf{PKE} = (\mathsf{KGen}, \mathsf{Enc}, \mathsf{Dec})$. We denote our construction by $\mathsf{PADW}[\mathsf{H}, \mathsf{PKE}]$. Essentially, our construction is the blind Schnorr signature scheme (cf. Appendix A) translated to the two-party setting with password-authentication and key derivation. That is, during the setup phase, the user generates a seed seed and a Schnorr secret key, which it splits into the server and the user secret key share $\mathsf{sk}_{\mathcal{S}}$ and $\mathsf{sk}_{\mathcal{U}}$. The user also samples all values for the initial password registration via the $\mathsf{Register}$ algorithm. More concretely, this algorithm first samples a random salt ϕ and generates a token $\tau := \mathsf{H}(\phi, \mathsf{pw})$. Additionally, it generates a public/secret key pair of a public key encryption scheme (PKE) $(\mathsf{pk}_{\mathsf{PKE}}, \mathsf{sk}_{\mathsf{PKE}})$ and sets $\tau_{\mathcal{S}}^{\mathsf{reg}} := (\tau, \mathsf{sk}_{\mathsf{PKE}})$ and $\tau_{\mathcal{U}}^{\mathsf{reg}} := (\phi, \mathsf{pk}_{\mathsf{PKE}})$. The user then sends $\tau_{\mathcal{S}}^{\mathsf{reg}}$ and $\mathsf{sk}_{\mathcal{S}}$ to the server.

For the secret and public key derivation algorithms, we closely follow the BIP32 specification [44], which is the most widely used standard for deterministic wallets. That is, we derive a random value $\rho \leftarrow \mathsf{H}(\mathsf{seed}, \mathsf{ID})$, which is then used to derive a new user secret key share as $\mathsf{sk}_{\mathcal{U}}^{\mathsf{ID}} := \mathsf{sk}_{\mathcal{U}} + \rho \mod q$ and the corresponding public key as $\mathsf{pk}^{\mathsf{ID}} := \mathsf{pk} \cdot g^\rho$.

Finally, the signing protocol is similar to the standard blind Schnorr signature scheme (cf. Appendix A) with some differences to accommodate for the two-party and password-authenticated setting. In order to authenticate to the server, the user encrypts the value $\mathsf{H}(sid, c, \mathsf{H}(\phi, \mathsf{pw}))$ under $\mathsf{pk}_{\mathsf{PKE}}$ where sid denotes the identifier for the current session.[11] The user then sends the ciphertext and c to the server, which decrypts the ciphertext using $\mathsf{sk}_{\mathsf{PKE}}$ and checks whether it holds that the decryption equals $\mathsf{H}(sid, c, \tau)$. Let us give an intuition for why the individual components of this authentication mechanism are necessary. Assume the user would not encrypt the value $\mathsf{H}(sid, c, \mathsf{H}(\phi, \mathsf{pw}))$, but just send it in plain to the server. Then, an adversary corrupting the user right after the computation of $\mathsf{H}(sid, c, \mathsf{H}(\phi, \mathsf{pw}))$ could run an offline attack on the password, since the adversary would know sid, c, and ϕ. Now assume the user would not include c into the hash value. Then the authentication would essentially be independent of the message that the user wishes to sign, allowing an adversary to authenticate to the server and signing an arbitrary message. Finally, assume sid was not included in the hash value. Then, the authentication would be independent of the current session such that an adversary could reuse the ciphertext to authenticate in future sessions.

Our $\mathsf{PADW}[\mathsf{H}, \mathsf{PKE}]$ scheme is defined w.r.t. a message space $\mathcal{M} := \{0,1\}^*$ and we use public key prefixing in our construction. More precisely, upon signing or verifying a message, we first prefix it with the corresponding public key. We do so, because Bitcoin implements a public key prefixed version of Schnorr signatures as a means to protect against related-key attacks [45].

Lemma 1. *The* PADW *scheme as depicted in Figures 2 and 3 is correct.*

[11] We assume that user and server agree for each run of the signing protocol on a unique session identifier, which is a reasonable assumption for authenticated channels.

```
PADW.Gen($1^\kappa$)
00 $\mathsf{sk} \xleftarrow{\$} \mathbb{Z}_q$, $\mathsf{pk} := g^{\mathsf{sk}}$
01 $\mathsf{sk}_{\mathcal{S}} \xleftarrow{\$} \mathbb{Z}_q$, $\mathsf{seed} \xleftarrow{\$} \{0,1\}^\kappa$
02 $\mathsf{sk}_{\mathcal{U}} := \mathsf{sk} - \mathsf{sk}_{\mathcal{S}} \mod q$
03 Return $(\mathsf{pk}, \mathsf{sk}_{\mathcal{U}}, \mathsf{sk}_{\mathcal{S}}, \mathsf{seed})$

PADW.Register($1^\kappa, \mathsf{pw}$)
00 $\phi \xleftarrow{\$} \mathbb{Z}_q$, $\tau := \mathsf{H}(\phi, \mathsf{pw})$
01 $(\mathsf{pk}_{\mathsf{PKE}}, \mathsf{sk}_{\mathsf{PKE}}) \leftarrow \mathsf{PKE.KGen}(1^\kappa)$
02 $\tau^{\mathsf{reg}}_{\mathcal{S}} := (\tau, \mathsf{sk}_{\mathsf{PKE}})$, $\tau^{\mathsf{reg}}_{\mathcal{U}} := (\phi, \mathsf{pk}_{\mathsf{PKE}})$
03 Return $(\tau^{\mathsf{reg}}_{\mathcal{S}}, \tau^{\mathsf{reg}}_{\mathcal{U}})$

PADW.SKDer($\mathsf{sk}_{\mathcal{U}}, \mathsf{seed}, \mathsf{ID}$)
00 $\rho \leftarrow \mathsf{H}(\mathsf{seed}, \mathsf{ID})$, $\mathsf{sk}^{\mathsf{ID}}_{\mathcal{U}} := \mathsf{sk}_{\mathcal{U}} + \rho \mod q$
01 Return $\mathsf{sk}^{\mathsf{ID}}_{\mathcal{U}}$

PADW.PKDer($\mathsf{pk}, \mathsf{seed}, \mathsf{ID}$)
00 $\rho \leftarrow \mathsf{H}(\mathsf{seed}, \mathsf{ID})$, $\mathsf{pk}^{\mathsf{ID}} := \mathsf{pk} \cdot g^\rho$
01 Return $\mathsf{pk}^{\mathsf{ID}}$

PADW.Verify($\mathsf{pk}, m, \sigma$)
00 $m' := (\mathsf{pk}, m)$, parse $\sigma := (R, s)$
01 $c \leftarrow \mathsf{H}(R, m')$
02 If $g^s = R \cdot \mathsf{pk}^c$: Return 1
03 Return 0
```

Fig. 2. Our $\mathsf{PADW[H, PKE]}$ scheme, where H is a hash function $\mathsf{H} : \{0,1\}^* \to \mathbb{Z}_q$ and $\mathsf{PKE} := (\mathsf{KGen}, \mathsf{Enc}, \mathsf{Dec})$ is a public key encryption scheme.

```
Server $\mathcal{S}(\mathsf{sk}_{\mathcal{S}}, \tau^{\mathsf{reg}}_{\mathcal{S}})$                User $\mathcal{U}(\mathsf{pk}, \mathsf{sk}_{\mathcal{U}}, \mathsf{pw}, \tau^{\mathsf{reg}}_{\mathcal{U}}, m)$

01 $r \xleftarrow{\$} \mathbb{Z}_q$, $R := g^r$, set $\mathsf{st}_{\mathcal{S}} := (r)$   $\xrightarrow{R}$
                                  02 $\alpha \xleftarrow{\$} \mathbb{Z}_q$, $\beta \xleftarrow{\$} \mathbb{Z}_q$
                                  03 $R' := R \cdot g^\alpha \cdot \mathsf{pk}^\beta$
                                  04 $c' := \mathsf{H}(R', (\mathsf{pk}, m))$
                                  05 $c := c' + \beta \mod q$
                                  06 Parse $\tau^{\mathsf{reg}}_{\mathcal{U}} := (\phi, \mathsf{pk}_{\mathsf{PKE}})$
                                  07 $\mathsf{ct} := \mathsf{PKE.Enc}(\mathsf{pk}_{\mathsf{PKE}}, \mathsf{H}(sid, c, \mathsf{H}(\phi, \mathsf{pw})))$
                                  08 Set $\mathsf{st}_{\mathcal{U}} := (\alpha, \beta, R, \mathsf{ct}, c, \mathsf{sk}_{\mathcal{U}}, \tau^{\mathsf{reg}}_{\mathcal{U}})$.
                                  09 Erase all other values.
                     $\xleftarrow{c, \mathsf{ct}}$
10 Parse $\tau^{\mathsf{reg}}_{\mathcal{S}} := (\tau, \mathsf{sk}_{\mathsf{PKE}})$ and $\mathsf{st}_{\mathcal{S}} := (r)$
11 If $\mathsf{PKE.Dec}(\mathsf{sk}_{\mathsf{PKE}}, \mathsf{ct}) \neq \mathsf{H}(sid, c, \tau)$:
12     return 0 and abort
13 $s = r + c \cdot \mathsf{sk}_{\mathcal{S}} \mod q$
14 Output 1                           $\xrightarrow{s}$
                                  15 Parse $\mathsf{st}_{\mathcal{U}} := (\alpha, \beta, R, \mathsf{ct}, c, \mathsf{sk}_{\mathcal{U}}, \tau^{\mathsf{reg}}_{\mathcal{U}})$
                                  16 If $g^s \neq R \cdot (\mathsf{pk} \cdot g^{-\mathsf{sk}_{\mathcal{U}}})^c$: $\sigma := \bot$
                                  17 $s' := s + \alpha + c \cdot \mathsf{sk}_{\mathcal{U}} \mod q$
                                  18 $R' := R \cdot g^\alpha \cdot \mathsf{pk}^\beta$
                                  19 Output $\sigma' := (R', s')$
```

Fig. 3. Signing protocol of our $\mathsf{PADW[H, PKE]}$ scheme.

Proof. Note that for any identity $\mathsf{ID} \in \{0,1\}^*$ and $(\mathsf{pk}^{\mathsf{ID}}, \mathsf{sk}^{\mathsf{ID}}_{\mathcal{U}}) \leftarrow (\mathsf{PKDer}(\mathsf{pk}, \mathsf{seed}, \mathsf{ID}), \mathsf{SKDer}(\mathsf{sk}_{\mathcal{U}}, \mathsf{seed}, \mathsf{ID}))$, we have that $\mathsf{pk}^{\mathsf{ID}} := \mathsf{pk} \cdot g^\rho$ and $\mathsf{sk}^{\mathsf{ID}}_{\mathcal{U}} := \mathsf{sk}_{\mathcal{U}} + \rho \mod q$ where $\rho := \mathsf{H}(\mathsf{seed}, \mathsf{ID})$. Then for a signature $\sigma' := (R', s')$ as output by the user signing protocol $\mathsf{Sign}_{\mathcal{U}}$ on input $(\mathsf{pk}^{\mathsf{ID}}, \mathsf{sk}^{\mathsf{ID}}_{\mathcal{U}}, \mathsf{pw}, \tau^{\mathsf{reg}}_{\mathcal{U}}, m)$ the following holds:

$$\begin{aligned} g^{s'} &= g^{s+\alpha+c\cdot\mathsf{sk}^{\mathsf{ID}}_{\mathcal{U}}} = g^{r+c\cdot\mathsf{sk}_{\mathcal{S}}+\alpha+c\cdot\mathsf{sk}^{\mathsf{ID}}_{\mathcal{U}}} = g^{r+c\cdot\mathsf{sk}_{\mathcal{S}}+\alpha+c\cdot(\mathsf{sk}_{\mathcal{U}}+\rho)} = g^{r+\alpha+c\cdot\mathsf{sk}+c\cdot\rho} \\ &= g^{r+\alpha+(c'+\beta)\cdot\mathsf{sk}+(c'+\beta)\cdot\rho} = g^{r+\alpha+\beta\cdot\mathsf{sk}+\beta\cdot\rho+c'\cdot\mathsf{sk}+c'\cdot\rho} = g^{r+\alpha+\beta\cdot(\mathsf{sk}+\rho)} \cdot g^{c'\cdot(\mathsf{sk}+\rho)} = R' \cdot {\mathsf{pk}^{\mathsf{ID}}}^{c'} \end{aligned}$$

□

3.3 Security Analysis

Theorem 1. *Let* $\mathsf{H} : \{0,1\}^* \to \mathbb{Z}_q$ *be a hash function modeled as a random oracle. Assuming the* $\mathsf{Schnorr[H]}$ *as described in Figure 4 is a* **uf-cma**-*secure signature scheme and* PKE *is an* **IND-CCA**-

secure public key encryption scheme, then $\mathsf{PADW}[\mathsf{H},\mathsf{PKE}]$ *is a secure password-authenticated deterministic wallet scheme.*

In order to prove Theorem 1, we must show that $\mathsf{PADW}[\mathsf{H},\mathsf{PKE}]$ satisfies user unforgeability, server unforgeability, and blindness. For brevity, we sometimes denote $\mathsf{PADW}[\mathsf{H},\mathsf{PKE}]$ by PADW.

