

# Delegatable Anonymous Credentials From Mercurial Signatures With Stronger Privacy

Scott Griffy<sup>1</sup>, Anna Lysyanskaya<sup>1</sup>, Omid Mir<sup>2</sup>, Octavio Pérez Kempner<sup>3</sup>, and Daniel Slamanig<sup>4</sup>

Brown University<sup>1</sup>, AIT Austrian Institute of Technology<sup>2</sup>, NTT Social Informatics Laboratories<sup>3</sup>, Universität der Bundeswehr München<sup>4</sup>

**Abstract.** Delegatable anonymous credentials (DACs) are anonymous credentials that allow a root issuer to delegate their credential-issuing power to secondary issuers who, in turn, can delegate further. This delegation, as well as credential showing, is carried out in a privacy-preserving manner, so that credential recipients and verifiers learn nothing about the issuers on the delegation chain. One particularly efficient approach to constructing DACs is due to Crites and Lysyanskaya (CT-RSA'19), based on mercurial signatures, which is a type of equivalence-class signatures. In contrast to previous approaches, this design is conceptually simple and does not require extensive use of non-interactive zero-knowledge proofs. Unfortunately, the “CL-type” DAC schemes proposed so far have a privacy limitation: if an adversarial issuer (even an honest-but-curious one) was part of an honest user’s delegation chain, the adversary will be able to detect this fact (and identify the specific adversarial issuer) when an honest user shows its credential. This is because underlying mercurial signature schemes allow the owner of a secret key to detect when his key was used in a delegation chain.

In this paper we show that it is possible to construct CL-type DACs that does not suffer from this privacy issue. We give a new mercurial signature scheme that provides adversarial public key class hiding; i.e. even if an adversarial signer participated in the delegation chain, the adversary won’t be able to identify this fact. This is achieved by introducing structured public parameters which for each delegation level, enabling strong privacy features in DAC. Since the setup of these parameters also produces trapdoors that are problematic in privacy applications, we show how to overcome this problem by using techniques from updatable structured reference string in zero-knowledge proof systems (Groth et al. CRYPTO’18).

In addition, we propose a simple way to realize revocation for CL-type DACs via the concept of revocation tokens. While we showcase this approach to revocation using our DAC scheme, it is generic and can be applied to any CL-type DAC system. Revocation is a feature that is largely unexplored and notoriously hard to achieve for DACs. However as it is a vital feature for any anonymous credential system, this can help to make DAC schemes more attractive for practical applications.

**Keywords:** Anonymous credentials · delegatable credentials · mercurial signatures · revocation · public key class-hiding.

## 1 Introduction

Anonymous credentials (ACs) allow an authority (the issuer) to issue user credentials that can then be used for anonymous authentication. This primitive was envisioned by Chaum in [16] and later technically realized by Camenisch and Lysyanskaya in [12]. Importantly, in an AC scheme, a verifier and a user (also called a “credential holder”) engage in a showing (also called a “proof” or “presentation”) which proves to the verifier that the user has a valid credential. The scheme is *anonymous* if a user can show their credential multiple times in an unlinkable fashion. Intuitively, anonymity means that after verifying the credentials of two users, an adversary should not be able to tell if the credentials are both from a single user or from two different users.

Delegatable anonymous credentials (DACs) were introduced by Chase and Lysyanskaya [15]. As the name suggests, DAC schemes allow a root issuer to delegate their credential-issuing power to other “intermediate” issuers. This delegation allows any intermediate issuer to issue credentials on behalf of the root issuer (or possibly, re-delegate their issuing power), creating a *delegation chain* between the root issuer, the intermediate issuers, and the credential holder. Belenkiy, Camenisch, Chase, Kohlweiss, Lysyanskaya and Shacham showed how to realize DACs for arbitrarily long delegation chains [2].

Delegation alleviates the burden on the root issuer without revealing the root issuer’s secrets to any other issuer, similar to a key hierarchy in a public key infrastructure (PKI) system. Unlinkability of DACs ensures the anonymity of credential holders, as well as the anonymity of any issuers who participated in that credential’s delegation chain. The anonymity of intermediate issuers implies that given the showing of two credentials, an adversarial verifier cannot determine if they were issued by the same intermediate issuer or different intermediate issuers. Hiding the intermediate issuer is important for a DAC scheme as revealing the identities of these intermediate issuers might reveal information about the end user. The root issuer is always identified in a showing as the verifier must trust some key for unforgeability.

An important property when practically using AC and DAC is non-transferability. This property ensures that users cannot easily share their credentials with other users. One way of providing this is to ensure that a user cannot share one of her credentials without sharing all of her credentials (and corresponding secrets). This is known as the “all-or-nothing” approach [12] and should disincentivize sharing of credentials by users’ fear of losing control over their credentials. Another feature that is particularly important for the practical application of (delegatable) anonymous credentials (but is unfortunately often neglected) is revocation. It is quite clear that when preserving user’s privacy, standard approaches to revocation known from classical PKI schemes do not work. While there are

various different approaches to revocation of anonymous credentials [7, 9–13, 20], in the delegatable setting this seems much harder to achieve and the topic is largely unexplored [1].

### 1.1 Previous Work on DAC and Motivation for our Work

As in the recent work of Mir, Slamanig, Bauer, and Mayrhofer [28], we are particularly interested on developing practical DAC schemes. For a broader understanding, readers are directed to their comprehensive overview. The first practical DAC was proposed by Camenisch, Drijvers, and Dubovitskaya [8], but unfortunately they do not support an anonymous delegation phase. This, however is a crucial privacy requirement. The DACs by Blömer and Bobolz [4] as well as [28] represent two relevant and efficient DAC candidates as they have anonymous delegations and additionally, compact credentials. Unfortunately, [28] does not provide the important property of non-transferability, and for both [4] and [28] the delegated credential is distributed independently of any of the previous delegators. Consequently, it seems very hard to efficiently achieve revocation of delegators for those schemes.

Crites and Lysyanskaya [18] came up with a simple architecture (which we will call “CL-type DAC”) for delegatable anonymous credentials that uses *mercurial signatures* (MS). These CL-type DACs bring the use of equivalence class signatures, extensively used in anonymous credentials [17, 20, 21, 25, 26, 28], to the DAC setting (with numerous follow up works [19, 27, 29, 31] on various aspects). In CL-type DACs, the “links” in a delegation chain are signatures; this chain includes the root’s signature on the first intermediate issuer’s public key; then for  $i \geq 1$ , the  $i^{th}$  intermediary’s signatures on the  $i + 1^{st}$  intermediary’s public key, and finally the last intermediate issuer’s signature on the credential holder’s public key. In order to ensure unlinkability, mercurial signatures allow randomization of both the signer’s public key to an equivalent but unlinkable public key, and the randomization of the message to an equivalent but unlinkable message. As an example delegation chain in a CL-type DAC, there would be a root (or certification) authority which holds a signing key of an MS scheme and to issue a credential to Alice, the root authority produces a signature  $\sigma_{CA,A}$  on an MS public key  $\mathbf{pk}_A$  of Alice. By demonstrating knowledge of the corresponding secret key to  $\mathbf{pk}_A$  along with the root’s signature on  $\mathbf{pk}_A$ , Alice can authenticate herself. Now if Alice wants to delegate a credential to Bob, she uses her corresponding secret key to produce a signature  $\sigma_{A,B}$  on Bob’s MS public key  $\mathbf{pk}_B$ , where the signature acts as a credential for Bob. Now Bob can authenticate himself by demonstrating knowledge of the corresponding secret key (to  $\mathbf{pk}_B$ ) and showing the signatures from both the root ( $\sigma_{CA,A}$ ) and from Alice ( $\mathbf{pk}_A, \sigma_{A,B}$ ). This principle can be applied to an arbitrarily long delegation chain. Now assume that Bob wants to show in a privacy-preserving way that he has been delegated a credential by CA. He can do this by demonstrating  $(\mathbf{pk}_{CA}, \mathbf{pk}_A', \mathbf{pk}_B'), (\sigma_{CA,A'}, \sigma_{A',B'})$  where  $\mathbf{pk}_A'$  and  $\mathbf{pk}_B'$  are new representatives of the respective key classes (and by the properties of MS, they are unlinkable to

the previous ones) and the signatures  $(\sigma_{CA,A'}, \sigma_{A',B'})$  are adapted to the new messages and public keys respectively (which are similarly unlinkable). In the concrete CL construction [18], the MS scheme works in a bilinear group setting where w.l.o.g. the public key of the CA lives in the second source group,  $\mathbb{G}_2$ , and the public key of Alice in  $\mathbb{G}_1$ . Consequently, since the public keys of the MS scheme on the next level need to live in the message space of the MS scheme of the previous level, one always needs to switch the groups for the MS scheme on every level, which is an important detail to keep in mind.

One important limitation of existing DAC approaches [18,19,28,31], besides not yet supporting revocation, is that they only satisfy a weak notion of privacy. In particular it is not possible to guarantee anonymity even in the case of an honest-but-curious delegator in the credential chain (or when the root authority and a single delegator on the delegation chain collaborate, in the case of [28]). In prior constructions of CL-type credentials [18,19,31] this is because public keys in these constructions are traceable (using the secret key) regardless of how they have been randomized. Thus, a malicious delegator can identify itself on a chain and break anonymity.

When it comes to revocation in DAC, the only work so far is by Acar and Nguyen [1], which is based on the generic DAC template in [2] from randomizable NIZK proofs and in addition uses homomorphic NIZK proofs. While this can be instantiated from the Groth-Sahai [24] proof system, this is not very attractive from a practical perspective due to significant costs. So having practical DACs with revocation is an open problem.

*Consequently, there exists a gap in that we do not have practical DAC schemes with strong privacy guarantees that support practical revocation of delegators. Our aim is to close this gap.*

Our work can be seen as the continuation of the research initiated by Crites and Lysyanskaya in [18], closing these existing gaps for practical DAC schemes. To do this, we create a mercurial signature scheme with a stronger privacy property called adversarial public key class-hiding. An overview of this approach is outlined in the technical overview in Section 1.3. Ensuring that maliciously created public keys in mercurial signatures are not traceable by their owners after being randomized has been an open problem since their introduction in [18]. A very recent work introduced the notion of interactive threshold mercurial signatures [29] to overcome said limitation, but it requires an interactive signing protocol that computes a signature from shares of a secret key that are distributed among parties. While such an approach is satisfactory for anonymous credentials and can also be used to distribute trust of the root authority in DAC schemes, it's unclear how it can be used to efficiently manage delegations. Instead, we introduce structured public parameters which we carefully glue over the delegation levels to enable strong privacy features (and without requiring any interaction). Since the setup of these parameters also produces trapdoors that endanger privacy, we show how to overcome this problem by using techniques well-known from

updatable structured reference string in zero-knowledge proof systems [23]. For revocation in DAC, we introduce a new and practical approach that is applicable to any CL-type DAC scheme.

## 1.2 Our Contributions

Subsequently, we summarize our contributions:

**New mercurial signatures.** First, as our core building block, we define a new flavor of a mercurial signature scheme which satisfies a stronger class hiding property, namely adversarial public-key class hiding. Unlike in the mercurial signature definition of Crites and Lysyanskaya [18], here the adversary comes up with a public key and signs a message of its choice; the challenge for an adversary is to distinguish between a randomized version of his own public key and signature and a fresh, unrelated public key and a signature on the same message under that fresh public key. (As we will see below, adversarial public-key class hiding is sufficient to construct a DAC scheme with strong privacy.) We give a construction of adversarial public-key class-hiding mercurial signatures in the generic group model.

**New technique for revocation in DAC.** We introduce a new revocation approach for DACs. The basic idea is that a revocation authority maintains a public deny list, which verifiers can use to ensure that a credential shown to them does not contain any revoked delegators. Thereby any user who wants to receive a credential or obtain delegation rights (while still supporting later revocation) must first register their key with the revocation authority. This registration is anonymous and neither a verification of the user’s identity is needed nor a proof of knowledge of their key needs to be performed. This gives a simple privacy-preserving way for revocation in DAC, can be used for any CL-type DAC and does so without resorting to heavy tools such as malleable NIZKs as done in [1].

**Model and instantiation of DAC with strong privacy and revocation.** We introduce a conceptually simple model for revocable DAC schemes with strong privacy using game-based security definitions. We believe that this notion is easier to use than the simulation-based security notions provided in [1]. Then, using our mercurial signature scheme with  $\ell = 2$ , we construct a conceptually simple DAC scheme with delegator revocation and without requiring extensive use of zero-knowledge proofs. We stress that this gives a CL-type DAC scheme, where privacy holds even when the adversary is allowed to corrupt the root and all delegators simultaneously.

## 1.3 Technical Overview

Public keys in the mercurial signature from [18] are vectors of group elements from a bilinear group and are constructed as  $\mathbf{pk} = \{\hat{X}_i\}_{i \in [\ell]} = \{\hat{P}^{x_i}\}_{i \in [\ell]}$  where the secret key is  $\mathbf{sk} = \{x_i\}_{i \in [\ell]} \in (\mathbb{Z}_p^*)^\ell$  and  $P$  and  $\hat{P}$  the generators of each

source group. Anyone can randomize a public key  $\mathbf{pk}$  to a new representative of the equivalence class to get  $\mathbf{pk}' = \mathbf{pk}^\rho = \{\hat{X}_i^\rho\}_{i \in [\ell]}$  for  $\rho \in \mathbb{Z}_p$ . Unfortunately, such public keys are immediately recognizable to an adversary who holds the corresponding secret key. For an adversary to recognize a public key, it suffices to exponentiate each element in the public key by the inverse of the corresponding element in the secret key and check that the result has the form:  $\{\hat{P}^\rho\}_{i \in [\ell]}$  (a vector of equal elements).

One approach to overcome the above limitation is to ensure that each element in a public key is computed over a distinct generator of the group where the discrete logarithms between these generators are random and not known. In other words, to add a trusted setup to the scheme:  $\text{Setup}(1^\lambda) \rightarrow \mathbf{pp} = \{\hat{P}_1, \hat{P}_2, \dots, \hat{P}_\ell\}$  where  $\hat{P}_i = \hat{P}^{\hat{b}_i}$  for  $\ell$  randomly sampled  $\hat{b}_i$  scalars in  $\mathbb{Z}_p$ . Consequently, public keys can be computed as  $\{\hat{X}_i = \hat{P}_i^{x_i}\}_{i \in [\ell]}$  and it can be shown that under the DDH assumption an adversary cannot distinguish a randomization of their public key from a freshly sampled key.

This appears promising, especially since these  $\hat{P}_i$  values are all distinct and can be efficiently sampled in the ROM. But, if an honest user receives a public key, it is not immediately clear how to ensure that it wasn't created maliciously so that they are recognizable (e.g., ensure the adversary did not compute the public key independent of the public parameters as  $\mathbf{pk} = \{\hat{P}^{x_i}\}_{i \in [\ell]}$  which would be recognizable). To ensure that maliciously created public keys are computed over these bases without the need for zero knowledge proofs, we add what we call "verification bases" to the public parameters. The verification bases are structured so that they can be paired with the key to prove that it was computed using the trusted public parameters. To accomplish this, we need to expand the size of the public key vectors to double their length ( $2\ell$ ). This extra half of the public key will be the result of exponentiating different bases in the public parameters with the same secret key as in the first half. Specifically, our public parameters (w.r.t. public keys) consist of  $\hat{\mathbf{B}} = \{\hat{B}_i\}_{i \in [2\ell]} = \{\hat{P}^{\hat{b}_1}, \dots, \hat{P}^{\hat{b}_\ell}, \hat{P}^{\hat{b}_1 \hat{d}_1}, \dots, \hat{P}^{\hat{b}_\ell \hat{d}_\ell}\}$  such that  $\text{dlog}_{\hat{B}_i}(\hat{B}_{\ell+i}) = \hat{d}_i$ . We then include the verification bases in the public parameters which are computed as:  $\mathbf{V} = \{V_i\}_{i \in [\ell]} = \{P^{v_1 \hat{d}_1}, \dots, P^{v_\ell \hat{d}_\ell}, P^{v_1}, \dots, P^{v_\ell}\}$  (for randomly sampled scalars,  $\{v_i\}_{i \in [\ell]}$ ) such that  $\forall i \in [\ell], e(V_i, \hat{B}_i) = e(V_{\ell+i}, \hat{B}_{\ell+i})$ . Then, key pairs are computed as  $\mathbf{sk} = \{x_i\}_{i \in [\ell]} \xleftarrow{\$} \mathbb{Z}_p$ ,  $\mathbf{pk} = \{\hat{X}_i\}_{i \in [\ell]} = \{\hat{B}_i^{x_i}\}_{i \in [\ell]} \parallel \{\hat{B}_{\ell+i}^{x_i}\}_{i \in [\ell]}$ . We can see now that  $\forall i \in [\ell], e(V_i, \hat{X}_i) = e(V_{\ell+i}, \hat{X}_{\ell+i})$ . Thus, a verifier can take this length  $2\ell$  public key and use the verification bases ( $\mathbf{V}$ ) to verify it by computing these pairings. If these pairing equations hold, then, because the elements in the upper half of  $\hat{\mathbf{B}}$  are the only elements in the second source group that is exponentiated with  $\hat{d}_i$ , the adversary must have computed the upper half of the public key with these bases. Through a similar argument, the lower half of the public key must be computed over the lower half of the public key bases in the public parameters. Thus, if the pairing equations hold for a public key, then randomizations of that public key are unrecognizable to the adversary. Because

we need correlated randomness in the trapdoors (e.g. both  $V_i$  and  $\hat{B}_{\ell+i}$  are computed using  $\hat{d}_i$ ) we can no longer use the ROM to generate these parameters and instead must use the common reference string (CRS) model.

We quickly run into a second problem as with these modified public keys, correctness falls apart when we attempt to sign messages. In [18], signatures are computed as  $\sigma = (Z, Y, \hat{Y})$  with  $Z = (\prod_{i \in [\ell]} M_i^{x_i})^y$ ,  $Y = P^{1/y}$ , and  $\hat{Y} = \hat{P}^{1/y}$ . Verification is done via a pairing product equation:  $e(Z, \hat{Y}) = \prod_{i \in [\ell]} e(M_i, \hat{X}_i)$ , which will not verify unless the messages are also computed using a compatible setup. Therefore, we expand the message space to vectors of  $2\ell$  elements instead of  $\ell$  elements and include  $\forall i \in [\ell], P_i = P^{\hat{b}_i}$  in the public parameters. Messages then have the form:  $M = \{P^{m_i}\}_{i \in [\ell]} || \{P_i^{m_i}\}_{i \in [\ell]}$  for some vector  $\{m_i\}_{i \in [\ell]} \in \mathbb{Z}_p^\ell$ . In this modified scheme, the structure of  $Y$  and  $\hat{Y}$  remains unchanged, but we then use the upper half of the message vector in our `Sign` function and the lower half of the message vector in our `Verify` function. This modification leads to a correct verification, now given by:  $e(Z, \hat{Y}) = e(P, \hat{P})^{\sum_{i \in [\ell]} m_i x_i \hat{b}_i}$  and  $\prod_{i \in [\ell]} e(M_i, \hat{X}_i) = e(P, \hat{P})^{\sum_{i \in [\ell]} m_i x_i \hat{b}_i}$ . We also have to add more structure to achieve a signature that yields DAC as the lower half of the message (which will be another public key in a DAC credential chain) is recognizable. We add this extra structure in Section 3.

**Constructing a strongly private DAC.** As previously mentioned, [18] works by alternating the use of two signature schemes so that even delegation levels are signed with one of the schemes and the odd ones with the other. This way, the message space in one of the schemes is the public key space of the other.

Our approach is to build a structured random string so that each scheme can sign public keys for the next level of the credential chain using unique blinding factors taken from the CRS for each level. This poses a challenge as we need to correlate the structure of both schemes for messages and public keys. To this end, the elements used in the CRS for each level  $L > 1$  is twice the size of  $\ell$ , the length of messages/public keys. Fortunately, for CL-type DACs  $\ell = 2$  and typical applications that use delegation do not require long delegation chains (e.g., driving licenses or official identity documents), making this approach entirely practical. To illustrate it, we consider a DAC scheme for  $L = 3$ . We can generate the public parameters,  $\mathbf{pp} = \{P^{\hat{b}_0}, P^{\hat{b}_1}\}$ ,  $\mathbf{pp}' = \{\hat{P}^{\hat{b}'_0}, \hat{P}^{\hat{b}'_1}, \hat{P}^{\hat{b}_0 \hat{b}'_0}, \hat{P}^{\hat{b}_1 \hat{b}'_1}\}$ ,  $\mathbf{pp}^* = \{P^{\hat{b}_0^*}, P^{\hat{b}_1^*}, P^{\hat{b}'_0 \hat{b}_0^*}, P^{\hat{b}'_1 \hat{b}_1^*}\}$ . We can see that the bases from  $\mathbf{pp}$  and  $\mathbf{pp}'$  have a structure that satisfies:  $e(P^{\hat{b}_i}, \hat{P}^{\hat{b}'_i}) = e(P, \hat{P}^{\hat{b}_i \hat{b}'_i})$  and similar for  $\mathbf{pp}'$  and  $\mathbf{pp}^*$ . Hence, such public parameters can be used to build public keys for the credential chain as:  $\mathbf{pk} = \{P^{\hat{b}_0 x_0}, P^{\hat{b}_1 x_1}\}$ ,  $\mathbf{pk}' = \{\hat{P}^{\hat{b}'_0 x'_0}, \hat{P}^{\hat{b}'_1 x'_1}, \hat{P}^{\hat{b}_0 \hat{b}'_0 x'_0}, \hat{P}^{\hat{b}_1 \hat{b}'_1 x'_1}\}$ ,  $\mathbf{pk}^* = \{P^{\hat{b}_0^* x_0^*}, P^{\hat{b}_1^* x_1^*}, P^{\hat{b}'_0 \hat{b}_0^* x_0^*}, P^{\hat{b}'_1 \hat{b}_1^* x_1^*}\}$ . It follows from inspection that if we use  $\mathbf{sk} = \{x_0, x_1\}$  to sign the third and fourth elements of  $\mathbf{pk}'$ , the signature will verify using the first and second elements from  $\mathbf{pk}'$ . Similarly, if we use  $\mathbf{sk}' = \{x'_0, x'_1\}$  to sign the third and fourth elements in  $\mathbf{pk}^*$ , the signature will verify under the first half of  $\mathbf{pk}'$  with the first half of  $\mathbf{pk}^*$  as the message. Because these trapdoors are

shared across schemes, we need the security properties of our signature scheme to hold when multiple correlated copies of the scheme are generated. We describe this requirement of our signature scheme further in Sec. 3.3 where we present the above example with more detail in Fig. 5.

**Removing trust in the parameter generation.** As it is apparent from our above discussion, the generation of parameters in setup involves a number of exponents that must not be available to any party. Putting trust in the party running this setup to discard these values is typically not desirable, especially in a DAC setting where there are numerous parties involved. One way to deal with this issue is to run specific multi-party computation protocols to generate the parameters [3,6], e.g., running distributed key generation protocols. In order to avoid interaction among many parties, one particularly appealing approach was proposed by Groth et al. [23] in the context of zk-SNARKs, i.e., so called updatable reference strings. This means that some (potentially malicious) party can generate a reference string and any (potentially malicious) party in the system can update the reference string. Thereby every party outputs a proof that the computation was performed honestly and when the chain of proofs from the generation until the last update of the reference string verifies and at least one of the involved parties is honest, it is ensured that no one knows the trapdoors. Since this process can be done strictly sequentially this seems much more interesting for practical application, as for instance demonstrated by the powers of tau ceremony recently run by Ethereum<sup>1</sup>, with around 95k contributors (cf. [30]). We note that in our case this can be done very efficiently using Fiat-Shamir transformed Schnorr proofs for discrete logarithm relations. In Supplementary Material A.1 we present the concrete relations for the updates. In brief, the costs are  $5\ell$  Pedersen commitments for the trapdoors,  $8\ell$  group elements for the Schnorr proofs for the base group elements in  $\mathbf{pp}$  and  $10\ell$   $\mathbb{Z}_p$  elements for the Schnorr proofs which include the trapdoors. Concretely, for  $\ell = 2$  this amounts to 26 group elements and 20  $\mathbb{Z}_p$  elements per level (where for usual applications  $L \leq 5$ ) being several orders of magnitude smaller than the one run by Ethereum. We also note that the computation and verification of these Schnorr proofs is highly efficient.

**Revoking intermediate issuers and users in a DAC scheme.** Conceptually (but not technically) our approach to revocation shares similarities with verifier local revocation known from group signatures [5]. In particular, to revoke these credentials, we generate revocation tokens that verify with a given public key and can later be provided to a trusted revocation authority (TRA). The TRA adds these tokens to a deny list. To achieve this, the TRA first creates the ephemeral secret and public keys. The TRA then registers the user by signing the user’s public keys with the corresponding ephemeral secret key. This forms a signature chain similar to the mercurial scheme described in [18].

---

<sup>1</sup> <https://github.com/ethereum/kzg-ceremony-specs>



When a user needs to prove their credentials, they present a revocation token for each public key in their chain. Since this revocation method mirrors the credential chain approach in [18], i.e., mercurial signatures on public keys, the tokens can be randomized to maintain user anonymity during the presentation.

To revoke a user or issuer in a credential chain (perhaps if the credential chain is used to perform some illegal action) these revocation tokens can be supplied from the verifier to the TRA who can then look through all the secret ephemeral keys they generated to recognize the credential chain and add the respective ones to a deny list.

Later, any verifier can verify the TRA’s signature in the revocation token and iterate through the deny list, using each secret key to attempt to match each public key in the chain. More specifically, the verifier exponentiates the ephemeral public key in the revocation token with the inverses of the secret keys in the deny list (as described earlier in Sec. 1.3). If a match is found, the verifier can confirm that the credential has been revoked (cf. Sec. 4 for details).

## 2 Background

**Notation.** We use  $[\ell]$  to denote the set,  $\{1, 2, \dots, \ell\}$ . We use the notation  $x \in \text{Func}$  to mean that  $x$  is a possible output of the function,  $\text{Func}$ . When drawing multiple values from a set, we may omit notation for products of sets, e.g.  $(x, y) \in \mathbb{Z}_p$  is the same as  $(x, y) \in (\mathbb{Z}_p)^2$  where only the latter is formally correct. For a map from the set  $Z$  to the set  $S$ ,  $m : Z \rightarrow S$ , we will denote  $m[i] \in S$  as the output of the map in  $S$  with input  $i \in Z$ . We use bold font to denote a vector (e.g.  $\mathbf{V}$ ). For brevity, we will sometimes denote the elements in a vector as  $\mathbf{V} = \{V_i\}_{i \in [\ell]} = \{V_1, \dots, V_\ell\}$ . For a vector,  $\mathbf{V} = \{V_1, \dots, V_\ell\}$ , of group elements, we denote the exponentiation of each element by a scalar ( $\rho \in \mathbb{Z}_p$ ) with the notation:  $\mathbf{V}^\rho = \{V_1^\rho, \dots, V_\ell^\rho\}$ . By  $(m, *) \in S$  we mean there is a tuple in the set  $S$  such that the first element of the tuple is  $m$  and the second element is another value which could be anything.  $\{(m, *) \in S : A(m)\}$  is the set of all tuples in  $S$  with  $m$  as their first element meeting condition  $A$ . We use “wildcards” ( $*$ ) in equations. For example, the equation  $(a, b) = (a', *)$ , holds if  $a = a'$  no matter what the value of  $b$  is. For two distributions,  $A$  and  $B$ , we use the notation,  $A \sim B$ , to denote that they are computationally indistinguishable.

### 2.1 Bilinear Pairings

A bilinear pairing is a set of cyclic groups  $\mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T$  of prime order  $p$ , along with a pairing function,  $e$  where  $e : \mathbb{G}_1 \times \mathbb{G}_2 \rightarrow \mathbb{G}_T$  which preserves structure. We call  $\mathbb{G}_T$  the “target group” and call  $\mathbb{G}_1$  and  $\mathbb{G}_2$  the first and second “source groups” respectively. In this work, we use Type III pairings, which means that there is no efficient, non-trivial homomorphism between  $\mathbb{G}_1$  and  $\mathbb{G}_2$ . The pairing function is efficiently computable and has a bilinearity property such that if  $\langle P \rangle = \mathbb{G}_1$  and  $\langle \hat{P} \rangle = \mathbb{G}_2$ , then for  $a, b \in \mathbb{Z}_p^*$ ,  $e(P^a, \hat{P}^b) = e(P, \hat{P})^{ab}$ . In our

pairing groups, the Diffie-Hellman assumptions hold in the related groups, such that for  $a, b, c \leftarrow \mathbb{Z}_p$ ,  $(P^a, P^b, P^{ab}) \sim (P^a, P^b, P^c)$ . Also, given  $(P^a, P^b)$  it is difficult to compute  $P^{ab}$ . We'll use hats to denote elements in the second source group, e.g.  $\hat{X} \in \mathbb{G}_2, X \in \mathbb{G}_1$ . We also use the generic group model in the bilinear pairing setting [33].

## 2.2 Mercurial Signatures

The original scheme from [18] comprises the following algorithms: **Setup**, **KeyGen**, **Sign**, **Verify**, **ConvertPK**, **ConvertSK**, **ConvertSig**, and **ChangeRep**. The scheme is parametrized by a length,  $\ell$ , which determines the upper bound on the size of messages that can be signed. A mercurial signature scheme is parameterized by equivalence relations for the message, public key, and secret key spaces:  $\mathcal{R}_M$ ,  $\mathcal{R}_{\text{pk}}$ ,  $\mathcal{R}_{\text{sk}}$ . These relations form *equivalence classes* for messages and keys and define exactly how messages and signatures can be randomized such that their corresponding signatures can correctly be updated to verify with the updated keys and messages. Allowing the keys and messages to be randomized is what gives this signature scheme its privacy-preserving properties. In this work, we introduce auxiliary algorithms to verify the correctness of messages and public keys with respect to the scheme parameters. These are **VerifyMsg** and **VerifyKey**, respectively. Thus, the syntax of mercurial signatures used in this work is given by:

- **Setup**( $1^\lambda, 1^\ell$ )  $\rightarrow$  (**pp**,  $td$ ): Outputs public parameters **pp**, including parameterized equivalence relations for the message, public key, and secret key spaces:  $\mathcal{R}_M$ ,  $\mathcal{R}_{\text{pk}}$ ,  $\mathcal{R}_{\text{sk}}$  and the sample space for key and message converters. This function also outputs a trapdoor ( $td$ ) that can be used (in conjunction with the corresponding secret key) to recognize public keys.
- **KeyGen**(**pp**)  $\rightarrow$  (**pk**, **sk**): Generates a key pair.
- **Sign**(**pp**, **sk**,  $M$ )  $\rightarrow$   $\sigma$ : Signs a message  $M$  with the given secret key.
- **Verify**(**pp**, **pk**,  $M$ ,  $\sigma$ )  $\rightarrow$  (0 or 1): Returns 1 iff  $\sigma$  is a valid signature for  $M$  w.r.t. **pk**.
- **ConvertPK**(**pp**, **pk**,  $\rho$ )  $\rightarrow$   $\text{pk}'$ : Given a key converter  $\rho$ , returns  $\text{pk}'$  by randomizing **pk** with  $\rho$ .
- **ConvertSK**(**pp**, **sk**,  $\rho$ )  $\rightarrow$   $\text{sk}'$ : Randomize a secret key such that it now corresponds to a public key which has been randomized with the same  $\rho$  (i.e. signatures from  $\text{sk}' = \text{ConvertSK}(\text{pp}, \text{sk}, \rho)$  verify by the randomized  $\text{pk}' = \text{ConvertPK}(\text{pk}, \rho)$ ).
- **ConvertSig**(**pp**, **pk**,  $M$ ,  $\sigma$ ,  $\rho$ )  $\rightarrow$   $\sigma'$ : Randomize the signature so that it verifies with a randomized  $\text{pk}'$  (which has been randomized with the same  $\rho$ ) and  $M$ , but  $\sigma'$  is otherwise unlinkable to  $\sigma$ .

- **ChangeRep**(pp, pk,  $M, \sigma, \mu$ )  $\rightarrow (M', \sigma')$ : Randomize the message-signature pair such that  $\text{Verify}(\text{pk}, M', \sigma') = 1$  (i.e.,  $\sigma'$  and  $\sigma$  are indistinguishable) where  $M'$  is a new representation of the message equivalence class defined by  $M$ .
- **VerifyKey**(pp, pk)  $\rightarrow 0, 1$ : Takes a public key and verifies if it is well-formed w.r.t the public parameters pp.
- **VerifyMsg**(pp,  $M$ )  $\rightarrow 0, 1$ : Takes a message and verifies if it is well-formed w.r.t the public parameters pp.

In the construction of [18] as well as our construction, relations and equivalence classes for messages, public keys, and secret key are defined as follows:

$$\mathcal{R}_M = \{(M, M') \in (\mathbb{G}_1^*)^\ell \times (\mathbb{G}_1^*)^\ell \mid \exists r \in \mathbb{Z}_p^* \text{ s.t. } M' = M^r\}$$

$$\mathcal{R}_{\text{pk}} = \{(\text{pk}, \text{pk}') \in (\mathbb{G}_1^*)^\ell \times (\mathbb{G}_1^*)^\ell \mid \exists r \in \mathbb{Z}_p^* \text{ s.t. } \text{pk}' = \text{pk}^r\}$$

$$\mathcal{R}_{\text{sk}} = \{(\text{sk}, \text{sk}') \in (\mathbb{Z}_p^*)^\ell \times (\mathbb{Z}_p^*)^\ell \mid \exists r \in \mathbb{Z}_p^* \text{ s.t. } \text{sk}' = r \cdot \text{sk}\}$$

Consequently, equivalence classes are denoted as  $[M]_{\mathcal{R}_M}$ ,  $[\text{pk}]_{\mathcal{R}_{\text{pk}}}$ ,  $[\text{sk}]_{\mathcal{R}_{\text{sk}}}$  for messages, public keys, and secret keys respectively, such that:  $[M]_{\mathcal{R}_M} = \{M' : (M, M') \in \mathcal{R}_M\}$ ,  $[\text{pk}]_{\mathcal{R}_{\text{pk}}} = \{\text{pk}' : (\text{pk}, \text{pk}') \in \mathcal{R}_{\text{pk}}\}$ ,  $[\text{sk}]_{\mathcal{R}_{\text{sk}}} = \{\text{sk}' : (\text{sk}, \text{sk}') \in \mathcal{R}_{\text{sk}}\}$ . Because of the properties of equivalence classes (reflexivity, symmetry, and transitivity), the following relations hold:  $[M]_{\mathcal{R}_M} = [M']_{\mathcal{R}_M}$  iff  $(M, M') \in \mathcal{R}_M$ ,  $[\text{pk}]_{\mathcal{R}_{\text{pk}}} = [\text{pk}']_{\mathcal{R}_{\text{pk}}}$  iff  $(\text{pk}, \text{pk}') \in \mathcal{R}_{\text{pk}}$ , and  $[\text{sk}]_{\mathcal{R}_{\text{sk}}} = [\text{sk}']_{\mathcal{R}_{\text{sk}}}$  iff  $(\text{sk}, \text{sk}') \in \mathcal{R}_{\text{sk}}$ .

Besides the usual notions for correctness and unforgeability, security of mercurial signatures requires message class-hiding, origin-hiding and public key class-hiding. We recall the original definitions and provide some intuition.

**Definition 1 (Correctness [18]).** *A mercurial signature for parameterized equivalence relations,  $\mathcal{R}_M$ ,  $\mathcal{R}_{\text{pk}}$ ,  $\mathcal{R}_{\text{sk}}$ , message randomizer space,  $\text{sample}_\mu$ , and key randomizer space,  $\text{sample}_\rho$ , is correct if for all parameters  $(\lambda, \ell)$ ,  $\forall (\text{pp}, \text{td}) \in \text{Setup}(1^\lambda, 1^\ell)$ , and  $\forall (\text{sk}, \text{pk}) \in \text{KGen}(1^\lambda)$ , the following holds:*

- **Verification.**  $\forall M \in \mathcal{M}, \sigma \in \text{Sign}(\text{sk}, M) : \text{Verify}(\text{pk}, M, \sigma) = 1 \wedge \text{VerifyMsg}(\text{pp}, M) = 1 \wedge \text{VerifyKey}(\text{pp}, \text{pk}) = 1$ .
- **Key conversion.**  $\forall \rho \in \text{sample}_\rho, (\text{ConvertPK}(\text{pk}, \rho), \text{ConvertSK}(\text{sk}, \rho)) \in \text{KGen}(1^\lambda)$ ,  $\text{ConvertSK}(\text{sk}, \rho) \in [\text{sk}]_{\mathcal{R}_{\text{sk}}}$ , and  $\text{ConvertPK}(\text{pk}, \rho) \in [\text{pk}]_{\mathcal{R}_{\text{pk}}}$ .
- **Signature conversion.**  $\forall M \in \mathcal{M}, \sigma, \rho \in \text{sample}_\rho, \sigma', \text{pk}' \text{ s.t. } \text{Verify}(\text{pk}, M, \sigma) = 1$ ,  $\sigma' = \text{ConvertSig}(\text{pk}, M, \sigma, \rho)$ , and  $\text{pk}' = \text{ConvertPK}(\text{pk}, \rho)$ , then  $\text{Verify}(\text{pk}', M, \sigma') = 1$ .
- **Change of message representation.**  $\forall M \in \mathcal{M}, \sigma, \mu \in \text{sample}_\mu, M', \sigma'$  such that  $\text{Verify}(\text{pk}, M, \sigma) = 1$  and  $(M', \sigma') = \text{ChangeRep}(\text{pk}, M, \sigma; \mu)$  then  $\text{Verify}(\text{pk}, M', \sigma') = 1$  and  $M' \in [M]_{\mathcal{R}_M}$ .

Unforgeability rules out forgeries on the same equivalence class for messages that have been queried to the signing oracle and public keys as these “forgeries” are

actually guaranteed to be computable by correctness. Thus, only forgeries on new equivalence classes are accepted as valid forgeries.

**Definition 2 (Unforgeability [18]).** *A mercurial signature scheme for parameterized equivalence relations  $\mathcal{R}_M, \mathcal{R}_{pk}, \mathcal{R}_{sk}$ , is unforgeable if for all parameters  $(\lambda, \ell)$  and all PPT adversaries  $\mathcal{A}$ , having access to a signing oracle, there exists a negligible function  $\text{negl}$  such that:*

$$\Pr \left[ \begin{array}{l} \text{pp} \leftarrow \text{Setup}(1^\lambda, 1^\ell) \\ (\text{pk}, \text{sk}) \leftarrow \text{KeyGen}(\text{pp}) \\ (\text{pk}^*, M^*, \sigma^*) \leftarrow \mathcal{A}^{\text{Sign}(\text{sk}, \cdot)}(\text{pk}) \end{array} \middle| \begin{array}{l} \text{Verify}(\text{pk}^*, M^*, \sigma^*) = 1 \\ \wedge [\text{pk}^*]_{\mathcal{R}_{pk}} = [\text{pk}]_{\mathcal{R}_{pk}} \\ \wedge \forall M \in Q, [M^*]_{\mathcal{R}_M} \neq [M]_{\mathcal{R}_M} \end{array} \right] \leq \text{negl}(\lambda)$$

Where  $Q$  is the list of messages that the adversary queried to the Sign oracle.

Message class-hiding provides unlinkability in the message space, and it's implied by the DDH assumption in the original scheme from [18].

**Definition 3 (Message class-hiding [18]).** *For all  $\lambda, \ell$  and all PPT adversaries  $\mathcal{A}$ , there exists a negligible  $\text{negl}$  such that:*

$$\Pr \left[ \begin{array}{l} \text{pp} \leftarrow \text{Setup}(1^\lambda, 1^\ell) \\ M_1 \leftarrow \mathcal{M}; M_2^0 \leftarrow \mathcal{M}; M_2^1 \leftarrow [M_1]_{\mathcal{R}_M} \\ b \leftarrow_{\$} \{0, 1\}; b' \leftarrow \mathcal{A}(\text{pp}, M_1, M_2^b) \end{array} \middle| b' = b \right] \leq \frac{1}{2} + \text{negl}(\lambda)$$

Importantly, converted signatures should look like freshly computed signatures in the space of all valid ones. This notion is captured with the *origin-hiding* definitions as shown below.

**Definition 4 (Origin-Hiding for ConvertSig [18]).** *A mercurial signature scheme is origin-hiding for ConvertSig if, given any tuple  $(\text{pk}, \sigma, M)$  that verifies, and given a random key randomizer  $\rho$ ,  $\text{ConvertSig}(\sigma, \text{pk}, \rho)$  outputs a new signature  $\sigma'$  such that  $\sigma'$  is a uniformly sampled signature in the set of verifying signatures,  $\{\sigma^* \mid \text{Verify}(\text{ConvertPK}(\text{pk}, \rho), M, \sigma^*) = 1\}$ .*

**Definition 5 (Origin-Hiding for ChangeRep [18]).** *A mercurial signature scheme is origin-hiding for ChangeRep if, given any tuple  $(\text{pk}, \sigma, M)$  that verifies, and given a random message randomizer  $\mu$ ,  $\text{ChangeRep}(\text{pk}, M, \sigma, \mu)$  outputs a new message and signature  $M', \sigma'$  such that  $M'$  is a uniform sampled message in the equivalence class of  $M$ ,  $[M]_{\mathcal{R}_M}$ , and  $\sigma'$  is uniformly sampled verifying signature in the set of verifying signatures for  $M'$ ,  $\{\sigma^* \mid \text{Verify}(\text{pk}, M', \sigma^*) = 1\}$ .*

For anonymous credentials such as the attribute-based credential (ABC) scheme from [21], the notion of message class-hiding is sufficient to provide unlinkability alongside origin-hiding. This is because in AC's the adversary doesn't know the

Diffie-Hellman coefficients of the message vector that is signed to produce a credential (these coefficients are only known to the honest user who created the message). In DAC's schemes from mercurial signatures the messages to be signed are public keys, which may be provided by the adversary. Since the adversary knows the corresponding secret key, achieving a class-hiding notion for public keys is much harder. The definition below only considers honestly generated keys for which the adversary doesn't know the secret key. This is similar to the case of message-class hiding but it clearly restricts applications.

**Definition 6 (Public key class-hiding [18]).** *For all  $\lambda, \ell$  and all PPT adversaries  $\mathcal{A}$ , there exists a negligible  $\text{negl}$  such that:*

$$\Pr \left[ \begin{array}{l} \text{pp} \leftarrow \text{Setup}(1^\lambda, 1^\ell); (\text{pk}_1, \text{sk}_1) \leftarrow \text{KGen}(\text{pp}); \\ (\text{pk}_2^0, \text{sk}_2^0) \leftarrow \text{KGen}(\text{pp}, \ell(\lambda)); \rho \leftarrow_{\$} (\text{pp}); \\ \text{pk}_2^1 = \text{ConvertPK}(\text{pk}_1, \rho); \text{sk}_2^1 = \text{ConvertSK}(\text{sk}_1, \rho); \\ b \leftarrow_{\$} \{0, 1\}; b' \leftarrow \mathcal{A}^{\text{Sign}(\text{sk}_1, \cdot), \text{Sign}(\text{sk}_2^b, \cdot)}(\text{pp}, \text{pk}_1, \text{pk}_2^b) \end{array} \middle| b' = b \right] \leq \frac{1}{2} + \text{negl}(\lambda)$$

**Mercurial Signature Construction CL19 [18].** We review the mercurial signature construction from [18] in Fig. 1, so that the differences are clear when we present our construction in Sec. 3.

<p><math>\text{Setup}(1^\lambda, 1^\ell) \rightarrow (\text{pp})</math>: Generate a description of a cryptographic bilinear pairing, <math>BP</math>.  Output <math>\text{pp} = BP</math>.  <math>\text{KGen}(\text{pp})</math>: Sample <math>\text{sk} = \{x_i\}_{i \in [\ell]} \leftarrow_{\\$} \mathbb{Z}_p</math> and let <math>\text{pk} = \{\hat{X}_i\}_{i \in [\ell]}</math> where <math>\forall i \in [\ell], \hat{X}_i = \hat{P}_i^{x_i}</math>.  <math>\text{Sign}(\text{sk}, M) \rightarrow \sigma</math>: Sample <math>y \leftarrow_{\\$} \mathbb{Z}_p^*</math> and compute a signature: <math>\sigma = (Z = (\prod_{i=1}^{\ell} M_i^{x_i})^y, Y = P^{1/y}, \hat{Y} = \hat{P}^{1/y})</math>.  <math>\text{Verify}(\text{pk}, M, \sigma) \rightarrow (0 \text{ or } 1)</math>: Accept iff <math>\prod_{i=1}^{\ell} e(M_i, \hat{X}_i) = e(Z, \hat{Y})</math>, and <math>e(Y, \hat{P}) = e(P, \hat{Y})</math>.  <math>\text{ConvertSig}(\text{pk}, M, \sigma, \rho) \rightarrow \sigma'</math>: Sample <math>\psi \leftarrow_{\\$} \mathbb{Z}_p</math>. Compute: <math>Z' = Z^{\psi\rho}</math>, <math>Y' = Y^{1/\psi}</math>, and <math>\hat{Y}' = \hat{Y}^{1/\psi}</math>. Output <math>\sigma' = (Z', Y', \hat{Y}')</math>.  <math>\text{ConvertPK}(\text{pk}, \rho) \rightarrow \text{pk}'</math>: Compute: <math>\text{pk}' = \text{pk}^\rho</math>.  <math>\text{ConvertSK}(\text{sk}, \rho) \rightarrow \text{sk}'</math>: Compute: <math>\text{sk}' = \rho\text{sk}</math>.  <math>\text{ChangeRep}(M, \sigma, \mu) \rightarrow (M', \sigma')</math>: Sample <math>\psi \leftarrow_{\\$} \mathbb{Z}_p</math> and compute: <math>\sigma' = (Z' = Z^{\psi\mu}, Y' = Y^{1/\psi}, \hat{Y}' = \hat{Y}^{1/\psi})</math>, valid for <math>M' = M^\mu</math>.</p>
--

**Fig. 1.** CL19 Mercurial Signature Construction [18]

We note that a function (RecognizePK shown in Def. 7) can be added to this scheme to recognize any randomization of a public key given a secret key [22]. We use this in our DAC construction to tell if users have been revoked.