Lemma 2. *The* PADW *scheme from Figures 2 and 3 is user unforgeable.*

Before we provide the full formal proof of Lemma 2, we give a proof sketch that outlines the main ideas of the formal proof.

Proof.(Sketch) We prove Lemma 2 via reduction to the **uf-cma**-security of $\mathsf{Schnorr}[\mathsf{H}]$. That is, we show that if there exists a PPT adversary $\mathcal{A}$ that can win game $\mathbf{u\text{-}wunf}^{\mathcal{A},k}_{\mathsf{PADW}}$ with more than $\frac{k}{|\mathcal{PW}|} + \mathsf{negl}(\kappa)$ probability, then we can construct a PPT adversary $\mathcal{B}$ that wins game $\mathbf{uf\text{-}cma}^{\mathcal{B}}_{\mathsf{Schnorr}}$ with more than negligible probability. The main difficulty of our reduction is to show that $\mathcal{A}$ cannot learn the user password with a better probability than simply guessing. Importantly, this should hold even when $\mathcal{A}$ corrupts the user during a signing execution where the correct password is being used for authentication. In more detail, we show that $\mathcal{A}$ does not learn any information about the correct user password regardless of when it decides to corrupt the user. Note that the only way for $\mathcal{A}$ to learn any information about the password is via an honestly generated ciphertext $\mathsf{ct} := \mathsf{PKE.Enc}(\mathsf{pk}_{\mathsf{PKE}}, \mathsf{H}(sid, c, \mathsf{H}(\phi, \mathsf{pw})))$, which the adversary can only learn when it corrupts the user during a signing oracle execution. In our proof, we therefore replace the value $\mathsf{H}(sid, c, \mathsf{H}(\phi, \mathsf{pw}))$ by a uniform random value and show via a reduction to the **IND-CCA**-security of the PKE scheme that $\mathcal{A}$ detects this change at most with negligible probability. After this change, $\mathcal{A}$'s view is independent of the correct user password, which means that the adversary can only try to guess the password. Since the amount of password guesses is restricted by the game parameter k, the probability that $\mathcal{A}$ guesses the correct password is $\frac{k}{|\mathcal{PW}|}$.

As a subsequent step in our reduction, we show how $\mathcal{B}$ can simulate signing oracle queries to $\mathcal{A}$ without knowing the server secret key share. Essentially, we show that $\mathcal{B}$ can use its own signing oracle from game $\mathbf{uf\text{-}cma}^{\mathcal{B}}_{\mathsf{Schnorr}}$ to generate valid signatures and adapt these signatures appropriately to answer $\mathcal{A}$'s signing oracle queries.

As a final step, we show that $\mathcal{B}$ can use a valid forgery as output by $\mathcal{A}$ in game $\mathbf{u\text{-}wunf}^{\mathcal{A},k}_{\mathsf{PADW}}$ to win its own game $\mathbf{uf\text{-}cma}^{\mathcal{B}}_{\mathsf{Schnorr}}$.

□

Proof. Game $\boldsymbol{G}_0$: This is the original game $\mathbf{u\text{-}wunf}^{\mathcal{A},k}_{\mathsf{PADW}}$. It holds that $\Pr[\mathbf{u\text{-}wunf}^{\mathcal{A},k}_{\mathsf{PADW}} = 1] = \Pr[\boldsymbol{G}^{\mathcal{A},k}_0 = 1]$.

Game $\boldsymbol{G}_1$: This game is similar to the previous game with a difference in the signing oracle. If a corruption query occurs right after line 09 in Figure 3 during the execution of the signing protocol on input a message m and an identity ID, the signing oracle behaves exactly as in the previous game with the following difference: Note that $\mathcal{A}$ receives the state $\mathsf{st}_{\mathcal{U}} := (\alpha, \beta, R, \mathsf{ct}, c, \mathsf{sk}^{\mathsf{ID}}_{\mathcal{U}}, \tau^{\mathsf{reg}}_{\mathcal{U}})$ from the corruption oracle. If $\mathcal{A}$ sends (c, ct) to the signing oracle in order to continue the signing execution, the signing oracle skips the check in line 11, i.e., it does not check whether it holds that $\mathsf{PKE.Dec}(\mathsf{sk}_{\mathsf{PKE}}, \mathsf{ct}) \neq \mathsf{H}(sid, c, \tau)$.

Note that since the signing oracle computed the values (c, ct), the check will always pass and therefore it holds that $\Pr[\boldsymbol{G}^{\mathcal{A},k}_0 = 1] = \Pr[\boldsymbol{G}^{\mathcal{A},k}_1 = 1]$.

Game $\boldsymbol{G}_2$: This game is similar to the previous game with a difference in the signing oracle. If a corruption query occurs right after line 09 in Figure 3 during the execution of the signing protocol with session id sid on input a message m and an identity ID, the signing oracle behaves exactly as in the previous game with the following difference: Note that $\mathcal{A}$ receives the state $\mathsf{st}_{\mathcal{U}} := (\alpha, \beta, R, \mathsf{ct}, c, \mathsf{sk}^{\mathsf{ID}}_{\mathcal{U}}, \tau^{\mathsf{reg}}_{\mathcal{U}})$ from the corruption oracle. If $\mathcal{A}$ sends ct to the signing oracle in order to authenticate in a future signing session with session id $sid' \neq sid$, the signing oracle does not execute the check in line 11 but simply outputs 0.

Note that ct is an encryption of $\mathsf{H}(sid, c, \tau)$. Therefore, $\mathcal{A}$ can distinguish this game from the previous one only if it finds a value $\tilde{c}$ such that $\mathsf{H}(sid', \tilde{c}, \tau) = \mathsf{H}(sid, c, \tau)$. Since $\mathcal{A}$ is polynomially bounded it finds such a value $\tilde{c}$ at most with negligible probability. It therefore holds that $\Pr[\boldsymbol{G}^{\mathcal{A},k}_1 = 1] = \Pr[\boldsymbol{G}^{\mathcal{A},k}_2 = 1] + \mathsf{negl}_1(\kappa)$, where negl_1 is a negligible function in κ.

Game $\boldsymbol{G}_3$: This game is similar to the previous game with a difference in the signing oracle. If a corruption query occurs right after line 09 in Figure 3 during the execution of the signing protocol on input a message m and an identity ID, the signing oracle behaves exactly as in the previous game with the following difference: Instead of computing $\mathsf{ct} := \mathsf{PKE.Enc}(\mathsf{pk}_{\mathsf{PKE}}, \mathsf{H}(sid, c, \mathsf{H}(\phi, \mathsf{pw})))$, the game computes $\mathsf{ct}' := \mathsf{PKE.Enc}(\mathsf{pk}_{\mathsf{PKE}}, \psi)$ for $\psi \xleftarrow{\$} \mathbb{Z}_q$.

We can show the indistinguishability of this game from the previous game via a reduction to the **IND-CCA**-security of PKE. More concretely, we can show that if $\mathcal{A}$ can distinguish this game from the previous one with non-negligible advantage, then we can construct a PPT adversary $\mathcal{C}$ which uses $\mathcal{A}$ to break the **IND-CCA**-security of PKE with the same non-negligible advantage. That is, during a signing oracle query before corruption from $\mathcal{A}$, the adversary $\mathcal{C}$ executes the signing protocol using its decryption oracle from the **IND-CCA** game. Then, upon $\mathcal{A}$ querying the corruption oracle right after line 09 in Figure 3 during the execution of the signing protocol, $\mathcal{C}$ sends two messages $m_0 := \mathsf{H}(sid, c, \mathsf{H}(\phi, \mathsf{pw}))$ and $m_1 := \psi$ to its game and outputs the resulting ciphertext to $\mathcal{A}$. The simulation of the rest of the game is straightforward with $\mathcal{C}$ still using its decryption oracle for the execution of subsequent signing oracle queries. Finally, if $\mathcal{A}$ decides that it plays in game $\boldsymbol{G}_3$, then $\mathcal{C}$ outputs 1 to its game and 0 otherwise. It is easy to see that $\mathcal{C}$ wins the **IND-CCA** game with the same advantage with which $\mathcal{A}$ distinguishes games $\boldsymbol{G}_3$ and $\boldsymbol{G}_2$.

We therefore have that $\Pr[\boldsymbol{G}_2^{\mathcal{A},k} = 1] \leq \Pr[\boldsymbol{G}_3^{\mathcal{A},k} = 1] + \mathsf{negl}_2(\kappa)$, where negl_2 is a negligible function in the security parameter κ.

Game $\boldsymbol{G}_4$: This game is similar to the previous game with a difference in the signing oracle. If a corruption query occurs right after line 09 in Figure 3 during the execution of the signing protocol, the signing oracle behaves as follows: Assume $\mathcal{A}$ received the state $\mathsf{st}_{\mathcal{U}} := (\alpha, \beta, R, \mathsf{ct}, c, \mathsf{sk}_{\mathcal{U}}^{\mathsf{ID}}, \tau_{\mathcal{U}}^{\mathsf{reg}})$ from the corruption oracle. If $\mathcal{A}$ sends $(\tilde{c}, \tilde{\mathsf{ct}}) \neq (c, \mathsf{ct})$ to the signing oracle in order to continue the signing execution, the signing oracle outputs 0 and aborts the execution.

Note that this game differs from the previous game only if it holds that $\mathsf{PKE.Dec}(\mathsf{sk}_{\mathsf{PKE}}, \tilde{\mathsf{ct}}) = \mathsf{H}(sid, \tilde{c}, \tau)$, where $\tau := \mathsf{H}(\phi, \mathsf{pw})$. We therefore bound the probability for $\mathcal{A}$ to output a tuple $(\tilde{c}, \tilde{\mathsf{ct}}) \neq (c, \mathsf{ct})$ such that the above condition holds in the following claim.

Claim. Let E_1 be the event that $\mathcal{A}$ sends a tuple $(\tilde{c}, \tilde{\mathsf{ct}}) \neq (c, \mathsf{ct})$ such that $\mathsf{PKE.Dec}(\mathsf{sk}_{\mathsf{PKE}}, \tilde{\mathsf{ct}}) = \mathsf{H}(sid, \tilde{c}, \tau)$. Then, we have that $\Pr[\mathsf{E}_1] \leq \frac{1}{|\mathcal{PW}|} + \mathsf{negl}_3(\kappa)$, where negl_3 is a negligible function in the security parameter κ.