**Definition 7 (Recognize function for CL19 public keys [22]).**

- $\text{RecognizePK}(\text{pp}, \text{sk}, \text{pk}) \rightarrow \{0, 1\}$  Parse  $\text{pk}$  as  $\text{pk} = \{\hat{X}_i\}_{i \in [\ell]}$ . Parse  $\text{sk} = \{x_i\}_{i \in [\ell]}$ . Check if  $\forall i \in [\ell - 1], \hat{X}_i^{x_{i+1}/x_i} = \hat{X}_{i+1}$ .

### 3 New Mercurial Signature Construction

In this section we present our core mercurial signature scheme satisfying a stronger notion of adversarial public key class-hiding (APKCH), which will then build the basis for our DAC construction with strong privacy.

#### 3.1 Modified Security Definitions

Subsequently, we present security definitions for our mercurial signature scheme that are modified (or added) when compared to previous work and before going into details we discuss their generality.

**On the generality of our definitions.** Since our main focus in this work is the construction of DAC, we include in our basic definitions (adversarial public-key class hiding and unforgeability) a “levels” parameter  $L$ , which tells the challenger how many correlated schemes to construct (i.e., how many levels there will be in the delegation chain of the DAC). In our definitions, after receiving the public parameters for every level, the adversary picks a level,  $i$ , and the challenger generates a public key for that level to complete the game with. This allows the reductions in our proofs to ensure that the DAC scheme appears correct to the adversary while reducing APKCH to the anonymity of the DAC scheme. To reduce to unforgeability, we need a similarly modified definition for unforgeability where the challenger generates a number of levels and the adversary chooses which level to continue the game with. Clearly, by setting  $L = i = 1$  we obtain versions of the definitions for a standalone mercurial signature scheme. In our definitions, **Setup** outputs a set of correlated parameters of different signature schemes ( $\text{pp}$ ) and we use  $\text{pp}_i$  to refer to the parameters that define level  $i$ .

First, we formalize the APKCH notion in Def. 8. In this definition, first the challenger generates the public parameters and a public key and gives these to the adversary. The adversary then constructs a message, a key pair and a signature and returns these to the challenger. The challenger then either randomizes the adversary’s public key and signature or creates a new signature on the same message from a randomization of the challenger’s key. The adversary is then challenged to determine if the returned key/signature pair is randomized from their own key/signature that they supplied, or if it is a signature from the randomized challenger key. Looking ahead, this property ensures that in a DAC scheme, an adversary cannot determine if they themselves provided a credential to the prover (user) or if another issuer created the credential.

**Definition 8 (Adversarial public key class-hiding).** *A mercurial signature,  $\Gamma$ , has adversarial public key class-hiding if the advantage of any PPT*

set of algorithms  $\mathcal{A} = \{\mathcal{A}_0, \mathcal{A}_1, \mathcal{A}_2\}$ , the following advantage  $\text{Adv}_{\Gamma, \mathcal{A}}^{\text{APKCH}}(\lambda)$  is negligible,

$$\text{Adv}_{\Gamma, \mathcal{A}}^{\text{APKCH}}(\lambda) := \left| \Pr \left[ \mathbf{Exp}_{\Gamma, \mathcal{A}}^{\text{APKCH}, 0}(\lambda) = 1 \right] - \Pr \left[ \mathbf{Exp}_{\Gamma, \mathcal{A}}^{\text{APKCH}, 1}(\lambda) = 1 \right] \right|$$

where  $\mathbf{Exp}_{\Gamma, \mathcal{A}}^{\text{APKCH}, b}(\lambda)$  is the experiment shown in Figure 2.

```

1: pp ← Setup(1λ, 1ℓ, 1L)
2: (i, st) ← A0(pp)
3: (sk, pk) ← KGen(ppi);
4: (pkA, σA, M, st) ← A1Sign(ppi, sk, ·)(ppi, pk, st)
5: σ ← Sign(ppi, sk, M)
6: ρ0 ←$ ℤp*; pk0 ← ConvertPK(ppi, pk, ρ0);
7: σ0 ← ConvertSig(ppi, σ, ρ0)
8: ρ1 ←$ ℤp*; pk1 ← ConvertPK(ppi, pkA, ρ1);
9: σ1 ← ConvertSig(ppi, σA, ρ1)
10: if Verify(ppi, pkA, σA, M) = 1 ∧ VerifyMsg(ppi, pp, M) = 1 ∧
11:   VerifyKey(ppi, pp, pkA) = 1, return A2Sign(ppi, sk, ·)(ppi, pp, st, pkb, σb)
12: else return A2Sign(ppi, sk, ·)(ppi, pp, st, ⊥, ⊥)

```

**Fig. 2.** Adversarial public key class-hiding experiment  $\mathbf{Exp}_{\Gamma, \mathcal{A}}^{\text{APKCH}, b}(\lambda)$ .

Finally, we provide the unforgeability definition that also considers multiple levels in Def. 9.

**Definition 9 (Unforgeability).** *A mercurial signature scheme for parameterized equivalence relations  $\mathcal{R}_M, \mathcal{R}_{pk}, \mathcal{R}_{sk}$ , is unforgeable if for all parameters  $(\lambda, \ell)$  and all PPT adversaries  $\mathcal{A}$ , having access to a signing oracle, there exists a negligible function  $\text{negl}$  such that:*

$$\Pr \left[ \begin{array}{l} \text{pp} \leftarrow \text{Setup}(1^\lambda, 1^\ell, 1^L) \\ (i, \text{st}) \leftarrow \mathcal{A}_0(\text{pp}) \\ (\text{sk}, \text{pk}) \leftarrow \text{KGen}(\text{pp}_i); \\ (\text{pk}^*, M^*, \sigma^*) \leftarrow \mathcal{A}_1^{\text{Sign}(\text{pp}_i, \text{sk}, \cdot)}(\text{pk}) \end{array} \middle| \begin{array}{l} \text{Verify}(\text{pp}_i, \text{pk}^*, M^*, \sigma^*) = 1 \\ \wedge [\text{pk}^*]_{\mathcal{R}_{\text{pk}}} = [\text{pk}]_{\mathcal{R}_{\text{pk}}} \\ \wedge \forall M \in Q, [M^*]_{\mathcal{R}_M} \neq [M]_{\mathcal{R}_M} \end{array} \right] \leq \text{negl}(\lambda)$$

Where  $Q$  is the list of messages that the adversary queried to the Sign oracle.

### 3.2 Construction

In Fig. 3, we construct a mercurial signature (MS) scheme which in particular provides adversarial public key class-hiding (APKCH) as defined in Def. 8. We fix  $L = 1$  for this construction and explain how we can set correlated parameters while still achieving APKCH in Sec. 3.3.

As in prior MS constructions, our equivalence classes will be parameterized by the public parameters consisting of a description of the bilinear group  $(\mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T, e, p, P, \hat{P})$ . Unlike prior constructions, they will also be parameterized by several length- $2\ell$  vectors that are part of the public parameters of the system. Specifically, the public parameters will include the vectors  $\mathbf{B} = (B_1, \dots, B_{2\ell})$ ,  $\hat{\mathbf{B}} = (\hat{B}_1, \dots, \hat{B}_{2\ell})$ , and  $\hat{\mathbf{V}} = (\hat{V}_1, \dots, \hat{V}_{2\ell})$ ,  $\mathbf{V} = (V_1, \dots, V_{2\ell})$ . These public parameters have trapdoors which include  $\mathbf{b} = (b_1, \dots, b_\ell)$ ,  $\hat{\mathbf{b}} = (\hat{b}_1, \dots, \hat{b}_\ell)$ , and  $\hat{\mathbf{d}} = (\hat{d}_1, \dots, \hat{d}_\ell)$  which are vectors in  $\mathbb{Z}_p^\ell$  and sampled uniformly randomly by **Setup**. The vector  $\mathbf{B}$  will be used to define the message space (and construct messages), while the vector  $\hat{\mathbf{B}}$  defines the public key space and allows users to create valid public keys. These public parameters are structured based on the trapdoors, such that  $\forall i \in [\ell], B_i = P^{b_i}, B_{\ell+i} = B_i^{\hat{b}_i}, \forall i \in [\ell], \hat{B}_i = \hat{P}^{b_i}$  and  $\hat{B}_{\ell+i} = \hat{B}_i^{\hat{d}_i}$ . To verify that keys and messages are computed over these bases, we include the verification bases (shown above as  $\hat{\mathbf{V}}$  and  $\mathbf{V}$ ) in the public parameters which are constructed as:  $\forall i \in [\ell], \hat{V}_i = \hat{P}^{\hat{b}_i b_i}, \hat{V}_{\ell+i} = \hat{P}^{\hat{b}_i \hat{d}_i}, \forall i \in [\ell], V_i = P^{v_i \hat{d}_i}$  and  $V_{\ell+i} = P^{v_i}$ . The vector  $\hat{\mathbf{V}}$  is used to verify messages while the vector  $\mathbf{V}$  is used to verify keys as described in Sec. 1.3. Structuring the parameters in this way ensures that our construction achieves adversarial public key class-hiding as discussed in Sec. 1.3 and defined in Def. 8. Let  $\mathbf{pp}$  denote the public parameters.

Our message space will consist of vectors of  $2\ell$  dimensions over  $\mathbb{G}_1$  that have certain structure determined by  $\mathbf{pp}$ ; i.e., not every  $2\ell$ -dimensional vector will be a valid message. Specifically, our message space,  $\mathcal{M}^{pp, \ell}$  is defined as:

$$\mathcal{M}^{pp, \ell} = \{(M_1, \dots, M_{2\ell}) \mid \exists \mathbf{m} = (m_1, \dots, m_\ell) \in \mathbb{Z}_p^\ell \text{ such that} \\ \forall 1 \leq i \leq \ell \ M_i = B_i^{m_i} \wedge M_{\ell+i} = B_{\ell+i}^{m_i}\}.$$

Note that, using  $\mathbf{pp}$ , it is possible to verify that a  $2\ell$ -dimensional vector is in the message space. In our scheme, the public parameters will include extra bases  $\hat{\mathbf{V}} = \{\hat{V}_1, \dots, \hat{V}_{2\ell}\}$  to pair with messages to verify them, i.e. ensuring that  $e(M_i, \hat{V}_i) = e(M_{\ell+i}, \hat{V}_{\ell+i})$ . Moreover, they include extra bases  $\mathbf{V} = \{V_1, \dots, V_{2\ell}\}$ , to pair with public keys in the same manner, i.e.  $e(V_i, \hat{X}_i) = e(V_{\ell+i}, \hat{X}_{\ell+i})$ . We label messages as  $\mathbf{M} = \{M_1, \dots, M_{2\ell}\}$  and public keys as  $\mathbf{pk} = \{\hat{X}_1, \dots, \hat{X}_{2\ell}\}$ . We are now ready to define our equivalence class over the message space, which is the same as in prior work [18]:

$$R_{\mathcal{M}}^{pp, \ell} = \{(\mathbf{M}, \mathbf{M}') : \exists \mu \in \mathbb{Z}_p \text{ s.t. } \mathbf{M}, \mathbf{M}' \in \mathcal{M}^{pp, \ell} \wedge \mathbf{M}' = \mathbf{M}^\mu\}.$$

Our public key space will be defined similarly to our message space, but defined over vectors  $\hat{\mathbf{B}} = (\hat{P}^{\hat{b}_1}, \dots, \hat{P}^{\hat{b}_\ell}, \hat{P}^{\hat{b}_1 \hat{d}_1}, \dots, \hat{P}^{\hat{b}_\ell \hat{d}_\ell})$  included in  $\mathbf{pp}$  as well. Like messages, our public key space is a strict subset of the space of  $2\ell$ -dimension vectors in  $\mathbb{G}_2^{2\ell}$ , defined below:

$$\mathcal{PK}^{pp, \ell} = \{(\hat{X}_1, \dots, \hat{X}_{2\ell}) \mid \exists \mathbf{x} = (x_1, \dots, x_\ell) \in \mathbb{Z}_p^\ell \text{ such that} \\ \forall 1 \leq i \leq \ell \ \hat{X}_i = \hat{B}_i^{x_i} \wedge \hat{X}_{\ell+i} = \hat{B}_{\ell+i}^{x_i}\}.$$



We define our equivalence classes over the public key space (similarly to [18]):

$$R_{\mathcal{PK}}^{pp,\ell} = \{(\mathbf{pk}, \mathbf{pk}') : \exists \rho \in \mathbb{Z}_p \text{ s.t. } \mathbf{pk}, \mathbf{pk}' \in \mathcal{PK}^{pp,\ell} \wedge \mathbf{pk}' = \rho \mathbf{pk}\}.$$

We will see in the construction that the related structure of these messages and public keys (i.e. the fact that they both use the values  $\mathbf{b}$  and  $\hat{\mathbf{b}}$ ) ensures that randomized keys and signatures are not linkable even when the adversary holds the secret key  $\{x_1, \dots, x_\ell\}$ , while also ensuring that signatures still correctly verify.

Our secret key space takes up the entire space of  $\ell$ -dimensional vectors in  $\mathbf{sk}^{pp,\ell} = (\mathbb{Z}_p)^\ell$  and is defined identically to [18] as:

$$R_{\mathbf{sk}}^{pp,\ell} = \{(\mathbf{sk}, \mathbf{sk}') : \exists \rho \in \mathbb{Z}_p \text{ s.t. } \mathbf{sk}, \mathbf{sk}' \in \mathbf{sk}^{pp,\ell} \wedge \mathbf{sk}' = \rho \mathbf{sk}\}.$$

In the sequel, when clear from the context, we will omit the superscript  $pp, \ell$ . In our construction, the key and message converter spaces are  $\mathcal{KC} \leftarrow_s \mathbb{Z}_p$  and  $\mathcal{MC} \leftarrow_s \mathbb{Z}_p$ .

**Our construction.** We are now ready to present our MS construction in Fig. 3 which achieves our APKCH definition for  $L = 1$ . In Sec. 3.3, we will discuss modifications to the public parameter generation, allowing the scheme's extension for constructing DAC. While verifying the message or public key is not required for unforgeability, we include it in the `Verify` function as it is needed for public key class-hiding and message class-hiding.

**Theorem 1 (Correctness).** *The mercurial signature construction in Fig. 3 is correct as described Def. 1.*

**Theorem 2 (Unforgeability).** *The mercurial signature construction in Fig. 3 meets the unforgeability definition in Def. 2 assuming that the mercurial signature construction in [18] is unforgeable in the generic group model.*

We prove Theorem 2 by noting that the CL19 construction [18] is unforgeable in the generic group model, and thus, by showing that our construction's unforgeability relies solely on the unforgeability of the construction of CL19, our construction is also unforgeable in the generic group model.

**Theorem 3 (APKCH).** *The mercurial signature construction in Fig. 3 meets the APKCH definition in 8 in the generic group model.*

**Theorem 4 (Origin-hiding of signatures).** *The mercurial signature construction in Fig. 3 meets the Origin-hiding of signatures definition in Def. 4.*

**Theorem 5 (Origin-hiding of ChangeRep).** *The mercurial signature construction in Fig. 3 meets the Origin-hiding of ChangeRep definition in Def. 5.*

<p>Setup(<math>1^\lambda, 1^\ell</math>) <math>\rightarrow</math> (pp): Sample <math>\{b_i, \hat{b}_i, \hat{d}_i, \hat{v}_i, v_i\}_{i \in [\ell]} \leftarrow \mathbb{Z}_p</math> and compute</p> <p><math>\mathbf{B} = \{B_i\}_{i \in [2\ell]}</math>, where <math>\forall i \in [\ell], B_i \leftarrow P^{b_i}, B_{\ell+i} \leftarrow P^{b_i \hat{b}_i}</math></p> <p><math>\hat{\mathbf{B}} = \{\hat{B}_i\}_{i \in [2\ell]}</math>, where <math>\forall i \in [\ell], \hat{B}_i \leftarrow \hat{P}^{\hat{b}_i}, \hat{B}_{\ell+i} \leftarrow \hat{P}^{\hat{b}_i \hat{d}_i}</math></p> <p><math>\hat{\mathbf{V}} = \{\hat{V}_i\}_{i \in [2\ell]}</math>, where <math>\forall i \in [\ell], \hat{V}_i \leftarrow \hat{P}^{\hat{v}_i \hat{b}_i}, \hat{V}_{\ell+i} \leftarrow \hat{P}^{\hat{v}_i}</math></p> <p><math>\mathbf{V} = \{V_i\}_{i \in [2\ell]}</math>, where <math>\forall i \in [\ell], V_i \leftarrow P^{v_i \hat{d}_i}, V_{\ell+i} \leftarrow P^{v_i}</math></p> <p>Output: pp = (<math>\mathbf{B}, \hat{\mathbf{B}}, \hat{\mathbf{V}}, \mathbf{V}</math>)</p> <p>KGen(pp): Sample sk = <math>\{x_i\}_{i \in [\ell]} \leftarrow \mathbb{Z}_p</math> and let pk = <math>\{\hat{X}_i\}_{i \in [2\ell]}</math> where <math>\forall i \in [\ell], \hat{X}_i = \hat{B}_i^{x_i}, \hat{X}_{i+\ell} = \hat{B}_{\ell+i}^{x_i}</math>.</p> <p>VerifyKey(pp, pk): Accept iff <math>\forall i \in [\ell], e(V_i, \hat{X}_i) = e(V_{i+\ell}, \hat{X}_{i+\ell})</math>.</p> <p>VerifyMsg(pp, M): Accept iff <math>\forall i \in [\ell], e(M_i, \hat{V}_i) = e(M_{i+\ell}, \hat{V}_{i+\ell})</math>.</p> <p>Sign(pp, sk, M) <math>\rightarrow</math> <math>\sigma</math>: If VerifyMsg(pp, M) = 1, sample <math>y \leftarrow \mathbb{Z}_p^*</math> and compute a signature:</p> <p><math>\sigma = (Z = (\prod_{i=1}^{\ell} M_{\ell+i}^{x_i})^y, Y = P^{1/y}, \hat{Y} = \hat{P}^{1/y})</math>.</p> <p>Verify(pk, M, <math>\sigma</math>) <math>\rightarrow</math> (0 or 1): Accept iff VerifyMsg(pp, M) = 1, VerifyKey(pp, pk) = 1, <math>\prod_{i=1}^{\ell} e(M_i, \hat{X}_i) = e(Z, \hat{Y})</math>, and <math>e(Y, \hat{P}) = e(P, \hat{Y})</math>.</p> <p>ConvertSig(pp, pk, M, <math>\sigma, \rho</math>) <math>\rightarrow</math> <math>\sigma'</math>: Sample <math>\psi \leftarrow \mathbb{Z}_p</math>. Compute: <math>Z' = Z^{\psi \rho}, Y' = Y^{1/\psi}</math> and <math>\hat{Y}' = \hat{Y}^{1/\psi}</math>. Output <math>\sigma' = (Z', Y', \hat{Y}')</math>.</p> <p>ConvertPK(pp, pk, <math>\rho</math>) <math>\rightarrow</math> pk': Compute: <math>\text{pk}' = \text{pk}^\rho</math>.</p> <p>ConvertSK(pp, sk, <math>\rho</math>) <math>\rightarrow</math> sk': Compute: <math>\text{sk}' = \rho \text{sk}</math>.</p> <p>ChangeRep(pp, M, <math>\sigma, \mu</math>) <math>\rightarrow</math> (<math>M', \sigma'</math>): Sample <math>\psi \leftarrow \mathbb{Z}_p</math> and compute: <math>\sigma' = (Z' = Z^{\psi \mu}, Y' = Y^{1/\psi}, \hat{Y}' = \hat{Y}^{1/\psi})</math>, valid for <math>M' = M^\mu</math>.</p>
---

Fig. 3. Our Mercurial Signature Construction.

**Theorem 6 (Message class-hiding).** *The mercurial signature construction in Fig. 3 meets the Message class-hiding definition in Def. 3.*

The proofs of theorems 4, 5, and 6, follow directly from those of CL19, and are thus omitted here. The proofs of unforgeability (Theorem 2) and APKCH (Theorem 3) are provided in the Supplementary Material B.

### 3.3 Extending our Construction to Multiple Levels

In a CL-type DAC scheme, we need chains of public keys that can sign each other. In [18], this is achieved by alternating the source groups of the mercurial signature scheme for each level in the chain. For example, to sign public keys in the highest level of the delegation chain  $L$ , if the public keys in level  $L$  are in source group  $\mathbb{G}_2$ , then, in the level  $L - 1$ , a scheme with public keys in  $\mathbb{G}_1$  will be used to sign the public keys of level  $L$ .

This approach works in [18] because the scheme is symmetric, meaning the public parameters are the same whether public keys are in  $\mathbb{G}_2$  or  $\mathbb{G}_1$ . Unfortunately, our scheme is not symmetrical. When we split our message bases and public key bases into halves,  $\mathbf{B} = \mathbf{B}^l \parallel \mathbf{B}^u$  and  $\hat{\mathbf{B}} = \hat{\mathbf{B}}^l \parallel \hat{\mathbf{B}}^u$ , we can see that the second

half of the bases for messages includes the trapdoors,  $\hat{b}_i$ , being formed as  $\mathbf{B}^u = \{P^{b_i \hat{b}_i}\}_{i \in [\ell]}$ . This trapdoor is included in the first half of the bases for public keys ( $\hat{\mathbf{B}}^1 = \{\hat{P}^{\hat{b}_i}\}_{i \in [\ell]}$ ). But, the upper half of the bases for public keys includes the trapdoors,  $\hat{d}_i$ . This upper half is formed as  $\hat{\mathbf{B}}^u = \{\hat{P}^{\hat{b}_i \hat{d}_i}\}_{i \in [\ell]}$ . The trapdoors  $\hat{d}_i$  are not seen in the lower bases for messages ( $\mathbf{B}^1 = \{P^{b_i}\}_{i \in [\ell]}$ ). Due to this asymmetry, we cannot simply invert the groups to start signing public keys from higher levels. At first glance, it appears we could fix this by setting  $\hat{d}_i = b_i$  (thus allowing messages to be used to sign public keys by computing the signatures on the second half of the public key). Unfortunately, this solution would break the APKCH property of our scheme as computing public keys over  $\hat{P}^{b_i \hat{b}_i}$  permits a recognition attack using the bases  $P^{b_i \hat{b}_i}$ . This forces us to choose a more involved solution.

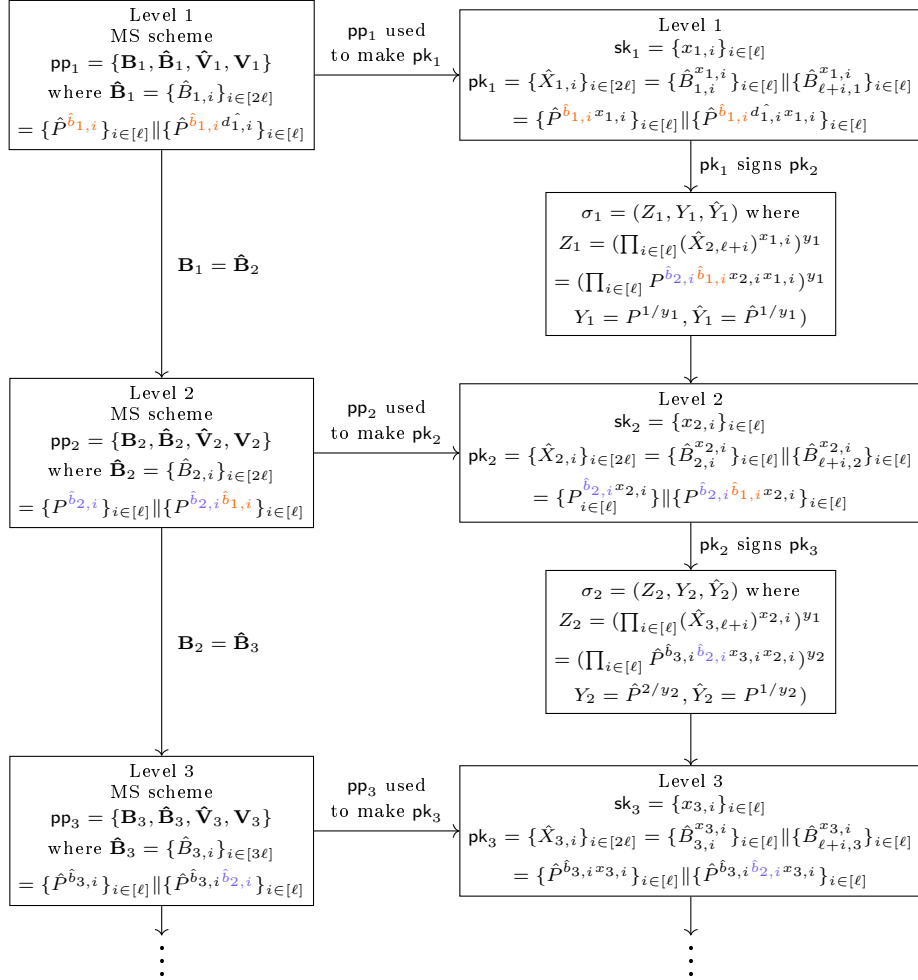
In practice, the parameters for our scheme will be constructed using the functions in Fig. 4. This function generates  $L-1$  sets of correlated public parameters. While each set of public parameters defines one signature scheme, generating each set using the function in Fig. 4 ensures that the message space of each scheme is compatible with the next level, which ensures that DAC can be realized from the set of parameters.

To further exemplify how our multi-level scheme is structured, we show an example delegation chain in Fig. 5. Effectively, the public key space of one scheme becomes the message space of a lower scheme.

**Setup**( $1^\lambda, 1^\ell, 1^L$ )  $\rightarrow$  (pp): Sample  $\{\tilde{b}_{i,j}, v_{i,j}\}_{i \in [\ell], j \in [L]} \leftarrow \mathbb{Z}_p$  and compute the following bases:  
 For  $j \in [L]$ , if  $j \bmod 2 = 0$ ,  $G = \hat{P}, G' = P$  and if  $j \bmod 2 = 1$ ,  $G = P, G' = \hat{P}$ ,  
 $\hat{\mathbf{B}}_j = \{\hat{B}_i\}_{i \in [2\ell]}$ , where  $\forall i \in [\ell], \hat{B}_i \leftarrow G^{\tilde{b}_{i,j}}, \hat{B}_{\ell+i} \leftarrow G^{\tilde{b}_{i,j} \tilde{b}_{i,j-1}}$   
 $\mathbf{V}_j = \{V_i\}_{i \in [2\ell]}$ , where  $\forall i \in [\ell], V_i \leftarrow (G')^{v_{i,j} \tilde{b}_{i,j-1}}, V_{\ell+i} \leftarrow (G')^{v_{i,j}}$   
 For  $j \in [L-1]$ :  $\text{pp}_j = \{\hat{\mathbf{B}}_{j+1}, \hat{\mathbf{B}}_j, \mathbf{V}_{j+1}, \mathbf{V}_j\}$  such that  $\hat{\mathbf{B}}_j, \mathbf{V}_j$  are for keys and  $\hat{\mathbf{B}}_{j+1}, \mathbf{V}_{j+1}$  are for messages.  
 Output  $\text{pp} = \{\text{pp}_j\}_{j \in [L-1]}$ .

**Fig. 4.** Parameter generation for multiple levels

**Proving the multi-level scheme secure** Because a generic group model proof for  $L-1$  sets of public parameters simultaneously might be difficult to comprehend, we instead introduce the notion of “extending” parameters. Intuitively, this ensures that the correlation between the sets of public parameters does not cause any individual scheme to leak information even when its trapdoors are shared with other schemes. We can then write a proof for APKCH against a generic adversary for our scheme in Fig. 3 (for  $L=1$ ) and then use a hybrid argument in the standard model to extend the proof to any polynomial number of levels.



**Fig. 5.** A series of compatible mercurial signature schemes and a credential chain.

When a reduction uses this extended scheme, the resulting parameters will be identically distributed as Fig. 4.

In the following discussion we refer to the keys at the end of delegation chains as “higher-level” keys as the root is at level 0 which is the lowest level of the delegation chain. In order to sign these higher-level public keys with lower-level public keys, starting from our construction in Fig. 3, we need to “extend” the signature scheme so that new public key bases can be used to sign public keys from another scheme. To do this, we need to create a new scheme (with public keys in  $\mathbb{G}_1$ ) with similar structure as in our original scheme in Fig. 3. We recall

that in this scheme the message bases and public key bases share the  $\hat{b}_i$  trapdoor values as described in the above paragraph. When extending our scheme to sign public keys, we'll treat  $\hat{d}_i$  as this shared value, including it in the trapdoor for the new public parameters (remember,  $\hat{d}_i$  is used in the upper half of the public key bases,  $\hat{\mathbf{B}}$  in the public parameters). In order to create security properties independent of the length of the delegation chain, we need to ensure that adversaries can extend the chain while still retaining security properties of the original signature scheme. We could choose to instead directly prove our scheme secure in a setting with multiple levels, but this would complicate the proof.

Unfortunately, we cannot simply reveal  $\hat{d}_i$  to use to extend the scheme as this will break the security properties of our original signature scheme. Specifically, if  $\hat{d}_i$  were revealed, the adversary could compute a public key as  $\mathbf{pk} = \{\hat{P}^{x_i}\}_{i \in [\ell]} \parallel \{\hat{P}^{x_i \hat{d}_i}\}_{i \in [\ell]}$  which would verify with the verification bases, but would easily be recognizable. Thus, instead we reveal  $\mathbf{D} = \{D_i\}_{i \in [\ell]}$  where  $\forall i \in [\ell], D_i = P^{\hat{d}_i}$ . Intuitively, because  $\mathbf{D}$  lives in  $\mathbb{G}_1$  instead of  $\mathbb{G}_2$ , the adversary cannot use it to create malicious public keys that verify. Revealing  $\mathbf{D}$  from the old scheme allows the new scheme to create a new set of public key bases (in  $\mathbb{G}_1$ ) that share the trapdoor from the public key bases in the original scheme. Thus, proving that our scheme is secure when revealing this vector  $\mathbf{D}$  ensures that it is secure for any number of extensions. We prove that our scheme's security holds with the addition in the Supplementary Material B.

Using  $\mathbf{D}$ , the bases for the lower level scheme could be formed as  $\hat{\mathbf{B}}' = \{D_1, \dots, D_\ell, D_{\ell+1}^{\hat{d}'_1}, \dots, D_{2\ell}^{\hat{d}'_1}\}$  where each  $\hat{d}'_i$  is sampled at random by the new scheme. We can see that with this structure, signatures on the higher level public keys from the lower level public keys now verify. For example, let's say we have a higher scheme (with bases  $\hat{\mathbf{B}} = \{\hat{B}_i\}_{i \in [\ell]}$ ) a with key pair  $(\mathbf{sk}, \mathbf{pk})$ , where  $\mathbf{sk} = \{x_i\}_{i \in [\ell]}$ ,  $\mathbf{pk} = \{\hat{X}_i\}_{i \in [\ell]} = \{\hat{B}_i^{x_i}\}_{i \in [\ell]} \parallel \{\hat{B}_{\ell+i}^{x_i}\}_{i \in [\ell]}$ ,  $\forall i \in [\ell], \hat{B}_i = \hat{P}^{\hat{b}_i}, \hat{B}_{\ell+i} = \hat{P}^{\hat{b}_i \hat{d}_i}$  and  $\hat{b}_i$  and  $\hat{d}_i$  are randomly sampled as a trapdoor of the public parameters. We can create a new scheme (with bases  $\hat{\mathbf{B}}' = \{(\hat{B}'_i)\}_{i \in [\ell]}$ ) and key pair  $(\mathbf{sk}', \mathbf{pk}')$  where  $\mathbf{sk}' = \{x'_i\}_{i \in [\ell]}$ ,  $\mathbf{pk}' = \{X'_i\}_{i \in [\ell]} = \{(\hat{B}'_i)^{x'_i}\}_{i \in [\ell]} \parallel \{(\hat{B}'_{\ell+i})^{x'_i}\}_{i \in [\ell]}$  and  $\forall i \in [\ell], (\hat{B}'_i) = D_i, (\hat{B}'_{\ell+i}) = D_i^{\hat{d}'_i}, D_i = P^{\hat{d}_i}$  and  $\hat{d}'_i$  is randomly sampled as a trapdoor of the public parameters. We can see that if we form signatures as we did in Fig. 3, these signatures still verify. In Fig. 3, (if we swap the source groups) signatures are formed as  $\sigma = (Z, Y, \hat{Y})$  where  $Z = (\prod_{i \in [\ell]} (\hat{X}_{\ell+i})^{x'_i})^y$ ,  $Y = \hat{P}^{1/y}$ ,  $\hat{Y} = P^{1/y}$  and  $y$  is randomly sampled. We see that  $e(\hat{Y}, Z) = e(P, \hat{P})^{\sum_{i \in [\ell]} \hat{d}_i \hat{b}_i x_i x'_i} = \prod_{i \in [\ell]} e(\hat{X}_i, X_i)$  which means that this signature verifies. To ease this difficult concept, we provide a diagram in Fig. 5. For example, in Fig. 5, we can see that when the lower level bases of levels 1 and 2 are paired together, they are equal to the pairing of the higher level bases of level 2, i.e.  $e(\hat{B}_{2,i}, \hat{B}_{1,i}) = e(P^{\hat{b}_{2,i}}, \hat{P}^{\hat{b}_{1,i}}) = e(P, \hat{P})^{\hat{b}_{2,i} \hat{b}_{1,i}} = e(\hat{B}_{2,\ell+i}, \hat{P})$ . This structure is what ensures that our signatures verify. We've pointed out this structure by highlighting  $\hat{b}_{1,i}$  in orange and  $\hat{b}_{2,i}$

in blue. This relation holds for levels 2 and 3 as well. Note that in Fig. 5, the source groups for level 2 are flipped, meaning that the  $\hat{B}_{2,i}$  elements are in  $\mathbb{G}_1$  and the  $Z_2$  element is in  $\mathbb{G}_2$ . For clarity, we keep the notation of the generators,  $P$  and  $\hat{P}$ , correct with respect to levels 1 and 3.

While a real setup will not reveal  $\mathbf{D}$  to an adversary, it is important that the signatures retain their security properties even when this is revealed. Intuitively, this is because schemes built on top of a signature scheme requires some correlated structure. Revealing  $P^{\hat{d}_i}$  in our security games ensures that this correlated structure cannot be leveraged to defeat the security of the higher scheme. We describe the formalization of extendable mercurial signatures in the Supplementary Material A.

## 4 Delegatable Anonymous Credentials

In this section, we introduce a new DAC construction, showcasing advanced features as strengthened privacy, revocation capabilities, and non-transferability, all while preserving efficiency.

### 4.1 DAC Functionality

In contrast to non-revocable DAC schemes, our approach integrates a Trusted Revocation Authority (TRA) to efficiently revoke malicious users and maintain a deny list. We outline the high-level functionality of our DAC scheme in Def. 10. This consists of the algorithms: **Setup** which initializes the scheme, **TKeyGen** which generates the TRA’s keys, **RootKeyGen** which generates the root’s keys, **UKeyGen** which generates a user’s or issuer’s keys, **RegisterUser** which allows the TRA to distribute revocation tokens, and **RevokeUser** which allows the TRA to revoke users. The scheme also consists of the interactive protocols to issue and show credentials: (**Issue**  $\leftrightarrow$  **Receive**) and (**Prove**  $\leftrightarrow$  **Verify**). This scheme begins with a trusted **Setup**<sup>2</sup>. Then, the TRA generates an opener secret key and public key using **TKeyGen**. A root authority (who can be malicious for the sake of anonymity) generates the root key using **RootKeyGen** and distributes the root public key to users. A user who wishes to receive a credential runs **UKeyGen** and then interacts with the TRA to receive a revocation token by providing their public key to the TRA so that the TRA can run **RegisterUser** on it. Subsequently the user interacts with the root (or an issuer that the root has delegated to) using (**Issue**  $\leftrightarrow$  **Receive**) and receives a credential (which includes their revocation token). The user then uses their credential and secret key in an interactive protocol (**Prove**  $\leftrightarrow$  **Verify**) with any verifier. These verifications can occur at any level within  $[L]$  (i.e. for some level,  $L'$  such that  $L' \leq L$ ). The verifier can check if the user has been authorized and has not been revoked using the

<sup>2</sup> Note that as already discussed in practice this can be done by multiple parties in a sequential way by using ideas from updatable common reference strings and only a single party among the set of all parties needs to be trusted.

TRA's public key  $tpk$ . More specifically, the verifier receives a revocation token for each level in the credential chain from the credential presentation. If the verifier discovers that the prover was malicious through an out-of-band method, they can submit these tokens to the TRA. The TRA will then update their deny list (this deny list is included in the TRA public key for the sake of simplifying the presentation), preventing any future showings that include the user or issuer corresponding to the revocation token from being verified.

We note that our scheme supports a strong model for anonymity where the holder of the root key cannot de-anonymize users. To model this, we allow the adversary to choose the root public key along with the corruption of any users of their choice, representing a delegation path from a (corrupted) root credential where delegations can be performed maliciously, in the privacy game.

**Definition 10 (Delegatable Anonymous Credentials).** *A DAC includes the following algorithms and protocols for issuing and showing of credentials and revoking users as:*

- $\text{Setup}(1^\lambda, 1^L) \rightarrow (\text{pp}, td)$ : Initializes the scheme, outputting public parameters and a trapdoor  $td$ .
- $\text{TKeyGen}(\text{pp}) \rightarrow (tsk, tpk)$ : Takes  $\text{pp}$  and outputs a keypair  $(tsk, tpk)$  for the TRA. The  $tpk$  includes a deny list of revoked users and is continuously updated.
- $\text{RootKeyGen}(\text{pp}, L) \rightarrow (\text{sk}_{root}, \text{pk}_{root})$ : Generates a key pair used for the root key  $\text{pk}_{root}$  (i.e.,  $L' = 0$ ) which is trusted for integrity but does not need to be trusted for anonymity.
- $\text{UKeyGen}(\text{pp}, L) \rightarrow (\text{sk}, \text{pk})$ : Generates a user's key pair for a specified level.
- $\text{RegisterUser}(\text{pp}, tsk, \text{pk}) \rightarrow (tok)$ : Creates a revocation token on the given public key so that the public key can be revoked later with a deny list.
- $(\text{Issue}(\text{sk}_I, \text{cred}_I, L') \leftrightarrow \text{Receive}(\text{sk}_R, tok, L')) \rightarrow \text{cred}_R$ : An interactive protocol to receive a signature on a pseudonym. It is run between the two users distinguished by  $I$  for issuer or  $R$  for receiver. If this is a root credential then  $\text{cred}_I = \perp$ .
- $(\text{Prove}(\text{sk}_P, \text{cred}_P, L') \leftrightarrow \text{Verify}(\text{pk}_{root}, L', tpk)) \rightarrow (b, \{tok_i\}_{i \in [L']})$ : A user proves they know a credential on a pseudonym that verifies under the given root key,  $\text{pk}_{root}$ . If the verification is successful, the verifier outputs 1 along with a list of revocation tokens for the prover and the chain of credentials. If the verification is unsuccessful, the verifier outputs 0.
- $\text{RevokeUser}(\text{pp}, tsk, tpk, tok) \rightarrow tpk'$ : Takes in the TRA's key pair  $(tsk, tpk)$  as well as the token for a registered public key and outputs an updated public key  $tpk'$  that can be used to recognize any showings in which this public key is part of the chain. For security reasons, this can fail, outputting  $\perp$ . As an example, we want this function to fail if a malicious  $tok$  is provided.

## 4.2 DAC Security Definitions

In Fig. 6 we formally define the oracles used in our security games. Any formal outputs of oracles are received by the adversary and any modified internal state of the challenger is listed in the description. When calling interactive functions from the DAC scheme (such as  $\text{Prove}(\cdot) \leftrightarrow \text{Verify}(\cdot)$ ), the challenger records the transcript of the interaction in addition to the output of the function. For example, in the  $\text{VerifyCred}$  oracle in Fig. 6, we have the challenger interact with the adversary using the  $\text{Verify}$  function, and in addition to outputting the result of the verification ( $b$ ) and the list of revocation tokens,  $\{tok_i\}_{i \in [L']}$ , the protocol also outputs a transcript ( $\tau$ ) of the interaction between the prover and the verifier. Throughout the game, the challenger maintains some state to keep track of honest users and credentials that were given to the adversary. This global state is used in the unforgeability game. Specifically, the challenger keeps track of one set,  $\text{DEL}_A$  to keep track of what has been delegated to the adversary. Moreover, the challenger initializes three maps to keep track of honest user state,  $\text{SK}$  holds user secret keys,  $\text{CRED}$  holds user credentials, and  $\text{LVL}$  records what level a user's credential is for. They are as follows:  $\text{SK} : \mathcal{H} \rightarrow \mathcal{SK}$ ,  $\text{CRED} : \mathcal{SK} \rightarrow \mathcal{CRED}$  and  $\text{LVL} : \mathcal{SK} \rightarrow [L]$ , where  $\mathcal{H}$  is the set of all honest user handles,  $\mathcal{SK}$  is the set of all secret keys, and  $\mathcal{CRED}$  is the set of all credentials. The root key is included in  $\text{SK}$  as  $id = 0$  with  $\text{LVL}[\text{SK}[0]] = 0$ . The root does not have a credential in  $\text{CRED}$ . The challenger also keeps track of what keys have been added to the deny list with the list  $\text{SK}_{DL} \subset \mathcal{SK}$ . At the start of any game, the challenger initializes all sets to the empty set and initializes all maps to be degenerate, such as mapping  $\forall i \in \mathcal{H}, \text{SK}[i] = \perp$ . In the unforgeability game, we give the adversary access to all of the oracles. In the  $\mathcal{O}^{\text{CreateHonestUser}}$  oracle, the adversary specifies two users with one issuing a credential to the other. We initialize users at the same time that they are issued a credential to simplify the scheme. As an example use of this oracle, the adversary can specify  $id_I = 0$  at the start of a game to have a credential be issued from the root. We do not allow the adversary to issue to a user multiple times, and thus if the specified user already exists when the adversary calls  $\mathcal{O}^{\text{CreateHonestUser}}$ , then the challenger aborts. We allow the adversary to issue credentials to honest users using the  $\mathcal{O}^{\text{ReceiveCred}}$  oracle. In this oracle, the adversary specifies a user to receive a credential at a specified level. If the adversary was never issued a credential that would allow them to delegate this credential to the honest user, the challenger set a forgery flag in the global state (labeled *forgery*) which is checked in the unforgeability game. In the  $\mathcal{O}^{\text{IssueFrom}}$  oracle, the adversary receives a credential from an honest user and the challenger records which adversarial key received this credential at which level. In the  $\mathcal{O}^{\text{ProveTo}}$  oracle, the adversary acts as the verifier for a user. In the  $\mathcal{O}^{\text{RegisterUser}}$  oracle the adversary can receive a revocation token for one of their public keys. In the  $\mathcal{O}^{\text{RevokeUser}}$  oracle, the adversary can add a user to the deny list.