Proof. Event E_1 only happens if $\mathcal{A}$ can either guess the value τ, or if it can guess a password $\mathsf{pw}' \in \mathcal{PW}$ such that $\mathsf{H}(\phi, \mathsf{pw}') = \mathsf{H}(\phi, \mathsf{pw}) = \tau$. Since we model H as a random oracle, τ is uniformly distributed in $\mathbb{Z}_q$ and therefore $\mathcal{A}$ has at most negligible probability to guess τ. In order to determine the probability of $\mathcal{A}$ guessing a $\mathsf{pw}' \in \mathcal{PW}$ such that $\mathsf{H}(\phi, \mathsf{pw}') = \mathsf{H}(\phi, \mathsf{pw}) = \tau$, we distinguish the following two cases: (1) $\mathsf{pw}' \neq \mathsf{pw}$, and (2) $\mathsf{pw}' = \mathsf{pw}$. The probability for case (1), i.e., that $\mathcal{A}$ finds a password $\mathsf{pw}' \neq \mathsf{pw}$ such that $\mathsf{H}(\phi, \mathsf{pw}') = \mathsf{H}(\phi, \mathsf{pw}) = \tau$ can be upper bounded by $\frac{|\mathcal{PW}|-1}{q}$, which is negligible since we have that $|\mathcal{PW}| = \mathrm{poly}(\kappa)$. Further, since pw is chosen uniformly at random from $\mathcal{PW}$, the probability of $\mathcal{A}$ guessing $\mathsf{pw}' = \mathsf{pw}$ is at most $\frac{1}{|\mathcal{PW}|}$. Therefore, overall we have that $\Pr[\mathsf{E}_1] \leq \frac{1}{|\mathcal{PW}|} + \mathsf{negl}_3(\kappa)$. □

Let a be the probability of $\mathcal{A}$ trying to guess either $\mathsf{pw}' \in \mathcal{PW}$ such that $\mathsf{H}(\phi, \mathsf{pw}') = \mathsf{H}(\phi, \mathsf{pw}) = \tau$ or τ. Then it holds that $\Pr[\boldsymbol{G}_3^{\mathcal{A},k} = 1] \leq \Pr[\boldsymbol{G}_4^{\mathcal{A},k} = 1] + a \cdot (\frac{1}{|\mathcal{PW}|} + \mathsf{negl}_3(\kappa))$.

Game $\boldsymbol{G}_5$: This game is similar to the previous game with a difference in the signing oracle. Upon $\mathcal{A}$ querying the signing oracle after corruption, the oracle always returns 0 during the authentication check (line 11 in Figure 3) and aborts.

First, note that this game differs from the previous game only as long as $\mathsf{ctr} \leq k$. Second, note that the signing oracle only returns 0 if it holds that $\mathsf{PKE.Dec}(\mathsf{sk}_{\mathsf{PKE}}, \mathsf{ct}) \neq \mathsf{H}(sid, c, \tau)$, where $\tau := \mathsf{H}(\phi, \mathsf{pw})$. That is, only if $\mathcal{A}$ sends a tuple (c, ct) during the execution of a signing protocol with id sid such that $\mathsf{PKE.Dec}(\mathsf{sk}_{\mathsf{PKE}}, \mathsf{ct}) = \mathsf{H}(sid, c, \tau)$ while it holds that $\mathsf{ctr} < k$, then game $\boldsymbol{G}_5$ differs from $\boldsymbol{G}_4$.

Claim. Let E_2 be the event that $\mathcal{A}$ sends a tuple (c, ct) during the execution of a signing protocol with id sid such that $\mathsf{PKE.Dec}(\mathsf{sk}_{\mathsf{PKE}}, \mathsf{ct}) = \mathsf{H}(sid, c, \tau)$ while it holds that $\mathsf{ctr} \leq k$. Then, we have that $\Pr[\mathsf{E}_2] \leq \frac{k}{|\mathcal{PW}|} + \mathsf{negl}_4(\kappa)$, where negl_4 is a negligible function in the security parameter κ.

Proof. The proof for this claim proceeds similarly to the proof of Claim 3.3 with the only difference that in this claim, we have to consider that $\mathcal{A}$ can make at most k password guesses. Therefore, we bound $\mathcal{A}$'s probability of guessing a password $\mathsf{pw}' \in \mathcal{PW}$ such that $\mathsf{H}(\phi, \mathsf{pw}') = \mathsf{H}(\phi, \mathsf{pw}) = \tau$ by $\frac{k}{|\mathcal{PW}|}$. Therefore, we have that $\Pr[\mathsf{E}_2] \leq \frac{k}{|\mathcal{PW}|} + \mathsf{negl}_4(\kappa)$. $\square$

Overall, we have that $\Pr[\boldsymbol{G}_4^{\mathcal{A},k} = 1] \leq \Pr[\boldsymbol{G}_5^{\mathcal{A},k} = 1] + a \cdot (\frac{k-1}{|\mathcal{PW}|} + \mathsf{negl}_4(\kappa)) + (1-a) \cdot (\frac{k}{|\mathcal{PW}|} + \mathsf{negl}_4(\kappa))$.

Game $\boldsymbol{G}_6$: This game is similar to the previous game with a difference in the signing oracle. If a corruption query occurs right after line 09 in Figure 3 during the execution of the signing protocol on input a message m and an identity ID, the signing oracle proceeds as follows: The signing oracle fetches $(\mathsf{pk}^{\mathsf{ID}}, \mathsf{sk}_{\mathcal{U}}^{\mathsf{ID}}) \leftarrow \mathsf{KeyList}[\mathsf{ID}]$ and executes

$$\left\langle \mathsf{Sign}_{\mathcal{S}}(\mathsf{sk}_{\mathcal{S}}, \tau_{\mathcal{S}}^{\mathsf{reg}}), \mathsf{Sign}_{\mathcal{U}}(\mathsf{pk}^{\mathsf{ID}}, \mathsf{sk}_{\mathcal{U}}^{\mathsf{ID}}, \mathsf{pw}, \tau_{\mathcal{U}}^{\mathsf{reg}}, m) \right\rangle$$

to compute a signature $\sigma := (R', s')$. The oracle then samples uniformly at random $\tilde{\alpha} \xleftarrow{\$} \mathbb{Z}_q$ and $\tilde{\beta} \xleftarrow{\$} \mathbb{Z}_q$ and sets $\tilde{R} := R' \cdot g^{-\tilde{\alpha}} \cdot \mathsf{pk}^{\mathsf{ID}^{-\tilde{\beta}}}$. The oracle then computes

$$\begin{aligned} &c' := \mathsf{H}(R', (\mathsf{pk}^{\mathsf{ID}}, m)), c := c' + \tilde{\beta} \mod q, \\ &\mathsf{ct} := \mathsf{PKE.Enc}(\mathsf{pk}_{\mathsf{PKE}}, \psi), \end{aligned}$$

for $\psi \xleftarrow{\$} \mathbb{Z}_q$. The signing oracle then sets the user's internal state to $\mathsf{st}_{\mathcal{U}} := (\tilde{\alpha}, \tilde{\beta}, \tilde{R}, \mathsf{ct}, c, \mathsf{sk}_{\mathcal{U}}^{\mathsf{ID}}, \tau_{\mathcal{U}}^{\mathsf{reg}})$, which is returned to $\mathcal{A}$ by the corruption oracle. Finally, upon $\mathcal{A}$ sending the tuple (c, ct), the signing oracle simply computes $\tilde{s} := s' - \tilde{\alpha} - \tilde{\beta} \cdot \mathsf{sk}_{\mathcal{U}}^{\mathsf{ID}} - c' \cdot \mathsf{sk}_{\mathcal{U}}^{\mathsf{ID}} \mod q$ and returns $\tilde{s}$ to $\mathcal{A}$ to complete the signing execution.

We need to show that $\mathcal{A}$'s view during the execution of the signing protocol is indistinguishable from the view of a regular signing execution during which $\mathcal{A}$ corrupts the user right after line 09. First, note that $\tilde{\alpha}$ and $\tilde{\beta}$ are uniformly distributed in $\mathbb{Z}_q$. Therefore, $\tilde{\alpha}$, $\tilde{\beta}$, and $\tilde{R}$ are identically distributed to α,β, and R in a regular signing execution. Second, the values ct, c, and $\mathsf{sk}_{\mathcal{U}}^{\mathsf{ID}}$ are computed exactly as in a signing execution of the previous game and are therefore identically distributed. Finally, $\tilde{s}$ and s are both distributed uniformly in $\mathbb{Z}_q$. It remains to show that $\tilde{s}$ satisfies the check of line 16 in Figure 3 and that $\tilde{s}$ indeed allows $\mathcal{A}$ to compute a valid signature for message m and identity ID. Note that for the check in line 16, it must hold that $g^{\tilde{s}} = \tilde{R} \cdot (\mathsf{pk}^{\mathsf{ID}} \cdot g^{-\mathsf{sk}_{\mathcal{U}}^{\mathsf{ID}}})^c$. We show that this check holds in the following:

$$\begin{aligned} g^{\tilde{s}} &= g^{s' - \tilde{\alpha} - \tilde{\beta} \cdot \mathsf{sk}_{\mathcal{U}}^{\mathsf{ID}} - c' \cdot \mathsf{sk}_{\mathcal{U}}^{\mathsf{ID}}} = g^{r'} \cdot \mathsf{pk}^{\mathsf{ID}^{c'}} \cdot g^{-\tilde{\alpha} - \tilde{\beta} \cdot \mathsf{sk}_{\mathcal{U}}^{\mathsf{ID}} - c' \cdot \mathsf{sk}_{\mathcal{U}}^{\mathsf{ID}}} \\ &= \underbrace{R' \cdot g^{-\tilde{\alpha}} \cdot \mathsf{pk}^{\mathsf{ID}^{-\tilde{\beta}}}}_{\tilde{R}} \cdot \mathsf{pk}^{\mathsf{ID}^{\tilde{\beta}}} \cdot g^{-\tilde{\beta} \cdot \mathsf{sk}_{\mathcal{U}}^{\mathsf{ID}}} \cdot \mathsf{pk}^{\mathsf{ID}^{c'}} \cdot g^{-c' \cdot \mathsf{sk}_{\mathcal{U}}{}^{\mathsf{ID}}} \\ &= \tilde{R} \cdot \mathsf{pk}^{\mathsf{ID}^{\tilde{\beta}}} \cdot g^{-\tilde{\beta} \cdot \mathsf{sk}_{\mathcal{U}}{}^{\mathsf{ID}}} \cdot \mathsf{pk}^{\mathsf{ID}^{c'}} \cdot g^{-c' \cdot \mathsf{sk}_{\mathcal{U}}{}^{\mathsf{ID}}} \\ &= \tilde{R} \cdot g^{\tilde{\beta} \cdot \mathsf{sk}_{\mathcal{S}}} \cdot g^{c' \cdot \mathsf{sk}_{\mathcal{S}}} \\ &= \tilde{R} \cdot g^{(\tilde{\beta} + c') \cdot \mathsf{sk}_{\mathcal{S}}} = \tilde{R} \cdot (\mathsf{pk}^{\mathsf{ID}} \cdot g^{-\mathsf{sk}_{\mathcal{U}}{}^{\mathsf{ID}}})^c \end{aligned}$$

Lastly, we have to show that if $\mathcal{A}$ follows the steps in lines 17 and 18 (in Figure 3), then $\mathcal{A}$'s output constitutes indeed a valid signature on message m. We have for s' as computed in line 17 the following:

$$\begin{aligned} &\tilde{s} + \tilde{\alpha} + c \cdot \mathsf{sk}_{\mathcal{U}}^{\mathsf{ID}} \\ &= s' - \tilde{\alpha} - \tilde{\beta} \cdot \mathsf{sk}_{\mathcal{U}}^{\mathsf{ID}} - c' \cdot \mathsf{sk}_{\mathcal{U}}^{\mathsf{ID}} + \tilde{\alpha} + (c' + \tilde{\beta}) \cdot \mathsf{sk}_{\mathcal{U}}^{\mathsf{ID}} = s' \end{aligned}$$

And for R' as computed in line 18 we have:

$$\tilde{R} \cdot g^{\tilde{\alpha}} \cdot \mathsf{pk}^{\mathsf{ID}^{\tilde{\beta}}} = R' \cdot g^{-\tilde{\alpha}} \cdot \mathsf{pk}^{\mathsf{ID}^{-\tilde{\beta}}} \cdot g^{\tilde{\alpha}} \cdot \mathsf{pk}^{\mathsf{ID}^{\tilde{\beta}}} = R'$$

Therefore, $\mathcal{A}$ obtains the signature $\sigma := (R', s')$ as output, which is a valid signature for message m and identity ID. Therefore, we have that $\Pr[\boldsymbol{G}_6^{\mathcal{A},k} = 1] = \Pr[\boldsymbol{G}_5^{\mathcal{A},k} = 1]$.