**Anonymity:** Our anonymity definition is shown in Def 11. The anonymity game involves the adversary and the challenger. The adversary controls all participants, including the root credential authority (but does not control the TRA). The game proceeds as follows: The challenger generates the public parameters,



$\mathcal{O}^{\text{CreateHonestUser}}(id_I, id_R, L') \rightarrow (\text{pk})$ <hr/> <b>if</b> $\text{SK}[id_R] \neq \perp$ , <b>return</b> $\perp$ <b>if</b> $\text{SK}[id_I] = \perp$ , <b>return</b> $\perp$ <b>if</b> $\text{LVL}[\text{SK}[id_I]] \neq L' - 1$ , <b>return</b> $\perp$ $(\text{sk}, \text{pk}) \leftarrow \text{KGen}(\text{pp})$ $\text{SK}[id_R] = \mathcal{E}_{\text{pk}}(\text{pk})^\dagger$ $\text{LVL}[\text{sk}] = L'$ $\text{tok} = \text{RegisterUser}(\text{pp}, \text{tsk}, \text{pk})$ $(\text{cred}, \tau) \leftarrow (\text{Receive}(\text{pp}, \text{SK}[id_R], \text{tok}, \text{pk}_{\text{root}}, L)$ $\quad \leftrightarrow \text{Issue}(\text{pp}, \text{SK}[id_I], \text{CRED}[id_I], L))$ $\text{CRED}[id_R] = \text{cred}$ <b>return</b> $(\tau, \text{pk})$	$\mathcal{O}^{\text{IssueFrom}}(id) \leftrightarrow \mathcal{A}$ <hr/> <b>if</b> $\text{SK}[id] = \perp$ , <b>return</b> $\perp$ <b>if</b> $\text{CRED}[id] = \perp$ , <b>return</b> $\perp$ $(\text{cred}, \tau) \leftarrow (\text{Issue}(\text{SK}[id], \text{CRED}[id]) \leftrightarrow \mathcal{A})$ $\text{DEL}_{\mathcal{A}} = \text{DEL}_{\mathcal{A}} \cup \{(\mathcal{E}_{\text{pk}, R}(\tau), \text{LVL}[\text{SK}[id]])\}$ <b>return</b> $\text{cred}$
$\mathcal{O}^{\text{ReceiveCred}}(id_R, L') \leftrightarrow \mathcal{A}$ <hr/> <b>if</b> $\text{SK}[id_R] \neq \perp$ , <b>return</b> $\perp$ $(\text{sk}, \text{pk}) \leftarrow \text{KGen}(\text{pp})$ $\text{SK}[id_R] = \mathcal{E}_{\text{pk}}(\text{pk})^\dagger$ $\text{LVL}[\text{sk}] = L'$ $\text{tok} = \text{RegisterUser}(\text{pp}, \text{tsk}, \text{pk})$ $(\text{cred}, \tau) \leftarrow (\text{Receive}(\text{pp}, \text{SK}[id_R], \text{tok}, \text{pk}_{\text{root}}, L)$ $\quad \leftrightarrow \mathcal{A}(\text{pk}))$ <b>if</b> $\text{cred} \neq \perp \wedge \forall i \in [L'], (*, i) \notin \text{DEL}_{\mathcal{A}}$ , $\quad \text{forgery} = 1$ $\text{CRED}[id_R] = \text{cred}$ <b>return</b> $\perp$	$\mathcal{O}^{\text{ProveTo}}(id) \leftrightarrow \mathcal{A}$ <hr/> <b>if</b> $\text{SK}[id] = \perp$ , <b>return</b> $\perp$ <b>if</b> $\text{CRED}[id] = \perp$ , <b>return</b> $\perp$ $\text{Prove}(\text{pp}, \text{SK}[id], \text{CRED}[id], \text{pk}_{\text{root}}) \leftrightarrow \mathcal{A}$ <b>return</b> $\perp$
$\mathcal{O}^{\text{RegisterUser}}(\text{pk}) \rightarrow \text{tok}$ <hr/> <b>return</b> $\text{RegisterUser}(\text{pk})$	$\mathcal{O}^{\text{RevokeUser}}(\text{pk})$ <hr/> $\text{tpk}' = \text{RevokeUser}(\text{pp}, \text{tsk}, \text{pk})$ <b>if</b> $\text{tpk}' \neq \perp$ , $\quad \text{tpk}' = \text{tpk}$ $\quad \text{SK}_{DL} = \text{SK}_{DL} \cup \{\mathcal{E}_{\text{sk}}(\text{pk})\}$ <b>return</b> $\text{tpk}'$

**Fig. 6.** Definition of Oracles.

$\dagger$  The oracle uses  $\mathcal{E}_{\text{pk}}$  to ensure  $\text{SK}[id]$  holds a canonical representation of the secret key.

which are given to the adversary along with the registrar's public key and access to a registration and revocation oracle. The adversary creates two credential chains of the same length, rooted at the same credential authority, and provides the secret keys of the end users of these chains to the challenger. The challenger ensures that they are valid credential chains. The challenger randomly selects one of the users and proves possession of the corresponding credential chain to the adversary. The adversary wins if it can correctly identify which user the challenger picked. No honest users are created in this game, as the adversary controls all aspects except for the registration and revocation oracles. To model issuer privacy and showing privacy, the adversary outputs a bit,  $j$ , to indicate

whether the challenger should act as the issuer or the shower. We formalize this game in Def. 11.

**Definition 11 (Anonymity).** *A DAC scheme is anonymous if the advantage any PPT adversary ( $\mathcal{A} = \{\mathcal{A}_0, \mathcal{A}_1\}$ ) in the following anonymity game, defined by the chance that the game outputs 1, is  $1/2 + \text{negl}(\lambda)$ :*

- 1:  $\text{pp} \leftarrow \text{Setup}(1^\lambda, 1^L)$
- 2:  $(\text{tsk}, \text{tpk}) \leftarrow \text{TKeyGen}(\text{pp})$
- 3:  $(st, j, \text{pk}_{\text{root}}, \text{sk}_0, \text{cred}_0, \text{sk}_1, \text{cred}_1, L') \leftarrow \mathcal{A}_0^{\mathcal{O}^{\text{RegisterUser}(\cdot)}, \mathcal{O}^{\text{RevokeUser}(\cdot)}}(\text{pp}, \text{tpk})$
- 4:  $\forall i \in \{0, 1\}$  :
- 5:   **if**  $(\text{Prove}(\text{pp}, \text{sk}_i, \text{cred}_i, L') \leftrightarrow \text{Verify}(\text{pk}_{\text{root}}, L', \text{tpk})) \neq (1, *)$ , **return**  $\perp$
- 6:  $b \leftarrow_{\$} \{0, 1\}$
- 7: **if**  $j = 0$ ,
- 8:    $\text{Prove}(\text{pp}, \text{sk}_b, \text{cred}_b, L') \leftrightarrow \mathcal{A}_1(st) \rightarrow b'$
- 9: **if**  $j = 1$ ,
- 10:    $\text{Issue}(\text{pp}, \text{sk}_b, \text{cred}_b, L') \leftrightarrow \mathcal{A}_1(st) \rightarrow b'$
- 11: **return**  $b' = b$

Unlike the anonymity definition in [18], we allow the adversary to participate in the challenge credential chain. Therefore, we do not need to control the state of the game with the challenger; in the anonymity game, the challenger only performs the role of the TRA and the challenge user. In addition, we aim to maintain the anonymity of honest users even when the anonymity of some adversarial users is revoked. This new definition represents a simplified and more comprehensive anonymity model, which we present as a novel contribution.

**Unforgeability:** Our unforgeability game is simpler than [18], even though it is conceptually similar. We remove oracles that reveal pseudonyms of honest users. Revealing pseudonyms alone has no real-world use-case in DAC and the adversary effectively reveals pseudonyms during a showing anyway. Also, we integrate user creation with credential issuance, as a user's key pair is not used until it is associated with a credential. Otherwise, our unforgeability definition (Def. 12) is mostly unchanged from [18], but we add the `RegisterUser` and `RevokeUser` functions that facilitate revocation.

Moreover, our unforgeability definition ensures that the adversary was correctly delegated a credential on line 10 in Def. 12, and that none of the keys in the adversary's credential are on the deny list, on line 11.

To ensure that the challenger can check the key classes, we parameterize the definition with the extractor,  $\mathcal{E}_{\text{pk}}$ , which takes in a public key and extracts a secret key from it. If  $\mathcal{E}_{\text{pk}}$  is run on the transcript of a showing, it extracts the secret key of the credential holder. If  $\mathcal{E}_{\text{pk}}$  is run on the transcript of an issuing, it extracts the secret key of the issuer. We denote these by  $\mathcal{E}_{\text{pk},R}$  for receiver and  $\mathcal{E}_{\text{pk},I}$  for issuer. This extractor must extract the same secret key no matter how the public key has been randomized. For mercurial signatures, this means that the extractor extracts a canonical secret key which is constant over any

representation of the equivalence class of secret keys. We also assume an extractor  $\mathcal{E}_{\text{cred}}$  that can take in a credential or the transcript of a showing of a credential and output the canonical secret keys used in the delegation chain including the end user of the credential.

**Definition 12 (Unforgeability).** *A DAC scheme is unforgeable if any PPT adversary's advantage in the following game, defined by the chance that the game outputs 1, is negligible in  $\lambda$ .  $\mathcal{A}$  is given all the oracles labeled as  $\mathcal{O}$  from Fig 6:*

```

1:  $(\text{pp}, td) \leftarrow \text{Setup}(1^\lambda, 1^L)$ 
2:  $(tsk, tpk) \leftarrow \text{TKeyGen}(\text{pp})$ 
3:  $(\text{sk}, \text{pk}) \leftarrow \text{RootKeyGen}(\text{pp})$ 
4:  $\text{SK}[0] = \text{sk}; \text{pk}_{\text{root}} = \text{pk}$ 
5:  $(st, L) \leftarrow \mathcal{A}_0^{\mathcal{O}}(\text{pp}, \text{pk})$ 
6:  $((b, *), \tau) \leftarrow (\text{Verify}(\text{pk}_{\text{root}}, L, tpk) \leftrightarrow \mathcal{A}_1(st)$ 
7:  $\{\text{sk}_i\}_{i \in [L']} \leftarrow \mathcal{E}_{\text{cred}}(\tau)$ 
8: if forgery = 1, return 1
9: if  $\text{sk}_0 \neq \text{sk}_{\text{root}}$ , return  $b$ 
10: if  $\forall i \in [L'], (\text{sk}_i, i) \notin \text{DEL}_{\mathcal{A}}$ , return  $b$ 
11: if  $\exists i \in [L'], s.t. \text{sk}_i \in \text{SK}_{DL}$ , return  $b$ 
12: return 0

```

### 4.3 DAC Construction

Our DAC construction uses a function `MultiSetup` which builds  $L$  sets of public parameters for our underlying mercurial signature scheme with  $\ell = 2$  such that the schemes have the structure described in Sec. 3.3. We note that for the root authority we use the CL19 scheme [18] instead of our scheme as this is sufficient for the root. This function is described in the Supplementary Material A.

To simplify our DAC construction, we add a function `TracePK`, which takes in a “linker” and a revocation token and returns if this linker is associated with the revocation token. These linker values will be stored in the deny list in the TRA’s public key  $tpk$ .

**Definition 13 (DAC construction).**

- $\text{Setup}(1^\lambda, 1^L) \rightarrow (\text{pp}, td)$ : Call the `MultiSetup` function described in the Supplementary Material A which generates  $L$  correlated parameters for our signature scheme in Fig. 3,  $\{\text{pp}_i\}_{i \in [L]} = \text{MultiSetup}(1^\lambda, 1^{\ell=2}, 1^L)$ . Then initialize extra bases for the revocation authority and the root authority using the CL19 scheme,  $(\text{pp}_{\text{CL19}}) \leftarrow \text{Setup}_{\text{CL19}}(1^\lambda, 1^{\ell=2})$ . Output  $\text{pp} = (\{\text{pp}_i\}_{i \in [L]}, \text{pp}_{\text{CL19}})$ ,  $td = (\{td_i\}_{i \in [\ell]})$ .
- $\text{RootKeyGen}(\text{pp}) \rightarrow (\text{sk}_{\text{root}}, \text{pk}_{\text{root}})$ : Generate a CL19 key pair using  $\text{pp}_{\text{CL19}}$ .
- $\text{UKeyGen}(\text{pp}, L') \rightarrow (\text{sk}, \text{pk})$ : Create a secret key for the corresponding scheme,  $(\text{sk}, \text{pk}) \leftarrow \text{KGen}(\text{pp}_{L'})$ . The user initializes their credential chain as  $\text{chain} = \perp$ .

- $\text{TKeyGen}(\text{pp}) \rightarrow (tsk, tpk)$ : Create a CL19 key  $(sk, pk)$  of length  $\ell = 2$ ,  $(tsk_{sk}, tpk_{pk}) \leftarrow \text{KGen}(\text{pp}_{\text{CL19}})$ . Initialize a set of linkers,  $tsk_{link} = \emptyset$ . Initialize a deny list,  $DL = \emptyset$ . Let  $tsk_{sk} = sk$ ,  $tsk = (tsk_{sk}, tsk_{link})$ , and  $tpk = (tpk_{pk}, DL)$ .
- $\text{RegisterUser}(\text{pp}, tsk, pk) \rightarrow (tok)$ : Generate a new key pair  $(sk_{rev}, pk_{rev})$  using  $\text{pp}_{\text{CL19}}$ . Sign  $pk_{rev}$  using  $tsk_{sk}$  where  $tsk = (tsk_{sk}, tsk_{link})$  yielding  $\sigma_0$ . Then, use  $sk_{rev}$  to sign  $pk$ , yielding  $\sigma_1$ . This yields the revocation token,  $tok = (pk_{rev}, \sigma_0, \sigma_1)$ . The secret key,  $sk_{rev}$ , will serve as the linker for this revocation token and it is denoted as  $link$ . Save this linker in the TRA's state,  $tsk'_{link} = tsk_{link} \cup \{link\}$  and update the state:  $tsk' = (tsk_{sk}, tsk'_{link})$ . Output revocation token  $tok$ .
- $\text{RevokeUser}(\text{pp}, tsk, tpk, tok) \rightarrow tpk'$ : Iterate through the linkers  $(link_i)$  in  $tsk_{link}$  and check if  $\text{TracePK}(\text{pp}, link_i, tok) = 1$  for each of them. If this holds for a linker,  $link_i$ , concatenate  $link_i$  to the linkers in  $tpk$  (the deny list) and output this new public key as  $tpk'$ .
- $\text{TracePK}(\text{pp}, link, tok) \rightarrow \{0, 1\}$ : Parse  $tok$  as  $tok = (pk_{rev}, \sigma_0, \sigma_1)$ . Check if  $\text{RecognizePK}(\text{pp}_{\text{CL19}}, link, pk_{rev}) = 1$  (cf. Sec. 2). Therefore, parse  $pk_{rev}$  as  $pk_{rev} = (\hat{X}_1, \hat{X}_2)$ . Parse  $link = (x_1, x_2)$ . Check if  $\hat{X}_1^{x_2/x_1} = \hat{X}_2$ . If this holds, output 1. Otherwise, output 0.
- $(\text{Issue}(sk_I, cred_I, L') \leftrightarrow \text{Receive}(sk_R, tok, L')) \rightarrow cred_R$ : The receiver samples  $\rho \leftarrow \mathcal{KC}$  and generates a randomized public key from their secret key,  $pk'$ , using the randomization factor,  $\rho$ . They also randomize their revocation token,  $tok$ , yielding,  $tok = (pk'_{rev}, \sigma'_0, \sigma'_1)$ , such that  $\text{Verify}_{\text{CL19}}(\text{pp}_{\text{CL19}}, tpk, pk'_{rev}, \sigma'_0) = 1$  and  $\text{Verify}_{\text{CL19}}(\text{pp}_{\text{CL19}}, pk'_{rev}, pk', \sigma'_0) = 1$ . The receiver sends over  $pk'$ . The issuer then randomizes all public keys in their credential chain along with the signatures, randomizing their secret key to match. They also randomize all revocation tokens in their chain as described above. They then sign  $pk'$  yielding a signature,  $\sigma$ . They send their randomized credential chain,  $chain$ , along with  $\sigma$  to the receiver. The receiver computes the chain,  $chain' = chain \parallel (pk', \sigma, tok')$ . The receiver stores their credential as  $cred = (chain', \rho)$ . The randomizer is also stored to ensure the receiver can correctly randomize their secret key to match their public key in the chain.
- $(\text{Prove}(sk_P, cred_P, L') \leftrightarrow \text{Verify}(pk_{root}, L', tpk)) \rightarrow (b, \{tok_i\}_{i \in [L']})$ : The prover randomizes all public keys and signatures in their credential  $cred$  using  $\rho^* = \rho * \rho'$  where  $\rho$  is the randomizer found in their credential. They send over their randomized credential chain,  $chain$ , and perform an interactive proof of knowledge that they know the  $sk$  that corresponds to the last public key in the chain. The verifier then verifies each public key with the signatures. The verifier also iterates through the revocation tokens in the credential chain checks that for each public key  $pk_i$  and  $tok_i = (pk_{rev,i}, \sigma_{i,0}, \sigma_{i,1})$  in the chain it holds that  $\text{Verify}_{\text{CL19}}(tpk, pk_{rev,i}, \sigma_{i,0}) = 1$  and  $\text{Verify}_{\text{CL19}}(pk_{rev,i}, pk_i, \sigma_{i,1}) = 1$ . They then also iterate through each  $link_i \in tpk$  and ensure that  $\text{TracePK}(\text{pp}, tok_i, link_i) = 0$  for each level  $i$  in the length of the chain. If all these checks hold, the ver-

ifier outputs 1 and if any checks fail, the verifier outputs 0. The verifier also outputs all of the  $tok_i$  values received from the prover.

**Theorem 7 (Unforgeability of the construction in Def. 13).** *If the underlying signature scheme is unforgeable with respect to Def. 2, our construction in Def. 13 is unforgeable with respect to Def. 12.*

**Theorem 8 (Anonymity of the construction in Def. 13).** *If the underlying signature scheme has origin-hiding and has adversarial public key-class hiding, the DAC scheme in Def. 13 is anonymous with respect to definition 11.*

We prove these theorems in Sec. B.3.

## 5 Conclusion and Future Work

In this paper, we constructed mercurial signatures with stronger security properties than seen in the literature, which could potentially be used for many privacy-preserving schemes just as the first such signatures in [18]. We use it as a basis for an efficient DAC scheme with strong privacy guarantees and delegator revocation functionality. Our DAC construction could be further adapted to support attributes extending its functionality, where the technique from [32] seems promising. We leave this extension to future work.

## References

1. Acar, T., Nguyen, L.: Revocation for delegatable anonymous credentials. In: Catalano, D., Fazio, N., Gennaro, R., Nicolosi, A. (eds.) PKC 2011. LNCS, vol. 6571, pp. 423–440. Springer, Heidelberg (Mar 2011). [https://doi.org/10.1007/978-3-642-19379-8\\_26](https://doi.org/10.1007/978-3-642-19379-8_26)
2. Belenkiy, M., Camenisch, J., Chase, M., Kohlweiss, M., Lysyanskaya, A., Shacham, H.: Randomizable proofs and delegatable anonymous credentials. In: Halevi, S. (ed.) CRYPTO 2009. LNCS, vol. 5677, pp. 108–125. Springer, Heidelberg (Aug 2009). [https://doi.org/10.1007/978-3-642-03356-8\\_7](https://doi.org/10.1007/978-3-642-03356-8_7)
3. Ben-Sasson, E., Chiesa, A., Green, M., Tromer, E., Virza, M.: Secure sampling of public parameters for succinct zero knowledge proofs. In: 2015 IEEE Symposium on Security and Privacy. pp. 287–304. IEEE Computer Society Press (May 2015). <https://doi.org/10.1109/SP.2015.25>
4. Blömer, J., Bobolz, J.: Delegatable attribute-based anonymous credentials from dynamically malleable signatures. In: Preneel, B., Vercauteren, F. (eds.) ACNS 18. LNCS, vol. 10892, pp. 221–239. Springer, Heidelberg (Jul 2018). [https://doi.org/10.1007/978-3-319-93387-0\\_12](https://doi.org/10.1007/978-3-319-93387-0_12)
5. Boneh, D., Shacham, H.: Group signatures with verifier-local revocation. In: Atluri, V., Pfitzmann, B., McDaniel, P. (eds.) ACM CCS 2004. pp. 168–177. ACM Press (Oct 2004). <https://doi.org/10.1145/1030083.1030106>
6. Bowe, S., Gabizon, A., Green, M.D.: A multi-party protocol for constructing the public parameters of the pinocchio zk-SNARK. In: Zohar, A., Eyal, I., Teague, V., Clark, J., Bracciali, A., Pintore, F., Sala, M. (eds.) FC 2018 Workshops. LNCS, vol. 10958, pp. 64–77. Springer, Heidelberg (Mar 2019). [https://doi.org/10.1007/978-3-662-58820-8\\_5](https://doi.org/10.1007/978-3-662-58820-8_5)

7. Brorsson, J., David, B., Gentile, L., Pagnin, E., Wagner, P.S.: Papr: Publicly auditable privacy revocation for anonymous credentials. In: Rosulek, M. (ed.) *Topics in Cryptology – CT-RSA 2023*. pp. 163–190. Springer International Publishing, Cham (2023)
8. Camenisch, J., Drijvers, M., Dubovitskaya, M.: Practical UC-secure delegatable credentials with attributes and their application to blockchain. In: Thuraisingham, B.M., Evans, D., Malkin, T., Xu, D. (eds.) *ACM CCS 2017*. pp. 683–699. ACM Press (Oct / Nov 2017). <https://doi.org/10.1145/3133956.3134025>
9. Camenisch, J., Kohlweiss, M., Soriente, C.: An accumulator based on bilinear maps and efficient revocation for anonymous credentials. In: Jarecki, S., Tsudik, G. (eds.) *PKC 2009*. LNCS, vol. 5443, pp. 481–500. Springer, Heidelberg (Mar 2009). [https://doi.org/10.1007/978-3-642-00468-1\\_27](https://doi.org/10.1007/978-3-642-00468-1_27)
10. Camenisch, J., Kohlweiss, M., Soriente, C.: Solving revocation with efficient update of anonymous credentials. In: Garay, J.A., Prisco, R.D. (eds.) *SCN 10*. LNCS, vol. 6280, pp. 454–471. Springer, Heidelberg (Sep 2010). [https://doi.org/10.1007/978-3-642-15317-4\\_28](https://doi.org/10.1007/978-3-642-15317-4_28)
11. Camenisch, J., Krenn, S., Lehmann, A., Mikkelsen, G.L., Neven, G., Pedersen, M.Ø.: Formal treatment of privacy-enhancing credential systems. In: Dunkelman, O., Keliher, L. (eds.) *SAC 2015*. LNCS, vol. 9566, pp. 3–24. Springer, Heidelberg (Aug 2016). [https://doi.org/10.1007/978-3-319-31301-6\\_1](https://doi.org/10.1007/978-3-319-31301-6_1)
12. Camenisch, J., Lysyanskaya, A.: An efficient system for non-transferable anonymous credentials with optional anonymity revocation. In: Pfitzmann, B. (ed.) *EUROCRYPT 2001*. LNCS, vol. 2045, pp. 93–118. Springer, Heidelberg (May 2001). [https://doi.org/10.1007/3-540-44987-6\\_7](https://doi.org/10.1007/3-540-44987-6_7)
13. Camenisch, J., Lysyanskaya, A.: Dynamic accumulators and application to efficient revocation of anonymous credentials. In: Yung, M. (ed.) *CRYPTO 2002*. LNCS, vol. 2442, pp. 61–76. Springer, Heidelberg (Aug 2002). [https://doi.org/10.1007/3-540-45708-9\\_5](https://doi.org/10.1007/3-540-45708-9_5)
14. Camenisch, J., Stadler, M.: Proof systems for general statements about discrete logarithms. Technical Report/ETH Zurich, Department of Computer Science **260** (1997)
15. Chase, M., Lysyanskaya, A.: On signatures of knowledge. In: Dwork, C. (ed.) *CRYPTO 2006*. LNCS, vol. 4117, pp. 78–96. Springer, Heidelberg (Aug 2006). [https://doi.org/10.1007/11818175\\_5](https://doi.org/10.1007/11818175_5)
16. Chaum, D.: Security without identification: transaction systems to make big brother obsolete. *Commun. ACM* **28**(10), 1030–1044 (oct 1985). <https://doi.org/10.1145/4372.4373>
17. Connolly, A., Lafourcade, P., Perez-Kempner, O.: Improved constructions of anonymous credentials from structure-preserving signatures on equivalence classes. In: Hanaoka, G., Shikata, J., Watanabe, Y. (eds.) *PKC 2022, Part I*. LNCS, vol. 13177, pp. 409–438. Springer, Heidelberg (Mar 2022). [https://doi.org/10.1007/978-3-030-97121-2\\_15](https://doi.org/10.1007/978-3-030-97121-2_15)
18. Crites, E.C., Lysyanskaya, A.: Delegatable anonymous credentials from mercurial signatures. In: Matsui, M. (ed.) *CT-RSA 2019*. LNCS, vol. 11405, pp. 535–555. Springer, Heidelberg (Mar 2019). [https://doi.org/10.1007/978-3-030-12612-4\\_27](https://doi.org/10.1007/978-3-030-12612-4_27)
19. Crites, E.C., Lysyanskaya, A.: Mercurial signatures for variable-length messages. *PoPETs* **2021**(4), 441–463 (Oct 2021). <https://doi.org/10.2478/popets-2021-0079>

20. Derler, D., Hanser, C., Slamanig, D.: A new approach to efficient revocable attribute-based anonymous credentials. In: Groth, J. (ed.) 15th IMA International Conference on Cryptography and Coding. LNCS, vol. 9496, pp. 57–74. Springer, Heidelberg (Dec 2015). [https://doi.org/10.1007/978-3-319-27239-9\\_4](https://doi.org/10.1007/978-3-319-27239-9_4)
21. Fuchsbauer, G., Hanser, C., Slamanig, D.: Structure-preserving signatures on equivalence classes and constant-size anonymous credentials. *Journal of Cryptology* **32**(2), 498–546 (Apr 2019). <https://doi.org/10.1007/s00145-018-9281-4>
22. Griffy, S., Lysyanskaya, A.: PACIFIC: Privacy-preserving automated contact tracing scheme featuring integrity against cloning. *Cryptology ePrint Archive*, Report 2023/371 (2023), <https://eprint.iacr.org/2023/371>
23. Groth, J., Kohlweiss, M., Maller, M., Meiklejohn, S., Miers, I.: Updatable and universal common reference strings with applications to zk-SNARKs. In: Shacham, H., Boldyreva, A. (eds.) CRYPTO 2018, Part III. LNCS, vol. 10993, pp. 698–728. Springer, Heidelberg (Aug 2018). [https://doi.org/10.1007/978-3-319-96878-0\\_24](https://doi.org/10.1007/978-3-319-96878-0_24)
24. Groth, J., Sahai, A.: Efficient non-interactive proof systems for bilinear groups. In: Smart, N.P. (ed.) EUROCRYPT 2008. LNCS, vol. 4965, pp. 415–432. Springer, Heidelberg (Apr 2008). [https://doi.org/10.1007/978-3-540-78967-3\\_24](https://doi.org/10.1007/978-3-540-78967-3_24)
25. Hanser, C., Slamanig, D.: Structure-preserving signatures on equivalence classes and their application to anonymous credentials. In: Sarkar, P., Iwata, T. (eds.) ASIACRYPT 2014, Part I. LNCS, vol. 8873, pp. 491–511. Springer, Heidelberg (Dec 2014). [https://doi.org/10.1007/978-3-662-45611-8\\_26](https://doi.org/10.1007/978-3-662-45611-8_26)
26. Hanzlik, L., Slamanig, D.: With a little help from my friends: Constructing practical anonymous credentials. In: Vigna, G., Shi, E. (eds.) ACM CCS 2021. pp. 2004–2023. ACM Press (Nov 2021). <https://doi.org/10.1145/3460120.3484582>
27. Mir, O., Bauer, B., Griffy, S., Lysyanskaya, A., Slamanig, D.: Aggregate signatures with versatile randomization and issuer-hiding multi-authority anonymous credentials. In: Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security. p. 30–44. CCS '23, Association for Computing Machinery, New York, NY, USA (2023). <https://doi.org/10.1145/3576915.3623203>, <https://doi.org/10.1145/3576915.3623203>
28. Mir, O., Slamanig, D., Bauer, B., Mayrhofer, R.: Practical delegatable anonymous credentials from equivalence class signatures. *Proc. Priv. Enhancing Technol.* **2023**(3), 488–513 (2023). <https://doi.org/10.56553/POPETS-2023-0093>, <https://doi.org/10.56553/popets-2023-0093>
29. Nanri, M., Kempner, O.P., Tibouchi, M., Abe, M.: Interactive threshold mercurial signatures and applications. *Cryptology ePrint Archive*, Paper 2024/625 (2024), <https://eprint.iacr.org/2024/625>, <https://eprint.iacr.org/2024/625>
30. Nikolaenko, V., Ragsdale, S., Bonneau, J., Boneh, D.: Powers-of-tau to the people: Decentralizing setup ceremonies. In: Pöpper, C., Batina, L. (eds.) Applied Cryptography and Network Security - 22nd International Conference, ACNS 2024, Abu Dhabi, United Arab Emirates, March 5-8, 2024, Proceedings, Part III. Lecture Notes in Computer Science, vol. 14585, pp. 105–134. Springer (2024). [https://doi.org/10.1007/978-3-031-54776-8\\_5](https://doi.org/10.1007/978-3-031-54776-8_5), [https://doi.org/10.1007/978-3-031-54776-8\\_5](https://doi.org/10.1007/978-3-031-54776-8_5)
31. Putman, C., Martin, K.M.: Selective delegation of attributes in mercurial signature credentials. In: Quaglia, E.A. (ed.) Cryptography and Coding - 19th IMA International Conference, IMACC 2023, London, UK, December 12-14, 2023, Proceedings. Lecture Notes in Computer Science, vol. 14421, pp. 181–196. Springer (2023). [https://doi.org/10.1007/978-3-031-47818-5\\_10](https://doi.org/10.1007/978-3-031-47818-5_10), [https://doi.org/10.1007/978-3-031-47818-5\\_10](https://doi.org/10.1007/978-3-031-47818-5_10)

32. Putman, C., Martin, K.M.: Selective delegation of attributes in mercurial signature credentials. In: Quaglia, E.A. (ed.) *Cryptography and Coding*, pp. 181–196. Springer Nature Switzerland, Cham (2024)
33. Shoup, V.: Lower bounds for discrete logarithms and related problems. In: *Proceedings of the 16th Annual International Conference on Theory and Application of Cryptographic Techniques*, p. 256–266. EUROCRYPT'97, Springer-Verlag, Berlin, Heidelberg (1997)

## Supplementary Material

### A Extendable Mercurial Signatures with an Updatable CRS

#### A.1 Updating the CRS

While we rely on a trusted setup (i.e. our construction is secure in the common reference string model) we describe a process to update the CRS which is very efficient. As described in [23], an updatable CRS has the functions: **Setup**, which generates the CRS, **Update** which updates the CRS and produces a proof of consistency, and **VerifyCRS** which takes in a proof and ensures that the CRS is correctly updated. Using these functions ensures that an efficient sequential setup can be performed and if only one of the users who include a proof in the setup is honest, the resulting CRS is secure (i.e. no party knows a full trapdoor that would allow them to break any security properties of the scheme). By “sequential” we mean that the parties that update the CRS do not have to be online at the same time, any updater can take an existing CRS and add their own trapdoors to it. We define these functions for our scheme below:

–  $(\mathbf{pp}, \pi) \leftarrow \mathbf{Setup}(1^\lambda)$ : Generate the public parameters as normal:

–  $\mathbf{Setup}(1^\lambda, 1^\ell) \rightarrow (\mathbf{pp}, \pi)$ :

Sample  $\{b_i, \hat{b}_i, \hat{d}_i, \hat{v}_i, v_i\}_{i \in [\ell]} \leftarrow \mathbb{Z}_p$ .

Compute:

$\mathbf{B} = \{B_i\}_{i \in [2\ell]}$ , where  $\forall i \in [\ell], B_i \leftarrow P^{b_i}, B_{\ell+i} \leftarrow P^{b_i \hat{b}_i}$

$\hat{\mathbf{B}} = \{\hat{B}_i\}_{i \in [2\ell]}$ , where  $\forall i \in [\ell], \hat{B}_i \leftarrow \hat{P}^{\hat{b}_i}, \hat{B}_{\ell+i} \leftarrow \hat{P}^{\hat{b}_i \hat{d}_i}$

$\hat{\mathbf{V}} = \{\hat{V}_i\}_{i \in [2\ell]}$ , where  $\forall i \in [\ell], \hat{V}_i \leftarrow \hat{P}^{\hat{v}_i \hat{b}_i}, \hat{V}_{\ell+i} \leftarrow \hat{P}^{\hat{v}_i}$

$\mathbf{V} = \{V_i\}_{i \in [2\ell]}$ , where  $\forall i \in [\ell], V_i \leftarrow P^{v_i \hat{d}_i}, V_{\ell+i} \leftarrow P^{v_i}$

Output:  $\mathbf{pp} = (\mathbf{B}, \hat{\mathbf{B}}, \hat{\mathbf{V}}, \mathbf{V})$ . As the proof,  $\pi$ , commit to each of the trapdoors and produce a proof that each of the trapdoors is known (i.e. a proof of equality of discrete logarithm representation over the generators of the bilinear pairing). This can be done efficiently with Pedersen commitments and Schnorr proofs [14] (in the ROM), using only 1 group element for the commitment and 2 elements in  $\mathbb{Z}_p$  for each proof base with constant computation cost.



– **Update**( $1^\lambda, \mathbf{pp}, (\pi_i)_{i \in [n]}$ )  $\rightarrow$  ( $\mathbf{pp}'$ ,  $\pi'$ ): Sample  $\{b'_i, \hat{b}_i, \hat{d}_i, \hat{v}_i, v_i\}_{i \in [\ell]} \leftarrow_{\$} \mathbb{Z}_p$  and compute:

$$\begin{aligned} \forall i \in [\ell], B_i' &\leftarrow B_i^{b'_i}, B_{\ell+i} \leftarrow B_{\ell+i}^{b'_i \hat{b}_i} \\ \hat{B}_i' &\leftarrow \hat{B}_i^{\hat{b}_i}, \hat{B}'_{\ell+i} \leftarrow \hat{B}_{\ell+i}^{\hat{b}_i \hat{d}_i} \\ \hat{V}_i' &\leftarrow \hat{V}_i^{\hat{v}_i \hat{d}_i}, \hat{V}'_{\ell+i} \leftarrow \hat{V}_{\ell+i}^{\hat{v}_i} \\ V_i' &\leftarrow V_i^{v'_i \hat{d}_i}, V_{\ell+i}' \leftarrow (V_{\ell+i}')^{v'_i} \end{aligned}$$

Output:  $\mathbf{pp}' = (\mathbf{B}', \hat{\mathbf{B}}', \hat{\mathbf{V}}', \mathbf{V}')$ . Commit to each trapdoor and prove that the resulting bases  $\mathbf{pp}'$  were computed correctly over  $\mathbf{pp}$  (using techniques similar to **Setup**). Output the Schnorr proofs and commitment as the proof,  $\pi$ . Include the old  $\mathbf{pp}$  in the proof as well. This can be represented using Camenisch-Stadler notation:  $\pi_R = \text{NIZK}[\{b_i, \hat{b}_i, \hat{d}_i, \hat{v}_i, v_i\}_{i \in [\ell]}, \{O_{b_i}, O_{\hat{b}_i}, O_{\hat{d}_i}, O_{\hat{v}_i}, O_{v_i}\}_{i \in [\ell]} : \forall i \in [\ell], C_{b_i} = \text{Com}(b_i, O_{b_i}), C_{\hat{b}_i} = \text{Com}(\hat{b}_i, O_{\hat{b}_i}), C_{\hat{d}_i} = \text{Com}(\hat{d}_i, O_{\hat{d}_i}), C_{\hat{v}_i} = \text{Com}(\hat{v}_i, O_{\hat{v}_i}), C_{v_i} = \text{Com}(v_i, O_{v_i})\}_{i \in [\ell]}, B_i' = B_i^{b'_i}, B_{\ell+i} = B_{\ell+i}^{b'_i \hat{b}_i}, \hat{B}_i' = \hat{B}_i^{\hat{b}_i}, \hat{B}'_{\ell+i} = \hat{B}_{\ell+i}^{\hat{b}_i \hat{d}_i}, \hat{V}_i' = \hat{V}_i^{\hat{v}_i \hat{d}_i}, \hat{V}'_{\ell+i} = \hat{V}_{\ell+i}^{\hat{v}_i}, V_i' = V_i^{v'_i \hat{d}_i}, V_{\ell+i}' = (V_{\ell+i}')^{v'_i}]$ . Thus,  $\pi_R$  using Schnorr proof with Fiat-Shamir applied consists of the  $5\ell$  Pedersen commitments to the trapdoors,  $\forall i \in [\ell], C_{b_i}, C_{\hat{b}_i}, C_{\hat{d}_i}, C_{\hat{v}_i}, C_{v_i}$ , as well as  $4\ell$  commitments to the multiplications of those trapdoors,  $\forall i \in [\ell], b_i \hat{b}_i, \hat{b}_i \hat{b}_i, \hat{v}_i \hat{b}_i, v_i \hat{d}_i$ , which makes up  $9\ell$  group elements. The Schnorr proof itself then consists of  $8\ell$  first message “commitments” (which are group elements), for each base in the public parameters, as well as  $9\ell$  first message commitments to each of the commitments to the trapdoors (including the multiplications of trapdoors). We then need  $18\ell$  final messages for the Schnorr proof, one for each trapdoors, and one for each opening of a commitment to a trapdoor.

## A.2 Extending the scheme to multiple levels

Extendability was an implicit property in [18] but was much simpler as the scheme was symmetric. Our scheme’s parameters are structured in a way that we must have explicit functions to extend the scheme such that public keys at one level in the resulting DAC scheme can sign public keys from another level. To that end, we include algorithms, **ExtendSetup** and **FinalizeSetup**, which are needed by the proof of our DAC scheme to ensure the scheme can be extended to multiple levels. We also update each of our security definitions to include an interactive setup which invokes these functions. The syntax of our mercurial signatures is thus given by the functions in Def. 14.

To explain why we need such functions and properties for our proof, let’s attempt to construct a reduction from APKCH to DAC anonymity where the APKCH challenger generates a single set of public parameters for one mercurial signature scheme, but the DAC scheme requires multiple levels of mercurial signature schemes such that public keys in the schemes can sign public keys from the next

higher scheme. This reduction receives  $\mathbf{pp}$  from the APKCH challenger. In the anonymity game, the reduction will need to make the credential depend on the APKCH challenger's secret bit. In our proof of anonymity, we'll create hybrids (shown in Def. 21) which replace each public key in the credential chain with a random public key (which, in our reduction is the APKCH challenger's public key). Thus, in our reduction, we need to replace one of the signatures in the credential chain with the signature which is returned by the APKCH challenger ( $\sigma^b$  in Fig. 2). To ensure that the adversary won't simply abort, we need this signature to appear as though it came from the DAC scheme. Thus, if this is some intermediate signature in the credential chain,  $\sigma^b$  must come from some public key,  $\mathbf{pk}_i$ , be signed by some public key,  $\mathbf{pk}_{i-1}$ , and sign another public key,  $\mathbf{pk}_{i+1}$  to continue the chain. The natural  $\mathbf{pk}_i$  to use is the  $\mathbf{pk}^b$  returned by the PKCH challenger as this verifies with  $\sigma^b$ . But, if this signature is in the middle of a credential chain, we'll need to sign  $\mathbf{pk}^b = \mathbf{pk}_i$  with some  $\mathbf{pk}_{i-1}$ . But,  $\mathbf{pk}^b$  is valid only for the public parameters that the APKCH challenger generated. Thus, this reduction must somehow include the  $\mathbf{pp}$  generated by the APKCH challenger into the public parameters for the DAC scheme. The public key bases for  $\mathbf{pp}$  are  $\hat{\mathbf{B}} = \{\hat{B}_i\}_{i \in [2\ell]}$  where  $\forall i \in [\ell], \hat{B}_i = \hat{P}^{\hat{b}_i}, \hat{B}_{\ell+i} = \hat{P}^{\hat{b}_i \hat{d}_i}$ . Let's assume we have the secret key for  $\mathbf{pk}_{i-1}$ ,  $\mathbf{sk}_{i-1} = \{x_j\}_{j \in [\ell]}$ . If we compute the signature the normal way,  $\sigma = (Z, Y, \hat{Y})$  where  $Z = (\prod_{i \in [\ell]} \hat{X}_{\ell+i}^{x_i})^y, Y = P^{1/y}, \hat{Y} = \hat{P}^{1/y}$ , and  $\mathbf{pk}^b = \{\hat{X}_i\}_{i \in [\ell]}$ , we can see that  $e(Z, \hat{Y}) = \prod_{i \in [\ell]} \hat{P}^{\hat{b}_i \hat{d}_i x_i}$ . Thus, to create public key bases for the signature scheme at level  $i-1$ , we would need to know  $\hat{b}_i$  or  $\hat{d}_i$  (otherwise, our public parameters would simply be  $\hat{\mathbf{B}}_{i-1} = \{P\}_{i \in [\ell]}$  which is trivially distinguishable). Unfortunately, the trapdoors,  $\hat{b}_i$  and  $\hat{d}_i$ , are only known to the APKCH challenger, and thus, our reduction cannot generate these bases. We could have the APKCH output  $P^{\hat{d}_i}$  (which is part of our solution), but then we run into a similar problem when we attempt to generate a  $\mathbf{pk}_{i+1}$  (which is considered a message in the APKCH challenger's scheme) that verifies with the APKCH challenger's public parameters,  $\mathbf{pp}$ , while still allowing the reduction to generate the public parameters for levels  $i+1, \dots, L$ . Defining `ExtendSetup` and `FinalizeSetup` solves this problem and allows our proof of APKCH to focus on a single set of parameters.

**Definition 14 (Extendable mercurial signatures).**

- `Setup`( $1^\lambda, 1^\ell$ )  $\rightarrow$  ( $\mathbf{pp}, td$ ): Outputs public parameters  $\mathbf{pp}$ , including parameterized equivalence relations for the message, public key, and secret key spaces:  $\mathcal{R}_M, \mathcal{R}_{\mathbf{pk}}, \mathcal{R}_{\mathbf{sk}}$  and the sample space for key and message converters. This function also outputs a trapdoor ( $td$ ) that can be used (in conjunction with the corresponding secret key) to recognize public keys.
- `ExtendSetup`( $\mathbf{pp}'$ )  $\rightarrow$  ( $\mathbf{pp}, td$ ): Extends a scheme (described by  $\mathbf{pp}'$ ) such that the outputted scheme defined by  $\mathbf{pp}$  can be used to sign the public keys of the scheme defined by  $\mathbf{pp}^*$  where  $\mathbf{pp}^* \leftarrow \text{FinalizeSetup}(\mathbf{pp}', td')$  where  $td'$  is the trapdoor for  $\mathbf{pp}'$ .

- $\text{FinalizeSetup}(\text{pp}, \text{td}, \text{td}') \rightarrow (\text{pp}^*, \text{td}^*)$ : Finalize a previously generated scheme,  $(\text{pp}, \text{td})$ , after it has been extended, where  $\text{td}'$  is the trapdoor from the extension.
- $\text{KeyGen}(\text{pp}) \rightarrow (\text{pk}, \text{sk})$ : Generates a key pair.
- $\text{Sign}(\text{sk}, M) \rightarrow \sigma$ : Signs a message  $M$  with the given secret key.
- $\text{Verify}(\text{pk}, M, \sigma) \rightarrow (0 \text{ or } 1)$ : Returns 1 iff  $\sigma$  is a valid signature for  $M$  w.r.t.  $\text{pk}$ .
- $\text{ConvertPK}(\text{pk}, \rho) \rightarrow \text{pk}'$ : Given a key converter  $\rho$ , returns  $\text{pk}'$  by randomizing  $\text{pk}$  with  $\rho$ .
- $\text{ConvertSK}(\text{sk}, \rho) \rightarrow \text{sk}'$ : Randomize a secret key such that it now corresponds to a public key which has been randomized with the same  $\rho$  (i.e. signatures from  $\text{sk}' = \text{ConvertSK}(\text{sk}, \rho)$  verify by the randomized  $\text{pk}' = \text{ConvertPK}(\text{pk}, \rho)$ ).
- $\text{ConvertSig}(\text{pk}, M, \sigma, \rho) \rightarrow \sigma'$ : Randomize the signature so that it verifies with a randomized  $\text{pk}'$  (which has been randomized with the same  $\rho$ ) and  $M$ , but  $\sigma'$  is unlinkable to  $\sigma$ .
- $\text{ChangeRep}(\text{pk}, M, \sigma, \mu) \rightarrow (M', \sigma')$ : Randomize the message-signature pair such that  $\text{Verify}(\text{pk}, M', \sigma') = 1$  (i.e.,  $\sigma'$  and  $\sigma$  are indistinguishable) where  $M'$  is a new representation of the message equivalence class,  $[M]_{\mathcal{R}_M}$ .
- $\text{VerifyKey}(\text{pp}, \text{pk}) \rightarrow 0, 1$ : Takes a public key and verifies if it is well-formed
- $\text{VerifyMsg}(\text{pp}, M) \rightarrow 0, 1$ : Takes a message and verifies if it is well-formed w.r.t public parameters  $\text{pp}$ .