Game $\boldsymbol{G}_7$: This game is similar to the previous game with a difference in the signing oracle. If a corruption query occurs right after line 09 in Figure 3 during the execution of the signing protocol on input a message m and an identity ID, the signing oracle behaves exactly as in the previous game with the following difference. Instead of computing the signature $\sigma := (R', s')$ as

$$\left\langle \mathsf{Sign}_{\mathcal{S}}(\mathsf{sk}_{\mathcal{S}}, \tau_{\mathcal{S}}^{\mathsf{reg}}), \mathsf{Sign}_{\mathcal{U}}(\mathsf{pk}^{\mathsf{ID}}, \mathsf{sk}_{\mathcal{U}}^{\mathsf{ID}}, \mathsf{pw}, \tau_{\mathcal{U}}^{\mathsf{reg}}, m) \right\rangle,$$

this game does the following: it first computes $\tilde{R}' := g^{\tilde{r}'}$ for $\tilde{r}' \xleftarrow{\$} \mathbb{Z}_q$ and then it sets $\tilde{s}' := \tilde{r}' + c' \cdot (\mathsf{sk}_{\mathcal{U}}^{\mathsf{ID}} + \mathsf{sk}_{\mathcal{S}}) \mod q$ $(= \tilde{r}' + c' \cdot (\mathsf{sk} + \rho) \mod q)$ for $c' := \mathsf{H}(\tilde{R}', \mathsf{pk}^{\mathsf{ID}}, m)$. The game then sets $\sigma := (\tilde{R}', \tilde{s}')$.

It is easy to see that $\sigma := (\tilde{R}', \tilde{s}')$ as computed in this game constitutes a valid signature for message m under public key $\mathsf{pk}^{\mathsf{ID}}$. Therefore, we have that $\Pr[\boldsymbol{G}_7^{\mathcal{A},k} = 1] = \Pr[\boldsymbol{G}_6^{\mathcal{A},k} = 1]$.

Game $\boldsymbol{G}_8$: This game is similar to the previous game with a difference in the signing oracle. If a corruption query occurs right after line 09 in Figure 3 during the execution of the signing protocol on input a message m and an identity ID, the signing oracle behaves exactly as in the previous game with the following difference. This game computes signature σ as follows: it computes $\tilde{R}' := g^{\tilde{r}'}$ for $\tilde{r}' \xleftarrow{\$} \mathbb{Z}_q$ and then it sets $\tilde{s}' := \tilde{r}' + c' \cdot (\mathsf{sk}_{\mathcal{U}} + \mathsf{sk}_{\mathcal{S}}) \mod q$ $(= \tilde{r}' + c' \cdot \mathsf{sk} \mod q)$ for $c' := \mathsf{H}(\tilde{R}', \mathsf{pk}^{\mathsf{ID}}, m)$. The game then computes $\rho := \mathsf{H}(\mathsf{seed}, \mathsf{ID})$ and sets $s' := \tilde{s}' + c' \cdot \rho \mod q$. The then sets $\sigma := (\tilde{R}', s')$.

It is easy to see that $\sigma := (\tilde{R}', s')$ as computed in this game constitutes a valid signature for message m under public key $\mathsf{pk}^{\mathsf{ID}}$ since we have that

$$\begin{aligned} s' := \tilde{s}' + c' \cdot \rho \mod q &= \tilde{r}' + c' \cdot \mathsf{sk} + c' \cdot \rho \mod q \\ &= \tilde{r}' + c' \cdot (\mathsf{sk} + \rho) \mod q. \end{aligned}$$

Therefore, we have that $\Pr[\boldsymbol{G}_8^{\mathcal{A},k} = 1] = \Pr[\boldsymbol{G}_7^{\mathcal{A},k} = 1]$.

Game $\boldsymbol{G}_9$: This game is similar to the previous game with a difference in the signing oracle. If a corruption query occurs right before line 15 in Figure 3 during the execution of the signing protocol on input a message m and an identity ID, the signing oracle behaves exactly as in the case when a corruption occurs right after line 09.

The indistinguishability of this game to the previous game follows in the same way as in game $\boldsymbol{G}_6$. Therefore, we have that $\Pr[\boldsymbol{G}_9^{\mathcal{A},k} = 1] = \Pr[\boldsymbol{G}_8^{\mathcal{A},k} = 1]$.

Game $\boldsymbol{G}_{10}$: This game is similar to the previous game with a difference in the key derivation oracle. Upon a query on input ID, the game does not compute the user session secret key share $\mathsf{sk}_{\mathcal{U}}^{\mathsf{ID}}$. Instead it stores $(\mathsf{pk}^{\mathsf{ID}}, \cdot)$ in $\mathsf{KeyList}[\mathsf{ID}]$.

Note that the game does not use the user session secret key share in the signing oracle. It only uses the fully secret key sk. Therefore, this game is equivalent to the previous game and we have that $\Pr[\boldsymbol{G}_{10}^{\mathcal{A}} = 1] = \Pr[\boldsymbol{G}_9^{\mathcal{A}} = 1]$.

Therefore, it holds that

$$\begin{aligned} &\Pr[\mathbf{u\text{-}wunf}_{\mathsf{PADW}}^{\mathcal{A},k} = 1] = \Pr[\boldsymbol{G}_0^{\mathcal{A},k} = 1] \\ &\leq \Pr[\boldsymbol{G}_{10}^{\mathcal{A},k} = 1] + \mathsf{negl}_1(\kappa) \\ &+ \mathsf{negl}_2(\kappa) + a \cdot \left(\frac{1}{|\mathcal{PW}|} + \mathsf{negl}_3(\kappa) \right) \\ &+ a \cdot \left(\frac{k-1}{|\mathcal{PW}|} + \mathsf{negl}_4(\kappa) \right) + (1-a) \cdot \left(\frac{k}{|\mathcal{PW}|} + \mathsf{negl}_4(\kappa) \right) \\ &= \Pr[\boldsymbol{G}_{10}^{\mathcal{A},k} = 1] + \frac{k}{|\mathcal{PW}|} + \mathsf{negl}(\kappa), \end{aligned}$$

where $\mathsf{negl}(\kappa) = \mathsf{negl}_1(\kappa) + \mathsf{negl}_2(\kappa) + a \cdot \mathsf{negl}_3(\kappa) + \mathsf{negl}_4(\kappa)$.

Reduction to the uf-cma security of $\mathsf{Schnorr}[\mathsf{H}]$. It remains to show that if adversary $\mathcal{A}$ wins game $\boldsymbol{G}_{10}^{\mathcal{A},k}$ with more than $\frac{k}{|\mathcal{PW}|} + \mathsf{negl}(\kappa)$ probability, i.e., $\Pr[\boldsymbol{G}_{10}^{\mathcal{A},k} = 1] > \frac{k}{|\mathcal{PW}|} + \mathsf{negl}(\kappa)$, then there exists an adversary $\mathcal{B}$ playing in game $\textbf{uf-cma}^{\mathcal{B}}_{\mathsf{Schnorr}}$ that simulates game $\boldsymbol{G}^{\mathcal{A},k}_{10}$ to $\mathcal{A}$ and uses $\mathcal{A}$'s output to win game $\textbf{uf-cma}^{\mathcal{B}}_{\mathsf{Schnorr}}$ with more than negligible probability. $\mathcal{B}$'s simulation differs from game $\boldsymbol{G}^{\mathcal{A},k}_{10}$ only in the following ways:

1. Instead of generating the public key pk and the user and server secret key shares $(\mathsf{sk}_{\mathcal{U}}, \mathsf{sk}_{\mathcal{S}})$ in the beginning of the game, $\mathcal{B}$ receives a public key $\mathsf{pk}_{\mathcal{B}}$ from game $\textbf{uf-cma}^{\mathcal{B}}_{\mathsf{Schnorr}}$, which it sets to the wallet public key in $\boldsymbol{G}^{\mathcal{A},k}_{10}$. Then $\mathcal{B}$ samples uniformly at random $\mathsf{sk}_{\mathcal{U}} \xleftarrow{\$} \mathbb{Z}_q$.
2. $\mathcal{B}$ forwards all random oracle queries from $\mathcal{A}$ to its own oracle.
3. Instead of computing $\sigma := (\tilde{R}', s')$ in the signing oracle, $\mathcal{B}$ queries its own signing oracle to receive a valid signature. Note that if $\mathcal{A}$ queries its signing oracle on input m and ID, then $\mathcal{B}$ queries its own signing oracle on input $(\mathsf{pk}^{\mathsf{ID}}, m)$ to obtain signature $\sigma := (\tilde{R}', s')$.

Clearly, $\mathcal{B}$ perfectly simulates game $\boldsymbol{G}^{\mathcal{A},k}_{10}$. It remains to show that $\mathcal{B}$ can use a valid forgery $(\sigma^*, m^*, \mathsf{ID}^*)$ as output by $\mathcal{A}$ in game $\boldsymbol{G}^{\mathcal{A},k}_{10}$ to win its own game $\textbf{uf-cma}^{\mathcal{B}}_{\mathsf{Schnorr}}$. Upon receiving the forgery, $\mathcal{B}$ does the following. It parses $\sigma^* := (R^*, s^*)$, computes $\rho^* := \mathsf{H}(\mathsf{seed}, \mathsf{ID}^*)$, $\mathsf{pk}^{\mathsf{ID}^*} := \mathsf{pk} \cdot g^{\rho^*}$, and $s^*_{\mathcal{B}} := s^* - c' \cdot \rho^* \bmod q$, where $c' := \mathsf{H}(R^*, \mathsf{pk}^{\mathsf{ID}^*}, m^*)$. $\mathcal{B}$ then sets $\sigma^*_{\mathcal{B}} := (R^*, s^*_{\mathcal{B}})$ and $m^*_{\mathcal{B}} := (\mathsf{pk}^{\mathsf{ID}*}, m^*)$ and outputs the forgery $(\sigma^*_{\mathcal{B}}, m^*_{\mathcal{B}})$ to its game $\textbf{uf-cma}^{\mathcal{B}}_{\mathsf{Schnorr}}$.

Claim. Let $(\sigma^*, m^*, \mathsf{ID}^*)$ be a valid forgery in game $\boldsymbol{G}^{\mathcal{A},k}_{10}$. Then the tuple $(\sigma^*_{\mathcal{B}}, m^*_{\mathcal{B}})$ constitutes a valid forgery in game $\textbf{uf-cma}^{\mathcal{B}}_{\mathsf{Schnorr}}$.

Proof. We know that it holds that $\mathsf{PADW.Verify}(\mathsf{pk}^{\mathsf{ID}^*}, m^*, \sigma^*) = 1$, where $\mathsf{pk}^{\mathsf{ID}^*} := \mathsf{pk} \cdot g^{\rho^*}$ for $\rho^* := \mathsf{H}(\mathsf{seed}, \mathsf{ID}^*)$. That is, for $\sigma^* := (R^*, s^*)$ and $c' := \mathsf{H}(R^*, \mathsf{pk}^{\mathsf{ID}^*}, m^*)$, it holds that

$$g^{s^*} = R^* \cdot \left(\mathsf{pk}^{\mathsf{ID}^*}\right)^{c'} = R^* \cdot \left(\mathsf{pk} \cdot g^{\rho^*}\right)^{c'} = R^* \cdot \mathsf{pk}^{c'} \cdot (g^{c' \cdot \rho^*}).$$

It is then easy to see that for $\sigma^*_{\mathcal{B}} := (R^*, s^*_{\mathcal{B}})$ it holds that $\mathsf{Schnorr.Verify}(\mathsf{pk}, \sigma^*_{\mathcal{B}}, m^*_{\mathcal{B}}) = 1$. More concretely, we have that

$$g^{s^*_{\mathcal{B}}} = g^{s^* - c' \cdot \rho^*} = g^{s^*} \cdot g^{-c' \cdot \rho^*} = R^* \cdot \mathsf{pk}^{c'} \cdot (g^{c' \cdot \rho^*}) \cdot g^{-c' \cdot \rho^*} = R^* \cdot \mathsf{pk}^{c'},$$

where $c' := \mathsf{H}(R^*, m^*_{\mathcal{B}}) = \mathsf{H}(R^*, \mathsf{pk}^{\mathsf{ID}^*}, m^*_{\mathcal{B}})$.

Note that for all signing oracle queries that happen before corruption by $\mathcal{A}$ in game $\textbf{u-wunf}^{\mathcal{A},k}_{\mathsf{PADW}}$ on message m and identity ID, adversary $\mathcal{B}$ queries its own signing oracle in game $\textbf{uf-cma}^{\mathcal{B}}_{\mathsf{Schnorr}}$ on message $(\mathsf{pk}^{\mathsf{ID}}, m)$. Further note that for all signing oracle queries by $\mathcal{A}$ after corruption, the oracle simply outputs 0, i.e., $\mathcal{B}$ does not have to query its own signing oracle. Therefore, since $(\sigma^*, m^*, \mathsf{ID}^*)$ is a valid forgery in game $\textbf{u-wunf}^{\mathcal{A},k}_{\mathsf{PADW}}$, we know that message m^* has never been queried for identity ID^* before and hence, we know that $\mathcal{B}$ has never queried message $m^*_{\mathcal{B}} := (\mathsf{pk}^{\mathsf{ID}*}, m^*)$ to its own signing oracle. □

We therefore have:

$$\Pr[\textbf{u-wunf}^{\mathcal{A},k}_{\mathsf{PADW}} = 1] = \Pr[\boldsymbol{G}^{\mathcal{A},k}_0 = 1] \leq \Pr[\boldsymbol{G}^{\mathcal{A},k}_{10} = 1] + \frac{k}{|\mathcal{PW}|} + \mathsf{negl}(\kappa)$$
$$= \Pr[\textbf{uf-cma}^{\mathcal{B}}_{\mathsf{Schnorr}} = 1] + \frac{k}{|\mathcal{PW}|} + \mathsf{negl}(\kappa).$$

□

Lemma 3. *The* PADW *scheme from Figures 2 and 3 is server unforgeable.*

Proof. Game $\boldsymbol{G}_0$: This is the original game $\textbf{s-unf}^{\mathcal{A}}_{\mathsf{PADW}}$. It holds that $\Pr[\textbf{s-unf}^{\mathcal{A}}_{\mathsf{PADW}} = 1] = \Pr[\boldsymbol{G}^{\mathcal{A}}_0 = 1]$.

Game $\boldsymbol{G}_1$: This game is similar to the previous game with a difference in the signing oracle before corruption. Upon $\mathcal{A}$ querying the signing oracle on input m and ID and if no corruption query happens during the execution of the signing protocol, the signing oracle proceeds as follows: Instead of computing the signature $\sigma := (R', s')$ as

$$\left\langle \mathsf{Sign}_{\mathcal{S}}(\mathsf{sk}_{\mathcal{S}}, \tau^{\mathsf{reg}}_{\mathcal{S}}), \mathsf{Sign}_{\mathcal{U}}(\mathsf{pk}^{\mathsf{ID}}, \mathsf{sk}^{\mathsf{ID}}_{\mathcal{U}}, \mathsf{pw}, \tau^{\mathsf{reg}}_{\mathcal{U}}, m) \right\rangle,$$

the oracle computes $\tilde{\sigma} := (\tilde{R}', \tilde{s}')$ as $\tilde{R}' := g^{\tilde{r}'}$ for $\tilde{r}' \xleftarrow{\$} \mathbb{Z}_q$ and $\tilde{s}' := \tilde{r}' + \tilde{c}' \cdot (\mathsf{sk}_{\mathcal{U}}^{\mathsf{ID}} + \mathsf{sk}_{\mathcal{S}}) \mod q$ $(= \tilde{r}' + \tilde{c}' \cdot (\mathsf{sk} + \rho) \mod q)$ for $\tilde{c}' := \mathsf{H}(\tilde{R}', \mathsf{pk}^{\mathsf{ID}}, m)$ and $\rho := \mathsf{H}(\mathsf{seed}, \mathsf{ID})$.

It is easy to see that $\tilde{\sigma} := (\tilde{R}', \tilde{s}')$ as computed in this game constitutes a valid signature for message m under public key $\mathsf{pk}^{\mathsf{ID}}$. Therefore, we have that $\Pr[\boldsymbol{G}_1^{\mathcal{A},k} = 1] = \Pr[\boldsymbol{G}_0^{\mathcal{A},k} = 1]$.

Game $\boldsymbol{G}_2$: This game is similar to the previous game with a difference in the signing oracle before corruption. Upon $\mathcal{A}$ querying the signing oracle on input m and ID and if no corruption query happens during the execution of the signing protocol, the signing oracle proceeds as follows: Instead of computing $\tilde{s}'$ of the signature $\tilde{\sigma} := (\tilde{R}', \tilde{s}')$ as $\tilde{s}' := \tilde{r}' + c' \cdot (\mathsf{sk}_{\mathcal{U}}^{\mathsf{ID}} + \mathsf{sk}_{\mathcal{S}}) \mod q$, the oracle computes $\tilde{s}' := \tilde{r}' + c' \cdot (\mathsf{sk}_{\mathcal{U}} + \mathsf{sk}_{\mathcal{S}}) + c' \cdot \rho \mod q$ $(= \tilde{r}' + c' \cdot (\mathsf{sk} + \rho) \mod q)$ for $\rho := \mathsf{H}(\mathsf{seed}, \mathsf{ID})$.

It is easy to see that $\tilde{\sigma} := (\tilde{R}', \tilde{s}')$ again constitutes a valid signature for message m under public key $\mathsf{pk}^{\mathsf{ID}}$. Therefore, we have that $\Pr[\boldsymbol{G}_2^{\mathcal{A},k} = 1] = \Pr[\boldsymbol{G}_1^{\mathcal{A},k} = 1]$.

Game $\boldsymbol{G}_3$: This game is similar to the previous game with the following difference: If a corruption query happens before or during a signing oracle query, the signing oracle proceeds as follows: Upon a query by $\mathcal{A}$ on input m and ID, the signing oracle does not compute the signature $\sigma := (R', s')$ as prescribed by the signing protocol, but first computes $\tilde{R}' := g^{\tilde{r}'}$ for $\tilde{r}' \xleftarrow{\$} \mathbb{Z}_q$ and then sets $\tilde{s}' := \tilde{r}' + \tilde{c}' \cdot (\mathsf{sk}_{\mathcal{U}}^{\mathsf{ID}} + \mathsf{sk}_{\mathcal{S}})$ mod q $(= \tilde{r}' + \tilde{c}' \cdot (\mathsf{sk} + \rho) \mod q)$ for $c' := \mathsf{H}(\tilde{R}', \mathsf{pk}^{\mathsf{ID}}, m)$ and $\rho := \mathsf{H}(\mathsf{seed}, \mathsf{ID})$.

In order to show that this game is indistinguishable from the previous game, we have to show that $\mathcal{A}$ cannot distinguish $\sigma := (R', s')$ as output by an execution of the signing protocol from $\tilde{\sigma} := (\tilde{R}', \tilde{s}')$. Note that $\mathcal{A}$'s view consists of its input to the signing protocol $(\mathsf{pk}, \mathsf{sk}_{\mathcal{S}}, \tau_{\mathcal{S}}^{\mathsf{reg}})$ as well as all messages exchanged during the signing protocol (R, c, ct, s) and the public key $\mathsf{pk}^{\mathsf{ID}}$. Further, note that for signature $\sigma := (R', s')$, we have $R' = g^{r'} = R \cdot g^{\alpha} \cdot {\mathsf{pk}^{\mathsf{ID}}}^{\beta}$, $c' := \mathsf{H}(R', (\mathsf{pk}^{\mathsf{ID}}, m))$ and $s' := s + \alpha + c \cdot \mathsf{sk}_{\mathcal{U}}^{\mathsf{ID}}$ mod $q = s + \alpha + (c' + \beta) \cdot \mathsf{sk}_{\mathcal{U}}^{\mathsf{ID}} \mod q$ where α and β are chosen uniformly at random from $\mathbb{Z}_q$. On the other hand, for $\tilde{\sigma} := (\tilde{R}', \tilde{s}')$ we have $\tilde{R}' := g^{\tilde{r}'}$, $\tilde{c}' := \mathsf{H}(\tilde{R}', \mathsf{pk}^{\mathsf{ID}}, m)$ and $\tilde{s}' := \tilde{r}' + \tilde{c}' \cdot (\mathsf{sk} + \rho) \mod q$ for a uniformly random $\tilde{r}' \xleftarrow{\$} \mathbb{Z}_q$.

We show that there exists a pair $(\tilde{\alpha}, \tilde{\beta}) \in \mathbb{Z}_q \times \mathbb{Z}_q$ such that the following distributions are identical.

$$\{\mathsf{pk}, \mathsf{pk}^{\mathsf{ID}}, \mathsf{sk}_{\mathcal{S}}, \tau_{\mathcal{S}}^{\mathsf{reg}}, R, c, \mathsf{ct}, s, \tilde{\alpha}, \tilde{\beta}, (\tilde{R}', \tilde{s}')\} \text{ and}$$
$$\{\mathsf{pk}, \mathsf{pk}^{\mathsf{ID}}, \mathsf{sk}_{\mathcal{S}}, \tau_{\mathcal{S}}^{\mathsf{reg}}, R, c, \mathsf{ct}, s, \alpha, \beta, (R', s')\}$$

Indeed, since we give an information-theoretic argument, we even show that the distributions

$$\{\mathsf{sk}_{\mathcal{U}}^{\mathsf{ID}}, \mathsf{sk}_{\mathcal{U}}, \mathsf{sk}_{\mathcal{S}}, \tau_{\mathcal{S}}^{\mathsf{reg}}, r, c, \mathsf{ct}, s, \tilde{\alpha}, \tilde{\beta}, (\tilde{r}', \tilde{s}')\} \text{ and}$$
$$\{\mathsf{sk}_{\mathcal{U}}^{\mathsf{ID}}, \mathsf{sk}_{\mathcal{U}}, \mathsf{sk}_{\mathcal{S}}, \tau_{\mathcal{S}}^{\mathsf{reg}}, r, c, \mathsf{ct}, s, \alpha, \beta, (r', s')\}$$

are identical.