**Definition 15 (Correctness).** A mercurial signature for parameterized equivalence relations,  $\mathcal{R}_M, \mathcal{R}_{\text{pk}}, \mathcal{R}_{\text{sk}}$ , message randomizer space,  $\text{sample}_\mu$ , and key randomizer space,  $\text{sample}_\rho$ , is correct if for all  $(\lambda, \ell)$ ,  $\forall(\text{pp}, \text{td}) \in \text{Setup}(1^\lambda, 1^\ell)$ , and  $\forall(\text{sk}, \text{pk}) \in \text{KGen}(1^\lambda)$ , the following holds:

- **Verification.**  $\forall M \in \mathcal{M}, \sigma \in \text{Sign}(\text{sk}, M) : \text{Verify}(\text{pk}, M, \sigma) = 1 \wedge \text{VerifyMsg}(\text{pp}, M) = 1 \wedge \text{VerifyKey}(\text{pp}, \text{pk}) = 1$ .
- **Key conversion.**  $\forall \rho \in \text{sample}_\rho, (\text{ConvertPK}(\text{pk}, \rho), \text{ConvertSK}(\text{sk}, \rho)) \in \text{KGen}(1^\lambda)$ ,  $\text{ConvertSK}(\text{sk}, \rho) \in [\text{sk}]_{\mathcal{R}_{\text{sk}}}$ , and  $\text{ConvertPK}(\text{pk}, \rho) \in [\text{pk}]_{\mathcal{R}_{\text{pk}}}$ .
- **Signature conversion.**  $\forall M \in \mathcal{M}, \sigma, \rho \in \text{sample}_\rho, \sigma', \text{pk}'$  s.t  $\text{Verify}(\text{pk}, M, \sigma) = 1$ ,  $\sigma' = \text{ConvertSig}(\text{pk}, M, \sigma, \rho)$ , and  $\text{pk}' = \text{ConvertPK}(\text{pk}, \rho)$ , then  $\text{Verify}(\text{pk}', M, \sigma') = 1$ .
- **Change of message representation.**  $\forall M \in \mathcal{M}, \sigma, \mu \in \text{sample}_\mu, M', \sigma'$  such that  $\text{Verify}(\text{pk}, M, \sigma) = 1$  and  $(M', \sigma') = \text{ChangeRep}(\text{pk}, M, \sigma; \mu)$  then  $\text{Verify}(\text{pk}, M', \sigma') = 1$  and  $M' \in [M]_{\mathcal{R}_M}$ .

- **Extension.**  $\forall(\text{pp}', td') \leftarrow \text{ExtendSetup}(\text{pp}), (\text{pp}', td') \in \text{Setup}(1^\lambda, 1^\ell). \forall(\text{pp}^*, td^*) \leftarrow \text{FinalizeSetup}(\text{pp}, td, td'), (\text{pp}^*, td^*) \in \text{Setup}(1^\lambda, 1^\ell)$ . Further,  $\mathcal{M}' = \mathcal{PK}^*$  where  $\mathcal{M}'$  is the message space for  $\text{pp}'$  and  $\mathcal{PK}^*$  is the public key space for  $\text{pp}^*$ .

To ensure our scheme stays secure when extended, we define an interactive setup protocol with an adversary ( $\text{Setup}^{\text{ext}}$  in Fig. 7) which generates a scheme with  $\text{Setup}$  then interacts with the adversary to extend it. This is used by the unforgeability and class-hiding properties in Definitions 2 and 8

```

      ( $\text{Setup}^{\text{ext}}(1^\lambda, 1^\ell) \leftrightarrow \mathcal{A}) \rightarrow (\text{pp}, td)$ 
1:  $\text{pp}_0 \leftarrow \mathcal{A}(1^\lambda, 1^\ell)$ 
2: if  $\text{pp}_0 = \perp$ ,
3:    $(\text{pp}_1, td_1) \leftarrow \text{Setup}(1^\lambda, 1^\ell)$ 
4: else ,
5:    $(\text{pp}_1, td_1) \leftarrow \text{ExtendSetup}(\text{pp}_0)$ 
6:  $td' \leftarrow \mathcal{A}(\text{pp}_1)$ 
7:  $(\text{pp}, td) \leftarrow \text{FinalizeSetup}(\text{pp}_1, td_1, td')$ 
8: return  $(\text{pp}, td)$ 

```

**Fig. 7.** An interactive setup extension function for security games

```

      ( $\text{Setup}^{\text{ext}, \mathcal{M}}(1^\lambda, 1^\ell) \leftrightarrow \mathcal{A}) \rightarrow (\text{pp}, td)$ 
1:  $(\text{pp}_0, td_0) \leftarrow \text{Setup}(1^\lambda, 1^\ell)$ 
2:  $td' \leftarrow \mathcal{A}(\text{pp}_0)$ 
3:  $(\text{pp}, td) \leftarrow \text{FinalizeSetup}(\text{pp}_1, td_1, td')$ 
4: return  $(\text{pp}, td)$ 

```

**Fig. 8.** An interactive setup extension function for message class hiding

**Definition 16 (Unforgeability under extension).** A mercurial signature scheme for parameterized equivalence relations  $\mathcal{R}_{\mathcal{M}}, \mathcal{R}_{\text{pk}}, \mathcal{R}_{\text{sk}}$ , is unforgeable if for all parameters  $(\lambda, \ell)$  and all probabilistic, polynomial-time (PPT) algorithms,  $\mathcal{A}$ , having access to a signing oracle, there exists a negligible function  $\text{negl}$  such that:

$$\Pr \left[ \begin{array}{l} \text{Verify}(pk^*, M^*, \sigma^*) = 1 \\ \wedge [pk^*]_{\mathcal{R}_{\text{pk}}} = [pk]_{\mathcal{R}_{\text{pk}}} \\ \wedge \forall M \in Q, [M^*]_{\mathcal{R}_{\mathcal{M}}} \neq [M]_{\mathcal{R}_{\mathcal{M}}} \end{array} \middle| \begin{array}{l} \text{PP} \leftarrow (\text{Setup}^{\text{ext}}(1^\lambda, 1^\ell) \leftrightarrow \mathcal{A}); \\ (\text{pk}, \text{sk}) \leftarrow \text{KeyGen}(\text{pp}); \\ (\text{pk}^*, M^*, \sigma^*) \leftarrow \mathcal{A}^{\text{Sign}(\text{sk}, \cdot)}(\text{pk}) \end{array} \right] \leq \text{negl}(\lambda) \quad (1)$$

Where  $Q$  is the list of messages that the adversary queried to the Sign oracle.

**Definition 17 (Adversarial public key class-hiding under extension).**

A mercurial signature,  $\Gamma$ , has adversarial public key class-hiding if the advantage of any PPT set of algorithms  $\mathcal{A} = \{\mathcal{A}_1, \mathcal{A}_2\}$ , the following advantage  $\text{Adv}_{\Gamma, \mathcal{A}}^{\text{APKCH}}(\lambda)$  is negligible,

$$\text{Adv}_{\Gamma, \mathcal{A}}^{\text{APKCH}}(\lambda) := \left| \Pr \left[ \mathbf{Exp}_{\Gamma, \mathcal{A}}^{\text{APKCH}, 0}(\lambda) = 1 \right] - \Pr \left[ \mathbf{Exp}_{\Gamma, \mathcal{A}}^{\text{APKCH}, 1}(\lambda) = 1 \right] \right|$$

where  $\mathbf{Exp}_{\Gamma, \mathcal{A}}^{\text{APKCH}, b}(\lambda)$  is the experiment shown in Figure 2.

<pre> 1: pp ← (Setup<sup>ext</sup>(1<sup>λ</sup>, 1<sup>ℓ</sup>) ↔ <math>\mathcal{A}</math>); 2: (sk, pk) ← KGen(pp); 3: (pk<sub><math>\mathcal{A}</math></sub>, <math>\sigma_{\mathcal{A}}</math>, M, st) ← <math>\mathcal{A}_1^{\text{O}^{\text{Sign}}(\text{sk}, \cdot)}</math>(pk, pp) 4: <math>\sigma</math> ← Sign(sk, M) 5: <math>\rho_0 \leftarrow_{\\$} \mathbb{Z}_p^*</math>; pk<sup>0</sup> ← ConvertPK(pk, <math>\rho_0</math>); 6: <math>\sigma^0 \leftarrow \text{ConvertSig}(\sigma, \rho_0)</math> 7: <math>\rho_1 \leftarrow_{\\$} \mathbb{Z}_p^*</math>; pk<sup>1</sup> ← ConvertPK(pk<sub><math>\mathcal{A}</math></sub>, <math>\rho_1</math>); 8: <math>\sigma^1 \leftarrow \text{ConvertSig}(\sigma_{\mathcal{A}}, \rho_1)</math> 9: if Verify(pk<sub><math>\mathcal{A}</math></sub>, <math>\sigma_{\mathcal{A}}</math>, M) = 1 ∧ VerifyMsg(pp, M) = 1 ∧ VerifyKey(pp, pk<sub><math>\mathcal{A}</math></sub>) = 1 10:   return <math>\mathcal{A}_2^{\text{O}^{\text{Sign}}(\text{sk}, \cdot)}</math>(pp, st, pk<sup>b</sup>, <math>\sigma^b</math>) 11: else return <math>\mathcal{A}_2^{\text{O}^{\text{Sign}}(\text{sk}, \cdot)}</math>(pp, st, <math>\perp</math>, <math>\perp</math>) </pre>
---

**Fig. 9.** Adversarial public key class-hiding under extension experiment  $\mathbf{Exp}_{\Gamma, \mathcal{A}}^{\text{APKCH}, b}(\lambda)$ .

**A.3 Construction of Extendable Mercurial Signatures**

We now complete our construction from Sec. 3.3. We left off with an insecure but correct scheme. While constructing a lower level scheme as described in Sec. 3.3 (i.e., simply using  $D_i$  as the new  $(\hat{B}'_i)$ ) allows us to ensure the two schemes are correct and the higher level scheme is secure, the naive construction does not ensure that this lower level scheme is secure. To fix this, we need to ensure that our signature scheme is secure when accepting a  $D_i$  from the higher level scheme. This requires extra work, as for the lower level scheme to remain secure, we need to inject new unknown trapdoors into  $\hat{\mathbf{B}}'$ . We describe this process in the `ExtendSetup` function in Fig. 10. To ensure the higher scheme is then correct after this modification, we also have to update the  $\hat{\mathbf{B}}$  vector in the higher level scheme using the trapdoor from the lower level scheme, which we describe in the `FinalizeSetup` function in Fig. 10.

We now describe the function, `MultiSetup`, in Def. 18 used to generate multiple correlated mercurial signature schemes. This function is used in the DAC scheme in Sec. 4. We can see by correctness in Def. 1 that because we've called

– **ExtendSetup** $(1^\lambda, \text{pp}') \rightarrow (\text{pp}, td)$ :  
Parse  $\text{pp}' = \{D_i, (B'_i), (B'_{\ell+i}), (\hat{B}'_i), (\hat{B}'_{\ell+i}), V'_i, \hat{V}'_i\}$ .  
Sample:  $b'_i, \hat{b}_i, \hat{d}_i, v_i, \hat{v}_i \leftarrow \mathbb{Z}_p$

$\forall i \in [\ell]$ ,

$$B_i \leftarrow (\hat{B}'_i)^{b_i}, \quad B_{\ell+i} \leftarrow (\hat{B}'_{\ell+i})^{b_i \hat{b}_i}$$

$$\hat{B}_i \leftarrow (D_i)^{\hat{b}_i}, \quad \hat{B}_{\ell+i} \leftarrow (D_i)^{\hat{b}_i \hat{d}_i}$$

$$\hat{V}_i \leftarrow P^{\hat{v}_i \hat{d}_i}, \quad \hat{V}_{\ell+i} \leftarrow P^{\hat{v}_i}$$

$$V_i \leftarrow (\hat{V}'_i)^{v_i \hat{d}_i}, \quad V_{\ell+i} \leftarrow (\hat{V}'_{\ell+i})^{v_i}$$

$$\mathbf{B}^1 = \{B_i\}_{i \in [\ell]}, \mathbf{B}^u = \{B_{\ell+i}\}_{i \in [\ell]},$$

$$\hat{\mathbf{B}}^1 = \{\hat{B}_i\}_{i \in [\ell]}, \hat{\mathbf{B}}^u = \{\hat{B}_{\ell+i}\}_{i \in [\ell]},$$

$$\hat{\mathbf{V}}^1 = \{\hat{V}_i\}_{i \in [\ell]}, \hat{\mathbf{V}}^u = \{\hat{V}_{\ell+i}\}_{i \in [\ell]},$$

$$\mathbf{V}^1 = \{V_i\}_{i \in [\ell]}, \mathbf{V}^u = \{V_{\ell+i}\}_{i \in [\ell]},$$

Output:  $\text{pp} = (\mathbf{B}^1, \mathbf{B}^u, \hat{\mathbf{B}}^1, \hat{\mathbf{B}}^u, \hat{\mathbf{V}}^1, \hat{\mathbf{V}}^u, \mathbf{V}^1, \mathbf{V}^u)$   
and  $td = \{\hat{b}_i, \hat{d}_i, v_i, \hat{v}_i\}_{i \in [\ell]}$

– **FinalizeSetup** $_i(1^\lambda, \text{pp}', td) \rightarrow (\text{pp}^*)$ :  
Parse  $\text{pp}' = \{D_i, (B'_i), (B'_{\ell+i}), (\hat{B}'_i), (\hat{B}'_{\ell+i}), V'_i, \hat{V}'_i\}_{i \in [\ell]}$ .  
Parse  $td = \{\hat{b}_i, \hat{d}_i, v_i, \hat{v}_i\}_{i \in [\ell]}$ .  
Compute  $\text{pk}^* = \{\mathbf{B}^{1*}, \mathbf{B}^{u*}, \hat{\mathbf{B}}^{1*}, \hat{\mathbf{B}}^{u*}, \mathbf{V}^{1*}, \mathbf{V}^{u*}, \hat{\mathbf{V}}^{1*}, \hat{\mathbf{V}}^{u*}\}$  where:

$$\mathbf{B}^{1*} = \{(B'_i)\}_{i \in [\ell]}$$

$$\mathbf{B}^{u*} = \{(B'_{\ell+i})^{b_i}\}_{i \in [\ell]}$$

$$\hat{\mathbf{B}}^{1*} = \{(\hat{B}'_i)^{b_i}\}_{i \in [\ell]}$$

$$\hat{\mathbf{B}}^{u*} = \{((\hat{B}'_{\ell+i})^{b_i \hat{b}_i})\}_{i \in [\ell]}$$

$$\mathbf{V}^{1*} = \{(V'_i)^{b_i}\}_{i \in [\ell]}$$

$$\mathbf{V}^{u*} = \{(V'_{\ell+i})\}_{i \in [\ell]}$$

$$\hat{\mathbf{V}}^{1*} = \{(\hat{V}'_i)^{b_i}\}_{i \in [\ell]}$$

$$\hat{\mathbf{V}}^{u*} = \{(\hat{V}'_{\ell+i})\}_{i \in [\ell]}$$

Output  $\text{pp}^* = \{\mathbf{B}^{1*}, \mathbf{B}^{u*}, \hat{\mathbf{B}}^{1*}, \hat{\mathbf{B}}^{u*}, \mathbf{V}^{1*}, \mathbf{V}^{u*}, \hat{\mathbf{V}}^{1*}, \hat{\mathbf{V}}^{u*}\}$ .

**Fig. 10.** Setup that can sign public keys from another scheme.

**ExtendSetup** and **FinalizeSetup**, with the correct values, by the **Extension** correctness property our resulting parameters will be in the possible output of **Setup** $(1^\lambda, 1^\ell)$  and thus they satisfy the rest of the properties in the correctness definition and their message and public key spaces will be correctly linked to create delegation chains.

**Definition 18 (Generating multiple levels of correlated mercurial signatures with APKCH).**

- $\text{MultiSetup}(1^\lambda, 1^2, 1^L) \rightarrow \{\text{pp}_i\}_{i \in [L]}$ : Compute:  $(\text{pp}'_L, \text{td}'_L) = \text{Setup}(1^\lambda, 1^{\ell=2})$  and then call  $\forall i \in [L], (\text{pp}'_{i-1}, \text{td}'_i) \leftarrow \text{ExtendSetup}(\text{pp}'_i)$ . To compute  $\text{ExtendSetup}$  correctly, an implementation will work backwards from  $i = L$ . Next, call  $\forall i \in [L], (\text{pp}_i, \text{td}_i) \leftarrow \text{FinalizeSetup}(\text{pp}'_i, \text{td}'_i, \text{td}'_{i-1})$  to finish the parameters and trapdoors,  $\{\text{pp}_i, \text{td}_i\}_{i \in [L]}$  (the final level of parameters,  $\text{pp}_0$ , do not need to be finalized as they have not been extended). The mercurial signature schemes are extended as described in Sec. 3.3.

*How extension impacts our proofs.* After the scheme is extended, the higher scheme which this was extended from will use the bases for public keys attained from the lower scheme as long as they were generated honestly (we model this by having the lower parameter generation function output the discrete logs used to update the bases so that the higher scheme can inspect them). When proving our construction with extension secure in the Supplementary Material B, our reduction receives these trapdoors,  $\hat{b}'_i, \hat{d}'_i, \hat{v}'_i$  from the adversary and because each of them is multiplied with a random scalar, the resulting  $\hat{b}^*_i, \hat{d}^*_i, \hat{v}^*_i$  are effectively random, and thus our reduction can proceed as if it had generated the trapdoors itself. We're also able to keep the generation of  $D_i$  out of any of our generic group model proofs by first proving Lemma 5 without these  $D_i$  values and then reducing to this lemma in the standard model while generating  $D_i$  values while proving Thm. 3.

We note that  $\text{FinalizeSetup}$  must be called one extra time to propagate the  $b_i$  which is exponentiated with  $(\hat{B}'_i)$  up one higher scheme in order to ensure correctness. A reader can verify that this ensures the schemes are correct. Also, since we do not propagate  $\hat{b}_i$  further down, proving this secure is a subcase of our proof for the existing call to  $\text{FinalizeSetup}$  as an adversary can simply set  $\hat{b}_i = 1$  to simulate this experiment. To fix our reduction in this case, we need the APKCH game to exponentiate an element in  $\mathbb{G}_1$  with  $b_i$  to ensure the DAC reduction can still create the correct parameters. Note that this would break message class-hiding for this scheme, but because in the APKCH game, the same message is used for both the challenger's signature and the adversary's signature, revealing  $b_i$  does not break our APKCH security proof.

## B Proofs

We first prove (in Sec. B.1) properties where the public parameters generated in the games are simply outputted by  $\text{Setup}$  instead of being outputted by  $\text{Setup}^{\text{ext}}$  as described in Fig. 7 in Sec. A. We then prove that our properties still hold when  $\text{Setup}^{\text{ext}}$  is used in Sec. B.2.

### B.1 Proofs using Setup

**Correctness of the mercurial signature scheme in Fig. 3**

**Unforgeability of the mercurial signature scheme in Fig. 3**

*Proof of Thm. 2 (Unforgeability of the construction in Fig. 3).* This proof reduces the unforgeability of our scheme to that of the original mercurial signature construction in [18]. To prove this, we give a reduction that, given a forger for our scheme, outputs a forgery for the Crites-Lysyanskaya mercurial signature scheme. Intuitively, this reduction works for two reasons: (1) the signature and verification algorithms of our scheme are identical to that of [18] (ignoring verification, which makes our construction stricter on what is accepted) and (2) the public parameters will be generated by the reduction, and thus, the reduction will be able to transform public keys and signatures from either scheme so that they appear as though they were generated from the other scheme.

Our reduction takes as input a public key  $\mathbf{pk} = \{\hat{X}_1, \dots, \hat{X}_\ell\}$  from its unforgeability challenger and needs to form a public key  $\mathbf{pk}'$  to forward to the adversary for our scheme. It forms  $\mathbf{pk}' = \{\hat{X}_1^{\hat{b}_1}, \hat{X}_2^{\hat{b}_2}, \dots, \hat{X}_\ell^{\hat{b}_\ell}, \hat{X}_1^{\hat{d}_1 \hat{b}_1}, \hat{X}_2^{\hat{d}_2 \hat{b}_2}, \dots, \hat{X}_\ell^{\hat{d}_\ell \hat{b}_\ell}\}$  where  $\mathbf{pk} = \{\hat{X}_1, \dots, \hat{X}_\ell\}$  is the public key from the challenger.

The reduction is now ready to receive signature queries from the adversary, and respond to them. When the adversary queries the reduction on message  $\mathbf{M}$ , the first step is to verify that the messages is in the message space using  $\text{VerifyMsg}(\mathbf{pp}, \mathbf{M})$ . Next, the reduction will parse the message  $\mathbf{M} = (M_1, \dots, M_{2\ell})$  into the lower half  $\mathbf{M}^l = (M_1, \dots, M_\ell)$  and upper half  $\mathbf{M}^u = (M_{\ell+1}, \dots, M_{2\ell})$ . It queries the Crites-Lysyanskaya mercurial signature challenger on the message  $\mathbf{M}^u$  and receives the signature  $\sigma = (Z, Y, \hat{Y})$ , and returns it to the adversary. We can see by inspection of the this signature computed the same way as our signature for the public key  $\mathbf{pk}'$ .

Finally, the adversary outputs its forgery: a message  $\mathbf{M}^*$  and a signature  $\sigma^*$  under public key  $\mathbf{pk}^*$ . For the forged message,  $\mathbf{M}^*$ , the reduction will use the upper half of the message vector output by the adversary, the lower half of the public key vector exponentiated with the inverse of the trapdoor ( $\hat{b}_i$ ), and will output the adversary's signature unchanged. Because the signature verification of our scheme is identical to that of [18], this is guaranteed to verify under [18] without changing the  $\sigma^*$  outputted by the adversary. To formalize this proof, we show the reduction in Fig. B.1 and analyze it.