Claim. There exists a unique $(\tilde{\alpha}, \tilde{\beta}) \in \mathbb{Z}_q \times \mathbb{Z}_q$ such that the following distributions are identical.

$$\{\mathsf{sk}_{\mathcal{U}}^{\mathsf{ID}}, \mathsf{sk}_{\mathcal{U}}, \mathsf{sk}_{\mathcal{S}}, \tau_{\mathcal{S}}^{\mathsf{reg}}, r, c, \mathsf{ct}, s, \tilde{\alpha}, \tilde{\beta}, (\tilde{r}', \tilde{s}')\} \text{ and}$$
$$\{\mathsf{sk}_{\mathcal{U}}^{\mathsf{ID}}, \mathsf{sk}_{\mathcal{U}}, \mathsf{sk}_{\mathcal{S}}, \tau_{\mathcal{S}}^{\mathsf{reg}}, r, c, \mathsf{ct}, s, \alpha, \beta, (r', s')\}$$

Proof. First note that the values $\mathsf{sk}_{\mathcal{U}}^{\mathsf{ID}}, \mathsf{sk}_{\mathcal{U}}, \mathsf{sk}_{\mathcal{S}}, \tau_{\mathcal{S}}^{\mathsf{reg}}, r, c, \mathsf{ct}$, and s are the same in both distributions and hence identically distributed. Let $\tilde{\beta} := c - \tilde{c}' \mod q$ and $\tilde{\alpha} = \tilde{s}' - s - c \cdot \mathsf{sk}_{\mathcal{U}}^{\mathsf{ID}} \mod q$. Clearly, $\tilde{\alpha}$ and $\tilde{\beta}$ are uniformly random in $\mathbb{Z}_q$ and thereby identically distributed to (α, β). Then it remains to show that the following two conditions hold:

$$\tilde{r}' = r + \tilde{\alpha} + (\mathsf{sk} + \rho) \cdot \tilde{\beta} \mod q \qquad (1)$$
$$\tilde{s}' = s + \tilde{\alpha} + c \cdot \mathsf{sk}_{\mathcal{U}}^{\mathsf{ID}} \mod q \qquad (2)$$

It is easy to see that condition (2) is satisfied. It remains to show that condition (1) holds:

$$\begin{aligned}
&r + \tilde{\alpha} + (\mathsf{sk} + \rho) \cdot \tilde{\beta} \mod q \\
&= r + \tilde{s} - s - c \cdot \mathsf{sk}_{\mathcal{U}}^{\mathsf{ID}} + (\mathsf{sk} + \rho) \cdot \tilde{\beta} \mod q \\
&= \tilde{s} - c \cdot \mathsf{sk}_{\mathcal{S}} - c \cdot \mathsf{sk}_{\mathcal{U}}^{\mathsf{ID}} + (\mathsf{sk} + \rho) \cdot \tilde{\beta} \mod q \\
&= \tilde{s} - c \cdot (\mathsf{sk} + \rho) + \tilde{\beta} \cdot (\mathsf{sk} + \rho) \mod q \\
&= \tilde{s} - c \cdot (\mathsf{sk} + \rho) + (c - \tilde{c}') \cdot (\mathsf{sk} + \rho) \mod q \\
&= \tilde{s}' - c' \cdot (\mathsf{sk} + \rho) \mod q = \tilde{r}'
\end{aligned}$$

Therefore, it holds that the two distributions of Claim 3.3 are identical. □

We have that $\Pr[\boldsymbol{G}_3^{\mathcal{A}} = 1] = \Pr[\boldsymbol{G}_2^{\mathcal{A}} = 1]$.

Game $\boldsymbol{G}_4$: This game is similar to the previous game with the following difference: If a corruption query happens before or during a signing oracle query, the signing oracle proceeds as follows: Upon a query by $\mathcal{A}$ on input m and ID, the signing oracle does not compute $\tilde{s}'$ of the signature $\tilde{\sigma} := (\tilde{R}', \tilde{s}')$ as $\tilde{s}' := \tilde{r}' + \tilde{c}' \cdot (\mathsf{sk}_{\mathcal{U}}^{\mathsf{ID}} + \mathsf{sk}_{\mathcal{S}}) \mod q$, but it computes $\tilde{s}' := \tilde{r}' + \tilde{c}' \cdot (\mathsf{sk}_{\mathcal{U}} + \mathsf{sk}_{\mathcal{S}}) + \tilde{c}' \cdot \rho \mod q$ $(= \tilde{r}' + \tilde{c}' \cdot (\mathsf{sk} + \rho) \mod q)$ for $\rho := \mathsf{H}(\mathsf{seed}, \mathsf{ID})$.

It is easy to see that $(\tilde{R}', \tilde{s}')$ again constitutes a valid signature for message m under public key $\mathsf{pk}^{\mathsf{ID}}$. Therefore, we have that $\Pr[\boldsymbol{G}_4^{\mathcal{A},k} = 1] = \Pr[\boldsymbol{G}_3^{\mathcal{A},k} = 1]$.

Game $\boldsymbol{G}_5$: This game is similar to $\boldsymbol{G}_4$ with a difference in the key derivation oracle. Upon a query on input ID, the game does not compute the user session secret key share $\mathsf{sk}_{\mathcal{U}}^{\mathsf{ID}}$. Instead it stores $(\mathsf{pk}^{\mathsf{ID}}, \cdot)$ in $\mathsf{KeyList}[\mathsf{ID}]$.

Note that the game does not use the user session secret key share in the signing oracle but only the full secret key sk. Therefore, this game is equivalent to the previous game and we have that $\Pr[\boldsymbol{G}_5^{\mathcal{A}} = 1] = \Pr[\boldsymbol{G}_4^{\mathcal{A}} = 1]$.

Therefore it holds that $\Pr[\textbf{s-wunf}_{\mathsf{PADW}}^{\mathcal{A}} = 1] = \Pr[\boldsymbol{G}_0^{\mathcal{A}} = 1] = \Pr[\boldsymbol{G}_5^{\mathcal{A}} = 1]$.

Reduction to the uf-cma security of Schnorr. It remains to show that if adversary $\mathcal{A}$ wins game $\boldsymbol{G}_5^{\mathcal{A}}$ with more than negligible probability, then there exists an adversary $\mathcal{B}$ playing in game $\textbf{uf-cma}_{\mathsf{Schnorr}}^{\mathcal{B}}$ that simulates game $\boldsymbol{G}_5^{\mathcal{A}}$ to $\mathcal{A}$ and uses $\mathcal{A}$'s output to win game $\textbf{uf-cma}_{\mathsf{Schnorr}}^{\mathcal{B}}$ with non-negligible probability. $\mathcal{B}$'s simulation differs from game $\boldsymbol{G}_5^{\mathcal{A}}$ only in the following ways:

1. Instead of generating the public key pk and the user and server secret key shares $(\mathsf{sk}_{\mathcal{U}}, \mathsf{sk}_{\mathcal{S}})$ in the beginning of the game, $\mathcal{B}$ receives a public key $\mathsf{pk}_{\mathcal{B}}$ from game $\textbf{uf-cma}_{\mathsf{Schnorr}}^{\mathcal{B}}$, which it sets to the wallet public key in $\boldsymbol{G}_5^{\mathcal{A}}$. Then $\mathcal{B}$ samples uniformly at random $\mathsf{sk}_{\mathcal{S}} \xleftarrow{\$} \mathbb{Z}_q$.
2. During a signing oracle query from $\mathcal{A}$ on input m and ID, adversary $\mathcal{B}$ does not compute the signature $\tilde{\sigma}$ itself, but queries its own signing oracle on input $(\mathsf{pk}^{\mathsf{ID}}, m)$.
3. $\mathcal{B}$ forwards all random oracle queries to its own oracle.

Clearly, $\mathcal{B}$ perfectly simulates game $\boldsymbol{G}_5^{\mathcal{A}}$. It remains to show that $\mathcal{B}$ can use a valid forgery $(\sigma^*, m^*, \mathsf{ID}^*)$ as output by $\mathcal{A}$ in game $\boldsymbol{G}_5^{\mathcal{A}}$ to win its own game $\textbf{uf-cma}_{\mathsf{Schnorr}}^{\mathcal{B}}$. Upon receiving the forgery, $\mathcal{B}$ does the following. It parses $\sigma^* := (R^*, s^*)$, computes $\rho^* := \mathsf{H}(\mathsf{seed}, \mathsf{ID}^*)$, $\mathsf{pk}^{\mathsf{ID}^*} := \mathsf{pk} \cdot g^{\rho^*}$ and $s_{\mathcal{B}}^* := s^* - c' \cdot \rho^* \mod q$, where $c' := \mathsf{H}(R^*, \mathsf{pk}^{\mathsf{ID}^*}, m^*)$. $\mathcal{B}$ then sets $\sigma_{\mathcal{B}}^* := (R^*, s_{\mathcal{B}}^*)$ and $m_{\mathcal{B}}^* := (\mathsf{pk}^{\mathsf{ID}*}, m^*)$ and outputs the forgery $(\sigma_{\mathcal{B}}^*, m_{\mathcal{B}}^*)$ to its game $\textbf{uf-cma}_{\mathsf{Schnorr}}^{\mathcal{B}}$.

Claim. Let $(\sigma^*, m^*, \mathsf{ID}^*)$ be a valid forgery in game $\boldsymbol{G}_5^{\mathcal{A}}$. Then the tuple $(\sigma_{\mathcal{B}}^*, m_{\mathcal{B}}^*)$ constitutes a valid forgery in game $\textbf{uf-cma}_{\mathsf{Schnorr}}^{\mathcal{B}}$.

The proof of the above claim follows in the same way as the proof of Claim 3.3.

We therefore have: $\Pr[\textbf{s-wunf}_{\mathsf{PADW}}^{\mathcal{A}} = 1] = \Pr[\textbf{uf-cma}_{\mathsf{Schnorr}}^{\mathcal{B}} = 1]$. □

Lemma 4. *The* PADW *scheme from Figures 2 and 3 satisfies blindness.*

Proof. Game $\boldsymbol{G}_0$: This is the original game $\mathbf{wblind}^{\mathcal{A}}_{\mathsf{PADW}}$. We have that $\Pr[\mathbf{wblind}^{\mathcal{A}}_{\mathsf{PADW}} = 1] = \Pr[\boldsymbol{G}^{\mathcal{A}}_0 = 1]$.

Game $\boldsymbol{G}_1$: This game is similar to the previous game with a difference in the challenge oracle. Instead of computing pk_0 and $\mathsf{sk}_{\mathcal{U},0}$ as the output of $\mathsf{PKDer}(\mathsf{pk}, \mathsf{seed}, \mathsf{ID})$ and $\mathsf{SKDer}(\mathsf{sk}_{\mathcal{U}}, \mathsf{seed}, \mathsf{ID})$ respectively, this game computes pk_0 and $\mathsf{sk}_{\mathcal{U},0}$ by first sampling uniformly at random a value $r_0 \xleftarrow{\$} \mathbb{Z}_q$ and then computing $\mathsf{pk}_0 \leftarrow \mathsf{pk} \cdot g^{r_0}$ and $\mathsf{sk}_{\mathcal{U},0} \leftarrow \mathsf{sk}_{\mathcal{U}} + r_0 \mod q$. That is, this game samples r_0 to compute pk_0 and $\mathsf{sk}_{\mathcal{U},0}$ instead of using the deterministically computed randomness $\rho \leftarrow \mathsf{H}(\mathsf{seed}, \mathsf{ID})$.

Since we model H as a random oracle, the adversary $\mathcal{A}$ can distinguish this game from the previous one only if it issues a query to the random oracle H on input $(\mathsf{seed}, \mathsf{ID})$.

Claim. Let E be the event that $\mathcal{A}$ issues a random oracle query on input $(\mathsf{seed}, \mathsf{ID})$. Then we have that $\Pr[\mathsf{E}] \leq \mathsf{negl}(\kappa)$, where negl is a negligible function in the security parameter κ.

Proof. Since seed is a uniformly random κ-bit string and since $\mathcal{A}$ can issue at most polynomially many (in κ) random oracle queries, the probability for event E to occur is negligible. □

We therefore have $\Pr[\boldsymbol{G}^{\mathcal{A}}_0 = 1] \leq \Pr[\boldsymbol{G}^{\mathcal{A}}_1 = 1] + \mathsf{negl}(\kappa)$, where negl is a negligible function in κ.

Game $\boldsymbol{G}_2$: This game proceeds in a similar way as the previous game with only one difference in the challenge oracle. Instead of computing the signature σ_1 as

$$\left\langle \mathsf{Sign}_{\mathcal{S}}(\mathsf{sk}_{\mathcal{S},1}, \tau^{\mathsf{reg}}_{\mathcal{S},1}), \mathsf{Sign}_{\mathcal{U}}(\mathsf{pk}_1, \mathsf{sk}_{\mathcal{U},1}, \mathsf{pw}_1, \tau^{\mathsf{reg}}_{\mathcal{U},1}, m_{1-\delta}) \right\rangle,$$

this game computes σ_1 as

$$\left\langle \mathsf{Sign}_{\mathcal{S}}(\mathsf{sk}_{\mathcal{S},1}, \tau^{\mathsf{reg}}_{\mathcal{S}}), \mathsf{Sign}_{\mathcal{U}}(\mathsf{pk}_1, \mathsf{sk}_{\mathcal{U},1}, \mathsf{pw}, \tau^{\mathsf{reg}}_{\mathcal{U}}, m_{1-\delta}) \right\rangle.$$

Note that σ_1 is a valid signature under pk_1 for message $m_{1-\delta}$ and independent of the values $\tau_{\mathsf{reg},1}, \mathsf{pk}_{\mathsf{PKE},1}$, pw_1, ϕ_1 and pw_1. We therefore have $\Pr[\boldsymbol{G}^{\mathcal{A}}_2 = 1] = \Pr[\boldsymbol{G}^{\mathcal{A}}_1 = 1]$.

Game $\boldsymbol{G}_3$: This game proceeds in a similar way as the previous game with only one difference in the challenge oracle. Instead of generating fresh setup values $(\mathsf{pk}_1, \mathsf{sk}_{\mathcal{U},1}, \mathsf{sk}_{\mathcal{S},1})$, seed_1, pw_1, and $(\tau^{\mathsf{reg}}_{\mathcal{S},1}, \tau^{\mathsf{reg}}_{\mathcal{U},1})$, the game samples a uniformly random element $r_1 \xleftarrow{\$} \mathbb{Z}_q$ and computes $\mathsf{pk}_1 \leftarrow \mathsf{pk} \cdot g^{r_1}$ and $\mathsf{sk}_{\mathcal{U},1} \leftarrow \mathsf{sk}_{\mathcal{U}} + r_1 \mod q$.

Note that, since r_1 is chosen uniformly at random in game $\boldsymbol{G}_3$, pk_1 is indistinguishable from a freshly generated public key. We therefore have $\Pr[\boldsymbol{G}^{\mathcal{A}}_3 = 1] = \Pr[\boldsymbol{G}^{\mathcal{A}}_2 = 1]$.

Consider the view of the adversary which consists of $\mathcal{A}$'s input pk, the values $\mathcal{A}$ receives upon querying the corruption oracle $(\mathsf{sk}_{\mathcal{S}}, \tau^{\mathsf{reg}}_{\mathcal{S}})$, the output of the challenge oracle $(\mathsf{pk}_b, \sigma_b)$ with $\sigma_b := (R'_b, s'_b)$, and the messages exchanged during the generation of signature σ_0, namely $(R, \mathsf{ID}, c, \mathsf{ct}, s)$ (we omit the view obtained from the signing and key derivation oracles since it is independent of bit b). Then, we can show in the same way as in the proof of Claim 3.3 that there exists a tuple $(\tilde{\alpha}, \tilde{\beta}) \in \mathbb{Z}_q \times \mathbb{Z}_q$ such that the following distributions are identical.

$$\begin{aligned}&\{\mathsf{pk}, \mathsf{pk}_0, \mathsf{sk}_{\mathcal{S}}, \tau^{\mathsf{reg}}_{\mathcal{S}}, R, \mathsf{ID}, c, \mathsf{ct}, s, \alpha, \beta, (R'_0, s'_0)\} \text{ and}\\ &\{\mathsf{pk}, \mathsf{pk}_1, \mathsf{sk}_{\mathcal{S}}, \tau^{\mathsf{reg}}_{\mathcal{S}}, R, \mathsf{ID}, c, \mathsf{ct}, s, \tilde{\alpha}, \tilde{\beta}, (R'_1, s'_1)\}\end{aligned}$$

Namely, let $\tilde{\beta} := c - c' \mod q$ where $c' := \mathsf{H}(R'_1, m_{1-\delta})$ and let $\tilde{\alpha} := s'_1 - s - c \cdot \mathsf{sk}^{\mathsf{ID}}_{\mathcal{U}} \mod q$. Then $(\tilde{\alpha}, \tilde{\beta})$ are distributed uniformly at random in $\mathbb{Z}^2_q$ and it holds that $R'_1 = R \cdot g^{\tilde{\alpha}} \cdot \mathsf{pk}^{\mathsf{ID}^{\tilde{\beta}}}$ and $s'_1 = s + \tilde{\alpha} + c \cdot \mathsf{sk}^{\mathsf{ID}}_{\mathcal{U}} \mod q$. Finally, it holds that the public keys pk_0 and pk_1 are identically distributed.

Hence, $\mathcal{A}$ learns no information about bit b from its view in game $\boldsymbol{G}^{\mathcal{A}}_3$ and we have that $\Pr[\boldsymbol{G}^{\mathcal{A}}_3 = 1] = \frac{1}{2}$.

Overall, we get that $\Pr[\mathbf{wblind}^{\mathcal{A}}_{\mathsf{PADW}} = 1] \leq \Pr[\boldsymbol{G}^{\mathcal{A}}_3 = 1] + \mathsf{negl}(\kappa) = \frac{1}{2} + \mathsf{negl}(\kappa)$.

□

4 Extensions and Limitations of our Work

4.1 Thresholdizing the Server

In our work, we consider the single server setting, where one server stores the entire server secret key share for all registered users. While this setting allows for an efficient construction of a PADW scheme, it might make the server an attractive target of attackers in practice. Attacks such as distributed denial of service (DDoS) might crash the server, which would lead to users being unable to generate signatures and thereby spending their funds unless they have a back-up. Worse yet, corruption of the server would essentially reveal the server secret key shares of all registered users at once. This would allow an adversary to then target individual users to learn their entire secret key.

A common way to avoid these issues is to distribute the server, i.e., to share the computation of the server among multiple devices while each device stores only a *share* of the server's secret key share. This approach has the advantage that the crash or even the corruption of a subset of server devices does not lead to a breakdown or corruption of the entire system.

In order to decentralize the server in our PADW scheme, we essentially require two building blocks: (1) a threshold signature scheme, and (2) a password-authentication mechanism in the threshold setting:

1. A (t, n)-threshold signature scheme allows to distribute the signature generation process among a set of n parties where each party stores a share of the signing secret key. Any subset of at least $t + 1$ parties can then jointly generate a valid signature, while any subset of $\leq t$ corrupted parties cannot forge a signature. In recent years, motivated by its use in cryptocurrency networks, several efficient threshold Schnorr schemes have been proposed (e.g., [28, 30]) which could potentially be used to thresholdize the server in our solution. While traditionally most such schemes only focus on security against static adversaries, the interest has recently increasingly shifted towards investigating adaptively secure constructions (e.g., [13]).
2. We require a mechanism that allows the user to *efficiently* authenticate itself to a set of servers. Several cryptographic building blocks could be considered for such a threshold authentication mechanism, such as threshold password-authenticated key exchange[12] [33], password-protected secret sharing [3], or password-based threshold authentication [1]. Importantly, the authentication mechanism should require only minimal amount of interaction between the user and the servers.

We regard the extension of our work to the threshold setting as an interesting open research question.

4.2 Key Backup

Our PADW scheme is a 2-party scheme, i.e., both parties must interact with each other to generate a signature. While this allows to provide strong security guarantees (no single party can forge a signature), it can be problematic in terms of availability. More concretely, in case of a corrupted or crashed server, the server might turn unresponsive leaving the user unable to generate signatures. We note that this is a general problem of 2-party schemes. In the context of cryptographic wallets, however, this can have devastating consequences, as the user may no longer be able to spend its funds. In our schemes, we can mitigate this risk since the user chooses the entire initial key pair (i.e., public key and user/server secret key shares) during the setup phase. That is, the user can simply create a backup of the server secret key share and store it, e.g., on a secure offline device. A popular technique in practice is to deterministically generate the initial key pair from a seed, which can be used in case of key loss to recover the initial key pair. This technique is widely used in practice and has been standardized by the Bitcoin Improvement Proposal 39 [35].

4.3 Limitations

Similarly to the work of Camenisch et al. [8], our work crucially relies on the assumption that once the user device is corrupted, the user will stop inserting the password into the device. This assumption is reasonable considering scenarios where (1) the user's device is running an intrusion detection system

[12] Similarly to the work of Xu and Sandhu [46] in which the authors build a server-assisted threshold signature scheme from threshold password-authenticated key exchange.

which can reliably detect malware, or (2) an adversary steals the user's device, i.e., the user does not have physical access to its device anymore. However, any malware that runs on the user's device undetected and that can read the device's entire storage (in particular extract the user secret key share and the password), can break the security of our construction. We believe that this issue is difficult to overcome, because, by definition, any party that knows the user password, secret key share, and salt should be able to generate signatures on arbitrary messages. In practice, we could address this issue, e.g., by inserting the password on a separate device that does not store the user secret key share, or by using a two-factor authentication mechanism, where the user must prove that it is in possession of a second device.

Acknowledgments. This work has been funded by the German Research Foundation (DFG) CRC 1119 CROSSING (project S7), by the German Federal Ministry of Education and Research and the Hessen State Ministry for Higher Education, Research and the Arts within their joint support of the National Research Center for Applied Cybersecurity ATHENE, and by the ERC Grant 101044770 (CRYPTO-LAYER).

References

[1] S. Agrawal et al. "PASTA: PASsword-based Threshold Authentication". In: 2018, pp. 2042–2059. DOI: 10.1145/3243734.3243839.

[2] N. Alkeilani Alkadri et al. "Deterministic Wallets in a Quantum World". In: 2020, pp. 1017–1031. DOI: 10.1145/3372297.3423361.

[3] A. Bagherzandi et al. "Password-protected secret sharing". In: 2011, pp. 433–444. DOI: 10.1145/2046707.2046758.

[4] F. Benhamouda et al. "On the (in)security of ROS". In: *Advances in Cryptology – EUROCRYPT 2021*. Ed. by A. Canteaut and F.-X. Standaert. Cham: Springer International Publishing, 2021, pp. 33–53. ISBN: 978-3-030-77870-5.

[5] A. Berwick. *Exclusive: At least $1 billion of client funds missing at failed crypto firm FTX*. https://www.reuters.com/markets/currencies/exclusive-least-1-billion-client-funds-missing-failed-crypto-firm-ftx-sources-2022-11-12/. 2022.

[6] J. Brost et al. "Threshold Password-Hardened Encryption Services". In: 2020, pp. 409–424. DOI: 10.1145/3372297.3417266.

[7] J. Camenisch et al. "Memento: How to Reconstruct Your Secrets from a Single Password in a Hostile Environment". In: 2014, pp. 256–275. DOI: 10.1007/978-3-662-44381-1_15.

[8] J. Camenisch et al. "Virtual Smart Cards: How to Sign with a Password and a Server". In: 2016, pp. 353–371. DOI: 10.1007/978-3-319-44618-9_19.

[9] M. Chase et al. *Acsesor: A New Framework for Auditable Custodial Secret Storage and Recovery*. Cryptology ePrint Archive, Paper 2022/1729. https://eprint.iacr.org/2022/1729. 2022. URL: https://eprint.iacr.org/2022/1729.

[10] D. Chaum and T. P. Pedersen. "Wallet Databases with Observers". In: 1993, pp. 89–105. DOI: 10.1007/3-540-48071-4_7.

[11] Coindesk. *Crypto Wallet BitKeep Hacked for $1M in BNB Chain, Polygon Tokens*. https://www.coindesk.com/markets/2022/10/18/crypto-wallet-bitkeep-hacked-for-1m-in-bnb-chain-polygon-tokens/. 2022.

[12] Cointelegraph. *Slope wallets blamed for Solana-based wallet attack*. https://cointelegraph.com/news/slope-wallets-blamed-for-solana-based-wallet-attack. 2022.

[13] E. Crites et al. "Fully Adaptive Schnorr Threshold Signatures". In: *Advances in Cryptology – CRYPTO 2023*. Ed. by H. Handschuh and A. Lysyanskaya. Springer, 2023.

[14] P. Das et al. "A Formal Treatment of Deterministic Wallets". In: 2019, pp. 651–668. DOI: 10.1145/3319535.3354236.

[15] P. Das et al. "The Exact Security of BIP32 Wallets". In: *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*. CCS '21. ACM, 2021. ISBN: 9781450384544. DOI: 10.1145/3460120.3484807. URL: https://doi.org/10.1145/3460120.3484807.

[16] A. Erwig and S. Riahi. "Deterministic Wallets for Adaptor Signatures". In: *Computer Security – ESORICS 2022*. Ed. by V. Atluri et al. Cham: Springer Nature Switzerland, 2022, pp. 487–506. ISBN: 978-3-031-17146-8.

[17] *FTX recovers $5bn but scale of losses to customers still unknown.* https://www.theguardian.com/business/2023/jan/11/ftx-fraud-value-crypto-sbf. 2023.

[18] G. Fuchsbauer and M. Wolf. *(Concurrently Secure) Blind Schnorr from Schnorr.* Cryptology ePrint Archive, Paper 2022/1676. https://eprint.iacr.org/2022/1676. 2022. URL: https://eprint.iacr.org/2022/1676.

[19] G. Fuchsbauer et al. "Blind Schnorr Signatures and Signed ElGamal Encryption in the Algebraic Group Model". In: 2020, pp. 63–95. DOI: 10.1007/978-3-030-45724-2_3.

[20] R. Ganesan. "Yaksha: augmenting Kerberos with public key cryptography". In: *NDSS 1995, San Diego, California, USA.* Ed. by J. T. Ellis et al. IEEE Computer Society. DOI: 10.1109/NDSS.1995.390639. URL: https://doi.org/10.1109/NDSS.1995.390639.

[21] K. Gjøsteen. "Partially blind password-based signatures using elliptic curves". In: *IACR Cryptol. ePrint Arch.* 2013 (2013), p. 472.

[22] K. Gjøsteen and Ø. Thuen. "Password-Based Signatures". In: *Public Key Infrastructures, Services and Applications.* Springer, 2012.

[23] K. Griffith. *REVEALED: FTX lost a staggering $8.8 BILLION in customer funds - after Alameda hedge fund 'borrowed' $9.3B and the crypto exchange lost $432M in 'unauthorized transactions'.* https://www.dailymail.co.uk/news/article-11816983/FTX-lost-staggering-8-8B-customer-funds-auditors-say.html. 2023.

[24] G. Gutoski and D. Stebila. "Hierarchical Deterministic Bitcoin Wallets that Tolerate Key Leakage". In: 2015, pp. 497–504. DOI: 10.1007/978-3-662-47854-7_31.

[25] Y.-Z. He et al. "Server-aided digital signature protocol based on password". In: *Proceedings 39th Annual 2005 International Carnahan Conference on Security Technology.* 2005, pp. 89–92. DOI: 10.1109/CCST.2005.1594836.

[26] S. Jarecki et al. "TOPPSS: Cost-Minimal Password-Protected Secret Sharing Based on Threshold OPRF". In: 2017, pp. 39–58. DOI: 10.1007/978-3-319-61204-1_3.

[27] J. Katz et al. "Boosting the Security of Blind Signature Schemes". In: *Advances in Cryptology – ASIACRYPT 2021.* Ed. by M. Tibouchi and H. Wang. Cham: Springer International Publishing, 2021, pp. 468–492. ISBN: 978-3-030-92068-5.

[28] C. Komlo and I. Goldberg. "FROST: Flexible Round-Optimized Schnorr Threshold Signatures". In: *Selected Areas in Cryptography.* Springer, 2021.

[29] Y. Kondi et al. "Refresh When You Wake Up: Proactive Threshold Wallets with Offline Devices". In: *2021 IEEE Symposium on Security and Privacy (SP).* 2021, pp. 608–625. DOI: 10.1109/SP40001.2021.00067.

[30] Y. Lindell. *Simple Three-Round Multiparty Schnorr Signing with Full Simulatability.* Cryptology ePrint Archive, Paper 2022/374. 2022. URL: https://eprint.iacr.org/2022/374.

[31] A. D. Luzio et al. "Arcula: A Secure Hierarchical Deterministic Wallet for Multi-asset Blockchains". In: 2020, pp. 323–343. DOI: 10.1007/978-3-030-65411-5_16.

[32] P. D. MacKenzie and M. K. Reiter. "Networked Cryptographic Devices Resilient to Capture". In: 2001, pp. 12–25. DOI: 10.1109/SECPRI.2001.924284.

[33] P. D. MacKenzie et al. "Threshold Password-Authenticated Key Exchange". In: 2002, pp. 385–400. DOI: 10.1007/3-540-45708-9_25.

[34] A. Marcedone et al. "Minimizing Trust in Hardware Wallets with Two Factor Signatures". In: *Financial Cryptography and Data Security - FC 2019.* Springer, 2019.

[35] M. Palatinus et al. *BIP39 proposal.* https://en.bitcoin.it/wiki/BIP_0039. 2013.

[36] C.-P. Schnorr. "Efficient Identification and Signatures for Smart Cards". In: 1990, pp. 239–252. DOI: 10.1007/0-387-34805-0_22.

[37] C.-P. Schnorr. "Security of Blind Discrete Log Signatures against Interactive Attacks". In: 2001, pp. 1–12.

[38] S. Tessaro and C. Zhu. "Short Pairing-Free Blind Signatures with Exponential Security". In: *Advances in Cryptology – EUROCRYPT 2022.* Ed. by O. Dunkelman and S. Dziembowski. Cham: Springer International Publishing, 2022, pp. 782–811. ISBN: 978-3-031-07085-3.

[39] N. Y. Times. *Binance Blockchain Hit by $570 Million Hack, Exposing Crypto Vulnerabilities.* https://www.nytimes.com/2022/10/07/business/binance-hack.html. 2022.

[40] N. Y. Times. *Cryptocurrency Hardware Wallets Can Get Hacked Too.* https://www.wired.com/story/cryptocurrency-hardware-wallets-can-get-hacked-too/. 2020.

[41] D. Wagner. “A Generalized Birthday Problem”. In: 2002, pp. 288–303. DOI: 10.1007/3-540-45708-9_19.
[42] B. Wiki. *Ledger*. https://en.bitcoinwiki.org/wiki/Ledger.
[43] B. Wiki. *Trezor Wallet*. https://en.bitcoinwiki.org/wiki/Trezor_Wallet.
[44] P. Wuille. *Bitcoin Improvement Proposal 32*. https://github.com/bitcoin/bips/blob/master/bip-0032.mediawiki. 2012.
[45] P. Wuille et al. *Bitcoin Improvement Proposal 340*. https://en.bitcoin.it/wiki/BIP_0340. 2020.
[46] S. Xu and R. Sandhu. “Two Efficient and Provably Secure Schemes for Server-Assisted Threshold Signatures”. In: *Topics in Cryptology — CT-RSA 2003*. Ed. by M. Joye. Springer Berlin Heidelberg, 2003.
[47] Yehuda Lindell. *Cryptography and MPC in Coinbase Wallet as a Service (WaaS)*. https://coinbase.bynder.com/m/687ea39fd77aa80e/original/CB-MPC-Whitepaper.pdf. 2023.

A (Blind) Schnorr Signature Scheme

Since the solution of our work relies on the Schnorr signature scheme, we briefly recall the scheme here. The Schnorr signature scheme is defined w.r.t. message space $\mathcal{M} := \{0,1\}^*$ and w.r.t. a cyclic group $\mathbb{G} = \langle g \rangle$ of prime order q where the discrete logarithm problem in $\mathbb{G}$ is hard. We describe the full scheme in Figure 4.

$\mathsf{KGen}(1^\kappa)$	$\mathsf{Sign}(\mathsf{sk}, m)$	$\mathsf{Verify}(\mathsf{pk}, m, \sigma)$
00 $\mathsf{sk} \xleftarrow{\$} \mathbb{Z}_q$	00 $r \xleftarrow{\$} \mathbb{Z}_q, R \leftarrow g^r$	00 Parse $\sigma := (R, s)$
01 $\mathsf{pk} \leftarrow g^{\mathsf{sk}}$	01 $c \leftarrow \mathsf{H}(R, m)$	01 $c \leftarrow \mathsf{H}(R, m)$
02 Return $(\mathsf{pk}, \mathsf{sk})$	02 $s := r + c \cdot \mathsf{sk} \mod q$	02 If $g^s = R \cdot \mathsf{pk}^c$: Return 1
	03 $\sigma := (R, s)$	03 Return 0
	04 Return σ	

Fig. 4. Schnorr signature scheme $\mathsf{Schnorr}[\mathsf{H}]$ instantiated with a hash function $\mathsf{H} : \{0,1\}^* \to \mathbb{Z}_q$.

For completeness, we also recall the signing protocol of the blind Schnorr scheme in Figure 5.

Signer $\mathcal{S}(\mathsf{sk})$		User $\mathcal{U}(\mathsf{pk}, m)$
$r \xleftarrow{\$} \mathbb{Z}_q$, $R := g^r$		
$\mathsf{st}_{\mathcal{S}} := r$	$\xrightarrow{R}$	
		$\alpha \xleftarrow{\$} \mathbb{Z}_q$, $\beta \xleftarrow{\$} \mathbb{Z}_q$
		$R' := R \cdot g^\alpha \cdot \mathsf{pk}^\beta$
		$c' := \mathsf{H}(R', m)$
		$c := c' + \beta \mod q$
		$\mathsf{st}_{\mathcal{U}} := (\alpha, \beta)$
	$\xleftarrow{c}$	
Parse $\mathsf{st}_{\mathcal{S}} := r$		
$s = r + c \cdot \mathsf{sk}_{\mathcal{S}} \mod q$	$\xrightarrow{s}$	
		If $g^s \neq R \cdot \mathsf{pk}^c$: $\sigma := \perp$
		Parse $\mathsf{st}_{\mathcal{U}} := (\alpha, \beta)$
		$s' := s + \alpha \mod q$
		$R' := R \cdot g^\alpha \cdot \mathsf{pk}^\beta$
		Output $\sigma' := (R', s')$

Fig. 5. Blind Schnorr scheme.