*Analysis.* We know that the adversary's outputted message, public key, and signature will all verify. Thus, we have that  $e(Z^A, Y^A) = e(M_i^A, \hat{X}_i^A)$ ,  $e(M_i^A, \hat{V}_i) = e(M_{\ell+i}^A, \hat{V}_{\ell+i})$ , and  $e(V_i, \hat{X}_i^A) = e(V_{\ell+i}, \hat{X}_{\ell+i}^A)$ . We also know that  $\mathbf{pk}^A$  is in the same equivalence class as  $\mathbf{pk}'$  and thus,  $\exists \rho$  such that  $(\mathbf{pk}')^\rho = \mathbf{pk}^A$ . Because  $e(M_i^A, \hat{V}_i) = e(M_{\ell+i}^A, \hat{V}_{\ell+i})$ , we know that  $M_{\ell+i} = M_i^{\hat{b}_i}$ . Thus, we can see that the modified public key,  $\mathbf{pk}^*$  verifies with  $\mathbf{M}^*$ :  $e(M_{\ell+i}, (\hat{X}_i^A)^{1/\hat{b}_i}) = e(M_{\ell+i}^{1/\hat{b}_i}, (\hat{X}_i^A)) = e(M_i, (\hat{X}_i^A)) = e(Z, \hat{Y})$ . We can see that because the lower half of  $\mathbf{pk}'$  is simply the challenge public key from CL19 raised pair-wise to the  $\hat{b}_i$  power, then because  $\mathbf{pk}^*$  is raised to the inverse of  $\hat{b}_i$  power, then if  $\mathbf{pk}^A$  is in the same equivalence class as  $\mathbf{pk}'$ , then  $\mathbf{pk}^*$  is in the same equivalence class as  $\mathbf{pk}$ . Thus, our reduction's outputted forgery,  $\mathbf{M}^*, \mathbf{pk}^*, \sigma^A$ , is valid and  $\mathbf{pk}^*$  is



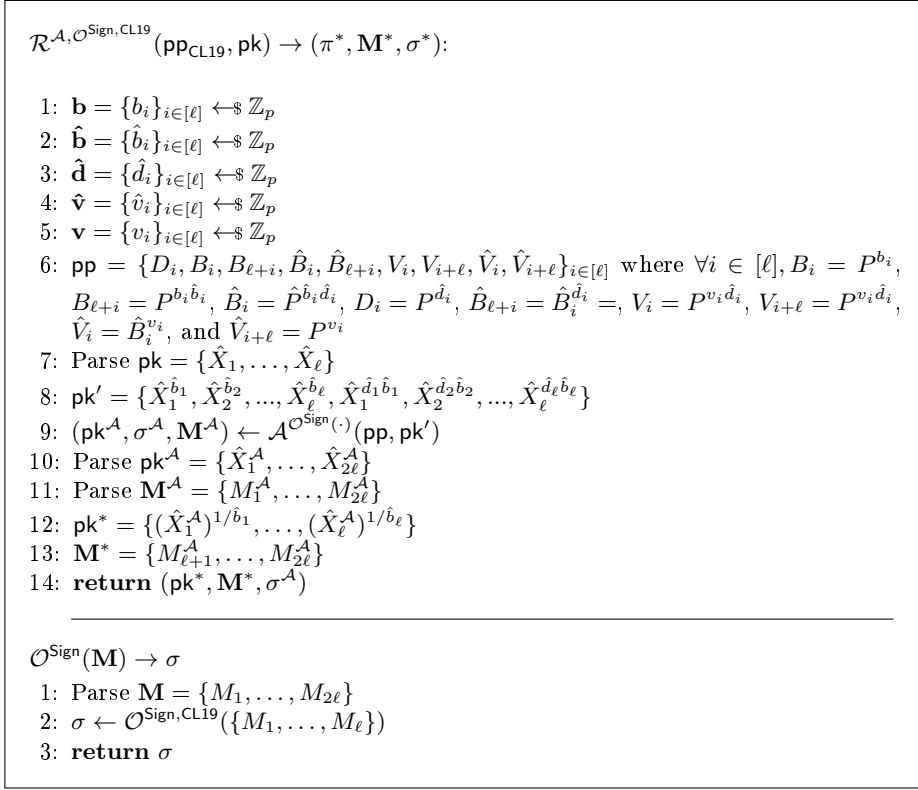


Fig. 11. Unforgeability proof reduction

in the same equivalence class as  $\text{pk}$ . Furthermore, if  $\mathbf{M}^{\mathcal{A}}$  doesn't belong to the equivalence class of any message submitted to the reduction's signature oracle by the adversary, it's also not in the equivalence class of any message submitted to the CL19 signature oracle, making it a valid forgery.

**Proof of Thm. 3 (APKCH of the construction in Fig. 3).** We show an overview of the proof structure in Fig. B.1. We will first prove that we can extract the discrete logs of messages and secrets keys from the adversary's message vectors and public keys, thus proving Lemmas 3 and 4. We will then prove that the construction in [18] satisfies a modified version of public key class-hiding (Lemma 5). We will then use these extractor of messages and keys in a reduction to a challenger of this modified public key class-hiding game, using an adversary from our strong public key class-hiding game to show that our construction achieves strong public key class-hiding if the construction in [18] satisfies this modified public key class hiding, and thus, by Lemma 5, will prove that our scheme is secure in the generic group model (GGM). This modified scheme works similarly to the game in Def. 6, but reveals the public key in

the first source group (as well as the second source group). This allows our reduction to construct its public parameters in a certain way, such signatures from the [18] construction will either be well-formed on keys that our reduction chooses, or random keys, depending on the challenge bit. We then use this to have our strong public key class-hiding adversary distinguish these two cases.

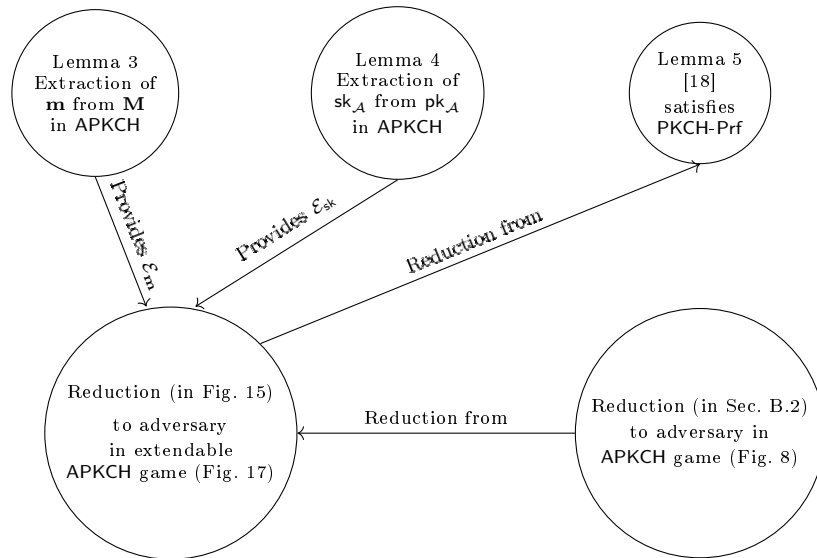


Fig. 12. Overview of the proof of Thm. 3

To make our proof simpler, we will first prove a lemma (Lemma 2) that we use in our proof.

Before we begin to write generic group model proofs, we first create a framework for proofs in the generic group model. We first define a generic group adversary in Def. 19 and informally define the GGM heuristic in Remark 1.

**Definition 19 (A generic group model adversary).** *A generic group model adversary is an adversary that receives a “group operation oracle” instead of knowing the exact parameters of the group which would allow the adversary to perform these operations itself. Instead, a challenger (or reduction) handles queries made to this oracle. Instead of returning group elements, the challenger returns encodings of elements. We define this oracle as  $\mathcal{O}^{\text{ggm}}(\text{elt}, s)$  where  $\text{elt}$  is an encoding of an element and  $s$  is a scalar in  $\mathbb{Z}_p$  that the adversary wishes to scale that element to (where  $p$  is the size of the generic group). If we are working with bilinear pairings, the adversary gets access to the oracles,  $\mathcal{O}^{\text{ggm},1}(\text{elt}, s)$ ,  $\mathcal{O}^{\text{ggm},2}(\text{elt}, s)$ ,  $\mathcal{O}^{\text{ggm},e}(\text{elt}_1, \text{elt}_2)$ , and  $\mathcal{O}^{\text{ggm},t}(\text{elt}, s)$ , for the first source group, sec-*

ond source group, pairing operation, and target group, respectively. The pairing operation takes in two encodings instead of a scalar.

*Remark 1 (The GGM heuristic (informal)).* A scheme that is proven to have certain properties against a generic group adversary has those properties in practice.

Remark 1 is false but generally holds for non-contrived schemes.

To ensure that the adversary’s view is correct, the challenger will usually maintain a map from group elements to encodings which we label  $\phi$ . At the beginning of the game, the challenger initializes this map with the parameters given to the adversary, which generally includes the generators of the bilinear pairing,  $P$  and  $\hat{P}$ . Thus, to begin, the challenger samples the encodings,  $e_{1,P} \leftarrow \{0,1\}^\lambda$  and  $e_{1,\hat{P}} \leftarrow \{0,1\}^\lambda$ , and then sets  $\phi(1,P) = e_{1,P}$  and  $\phi(1,\hat{P}) = e_{1,\hat{P}}$ . If the adversary queries  $\mathcal{O}^{\text{ggm},1}(P,\alpha)$ , the challenger will sample and return  $e_{\alpha,1}$  and set  $\phi(\alpha,P) = e_{\alpha,1}$ . If the adversary then samples  $\mathcal{O}^{\text{ggm},1}(P,1/\alpha)$ , the challenger finds that it must return  $e_{1,P}$  in order to create a valid view for the adversary.

A challenger using a generic adversary can choose not to sample values in the scheme. Instead, the challenger can leave these values as *indeterminate*. For example, for DDH, instead of sampling  $a, b$ , and  $c$ , the challenger can set the maps  $\phi(1,P^a) = e_{1,P^a}$ ,  $\phi(1,P^b) = e_{1,P^b}$  and  $\phi(1,P^c) = e_{1,P^c}$  without defining  $a, b$ , and  $c$ .

The usual proof strategy with the GGM is to leave all trapdoors of the scheme undefined until the end of the game and then use the Schwartz-Zippel lemma to prove that there’s only a negligible chance that challenger created an invalid view.

If DDH is secure in the generic group model (which it is) our challenger can create a map,  $\phi$ , that “works” for both cases (where  $c = ab$  or  $c \leftarrow \mathbb{Z}_p$ ) i.e. the encoding is a valid view of both schemes with high probability. In other words, a generic adversary cannot distinguish the case where  $c = ab$  or  $c \leftarrow \mathbb{Z}_p$  (the two experiments) without finding some query to the oracle that returns a different encoding in one of the games. We call this the “distinguishing query”. This allows us to state Lemma Lemma 1.

**Lemma 1 (Distinguishing polynomial).** *For any bit-guessing security game against a generic adversary (as defined in Def. 19), either the generic adversary cannot defeat the game or there exists an extractor that can extract a “distinguishing polynomial” such that the polynomial is identically zero in one experiment and identically non-zero in the other. A bit-guessing game is defined as a game where the adversary outputs a bit and their advantage wins if they guess the challenger’s random bit with greater than  $1/2 + \text{negl}(\lambda)$  chance.*

*Proof of Lemma 1.* We observe that any query the adversary makes to a single oracle could instead be made to a 'polynomial' oracle. We define this polynomial oracle as such:  $\mathcal{O}^{\text{ggm}}(\eta, \alpha, \beta, \kappa)$  where  $\eta$  refers to the scalar the adversary wishes to exponentiate  $P$  by,  $\alpha$  refers to the scalar the adversary wishes to exponentiate  $P^a$  by,  $\beta$  for  $P^b$ , and  $\kappa$  for  $P^c$ .

We call this the polynomial oracle because a reduction can represent queries as polynomials. For example, in the above query, the reduction can represent the adversary's query as  $P(\eta, \alpha, \beta, \kappa) = \eta + \alpha a + \beta b + \kappa c$ . The reduction can then map these polynomials to encodings.

If the adversary has an oracle in the experiment, we can update this GGM oracle. Perhaps the adversary has a signature oracle which returns a single element ( $\sigma$ ) as the signature. We can update this oracle as the adversary makes more and more queries. After the adversary makes one signature query, the challenger updates the oracle to accept  $\mathcal{O}^{\text{ggm}}(\eta, \alpha, \beta, \kappa, \delta_1)$  where  $\delta_1$  is the scalar the adversary wishes to exponentiate the first signature by. After  $q$  queries, the adversary has access to the oracle,  $\mathcal{O}^{\text{ggm}}(\eta, \alpha, \beta, \kappa, \delta_1, \dots, \delta_q)$ .

Further, we see that this distinguishing query must be equal to a polynomial that the adversary previously queried in one experiment, and distinct from that query in the other experiment. If this is not the case, the reduction could update the map,  $\phi$ , to make the adversary's view the same in both games.

Thus, if the adversary makes this distinguishing query to distinguish the two experiments, we find two polynomials in experiment 1,  $p_{1,1}$  and  $p_{2,1}$  such that  $p_{2,1}(K_2) - p_{1,1}(K_1) = 0$  where  $K_1$  and  $K_2$  are the sets of scalars the adversary used in these queries. But, in the second experiment,  $p'_{2,1}(K_2) - p'_{1,1}(K_1) \neq 0$ .

Thus, we see that if the adversary can distinguish the two experiments, the challenger must be able to sift through the adversary's queries to find a polynomial in one of the experiments that is identically zero, while in the other experiment, the polynomial is not identically zero.

We use this observation in our proofs by proving that there is no such polynomial that is identically zero in one experiment while being non zero in the other. Thus, the adversary must not have been able to distinguish the two games.

As an extra note, we can see that because we only increase the options of the GGM oracle as the adversary makes new queries, any distinguishing query the adversary could've made during the game could've instead been made at the end of the game, after they've finished querying the oracles. This observation simplifies our proofs.

**Lemma 2 (Extraction of  $m_i$  values from valid messages).** *In the APKCH hiding game, before the challenge, because the message satisfies  $\forall i \in [\ell], e(M_i, \hat{V}_i) = e(M_{i+\ell}, \hat{V}_{i+\ell})$ , the adversary must know the discrete logs,  $\{m_i\}_{i \in [\ell]}$ , such that  $m_i = \text{dlog}_{B_i}(M_i) = \text{dlog}_{B_{\ell+i}}(M_{\ell+i})$  for  $0 < i \leq \ell$ .*

*Proof intuition for Lemma 2.* We can see that if  $b_i, \hat{b}_i, \hat{v}_i$  are independently random, then, verifying messages using  $\hat{V}_i$  allows us to extract the  $m_i$  values in the generic group model. As a quick example to give the reader intuition as well as a review of the generic group model, we'll show this holds for messages when  $\ell = 2$ . Suppose an adversary supplies  $M = (M_1, M_2, M_3, M_4)$  such that  $e(M_1, \hat{V}_1) = e(M_3, \hat{V}_3)$  and  $e(M_2, \hat{V}_2) = e(M_4, \hat{V}_4)$ . We want to prove that we can extract  $m_1 = \text{dlog}_{B_1}(M_1) = \text{dlog}_{B_{\ell+1}}(M_3)$  and  $m_2 = \text{dlog}_{B_2}(M_2) = \text{dlog}_{B_{\ell+2}}(M_4)$ . By “extract” we mean that the adversary must query these values into the GGM oracle. A generic adversary in the GGM cannot compute the group operations themselves and instead must query an oracle with scalars and encodings of group elements to perform group operations. The adversary receives encodings of the generators of the generic group, as well as encodings of any elements that the challenger creates. Thus, for  $\text{pp}$  generated for  $\ell = 2$ , the adversary has encodings of  $P, B_1, B_2, B_{\ell+1}, B_{\ell+2}$ . Ignoring the signature oracle, the adversary must construct  $M_1$  from known encodings, i.e. the public parameters. This means they can choose a set of scalars:  $K^{(1)} = (\gamma^{(1)}, \alpha_1^{(1)}, \alpha_2^{(1)}, \kappa_1^{(1)}, \kappa_2^{(1)}) \in \mathbb{Z}_p$ , and query the group operation oracle for  $\mathbb{G}_1$  to compute:  $M_1 = P^{\gamma^{(1)}} B_1^{\alpha_1^{(1)}} B_2^{\alpha_2^{(1)}} B_{\ell+1}^{\kappa_1^{(1)}} B_{\ell+2}^{\kappa_2^{(1)}}$ . Let us also label their computation of  $M_3$  as  $M_3 = P^{\gamma^{(3)}} B_1^{\alpha_1^{(3)}} B_2^{\alpha_2^{(3)}} B_{\ell+1}^{\kappa_1^{(3)}} B_{\ell+2}^{\kappa_2^{(3)}}$ . Because  $e(M_1, \hat{V}_1) = e(M_3, \hat{V}_3)$  (and  $\text{dlog}_P(\hat{V}_1) = (\hat{v}_1 \hat{b}_1)$  and  $\text{dlog}_P(\hat{V}_3) = (\hat{v}_1)$ ) we know that:

$$\begin{aligned} & (\gamma^{(1)} + b_1 \alpha_1^{(1)} + b_2 \alpha_2^{(1)} + (b_1 \hat{b}_1) \kappa_1^{(1)} + (b_2 \hat{b}_2) \kappa_2^{(1)}) * \hat{v}_1 \hat{b}_1 \\ & = (\gamma^{(3)} + b_1 \alpha_1^{(3)} + b_2 \alpha_2^{(3)} + (b_1 \hat{b}_1) \kappa_1^{(3)} + (b_2 \hat{b}_2) \kappa_2^{(3)}) * \hat{v}_1 \end{aligned} \quad (2)$$

We can simplify this by dividing both sides by  $1/(b_1 \hat{b}_1)$ :

$$\begin{aligned} & (\gamma^{(1)} + b_1 \alpha_1^{(1)} + b_2 \alpha_2^{(1)} + (b_1 \hat{b}_1) \kappa_1^{(1)} + (b_2 \hat{b}_2) \kappa_2^{(1)}) * \hat{v}_1 / b_1 \\ & = (\gamma^{(3)} + b_1 \alpha_1^{(3)} + b_2 \alpha_2^{(3)} + (b_1 \hat{b}_1) \kappa_1^{(3)} + (b_2 \hat{b}_2) \kappa_2^{(3)}) * \hat{v}_1 / (b_1 \hat{b}_1) \end{aligned} \quad (3)$$

Distributing Eq. 3 we get Eq 4:

$$\begin{aligned} & (\gamma^{(1)} \hat{v}_1 / b_1 + \alpha_1^{(1)} \hat{v}_1 + b_2 \alpha_2^{(1)} \hat{v}_1 / b_1 + \hat{b}_1 \kappa_1^{(1)} \hat{v}_1 + b_2 \hat{b}_2 \kappa_2^{(1)} \hat{v}_1 \hat{b}_1) \\ & = (\gamma^{(3)} \hat{v}_1 / (b_1 \hat{b}_1) + \alpha_1^{(3)} \hat{v}_1 / \hat{b}_1 + b_2 \alpha_2^{(3)} \hat{v}_1 / (b_1 \hat{b}_1) + \kappa_1^{(3)} \hat{v}_1 + b_2 \hat{b}_2 \kappa_2^{(3)} \hat{v}_1 / (b_1 \hat{b}_1)) \end{aligned} \quad (4)$$

This implies that when you subtract the right side of Eq. 4 from the left, it equals 0. The general proof strategy in the GGM is to leave values that the adversary does not know undefined and instead have the oracle remember algebraic relations between encodings to return encodings such that the adversary

will not notice that the challenger has not initialized the undefined variables. We call these undefined variables “indeterminate”. The oracle then samples the indeterminate after the game is complete and proves that there is a negligible chance that the adversary’s scalar operations resulted in winning the security game (e.g. creating a forgery). We can see that every term on the right side of Eq. 4 includes indeterminate  $1/\hat{b}_1$  except for  $\hat{v}_1\kappa_1^{(3)}$ . On the left side, every term has  $1/b_1$  or  $\hat{b}_1$  except for  $\hat{v}_1\alpha_1^{(1)}$ . Thus, because  $\hat{b}_1$  and  $b_1$  are indeterminate, the only way for the adversary to satisfy this relation in the GGM after the GGM oracle instantiates variables  $b_1$  and  $\hat{b}_1$  randomly is to set  $\kappa_1^{(3)} = \alpha_1^{(1)}$  and leave all other chosen values 0. This gives us  $M_1 = B_1\alpha_1^{(1)}$  and  $M_3 = B_{\ell+1}\kappa_1^{(3)}$ , allowing a reduction to extract an  $m_1$  value that satisfies  $M_1 = B_1^{m_1}$  and  $M_3 = B_{\ell+1}^{m_1}$ . While the  $\hat{v}_1$  and  $\hat{v}_2$  values seem unnecessary here, they ensure that an adversary that knows  $m_1, m_2$  cannot use the bases  $(\hat{V}_i)$  to recognize a message after it has been randomized. We now introduce the formal proof of Lemma 2.

*Proof of Lemma 2.* To prove this, we’ll first show that using the public parameters alone, a generic adversary cannot produce a message where they do not know the correct discrete log.

The adversary sees  $\mathbf{pp} = \{B_i, \hat{B}_i, B_{\ell+i}, \hat{B}_{\ell+i}, V_i, V_{i+\ell}, \hat{V}_i, \hat{V}_{i+\ell}\}_{i \in [\ell]}$ , where  $\forall i \in [\ell], B_i \leftarrow P^{b_i}, \hat{B}_i \leftarrow \hat{P}^{\hat{b}_i}, B_{\ell+i} \leftarrow P^{b_i b_i}, \hat{B}_{\ell+i} \leftarrow \hat{P}^{\hat{b}_i \hat{b}_i}, \hat{V}_i = \hat{P}^{\hat{v}_i/b_i}, \hat{V}_{i+\ell} = \hat{P}^{\hat{v}_{i+\ell}/b_i \hat{b}_i}, V_i = P^{v_i/\hat{b}_i}, V_{i+\ell} = P^{v_i/b_i \hat{b}_i}$ .

The adversary sees  $\{B_i, B_{\ell+i}, V_i, V_{i+\ell}\}_{i \in [\ell]}$  in  $\mathbb{G}_1$  and can exponentiate these values to create a message. Thus, we can represent any element in  $\mathbb{G}_1$  that the adversary can create with the following polynomial:  $Q(\{\alpha_i, \kappa_i, \nu_i, \nu_{i+\ell}\}_{i \in [\ell]}) = \sum_{i \in [\ell]} \alpha_i b_i + \sum_{i \in [\ell]} \kappa_i b_i \hat{b}_i + \sum_{i \in [\ell]} \nu_i v_i / \hat{b}_i + \sum_{i \in [\ell]} \nu_{i+\ell} v_{i+\ell} / (b_i \hat{b}_i)$ , where  $\{\alpha_i, \kappa_i, \nu_i, \nu_{i+\ell}\}_{i \in [\ell]}$  can be chosen by the adversary. We’ll look at one verification equation for  $M_j$  and  $M_{j+\ell}$  and label  $Q(K)$  where  $K = \{\alpha_i, \kappa_i, \nu_i, \nu_{i+\ell}\}_{i \in [\ell]}$  as the computation of  $M_j$  and  $Q'(K')$  where  $K' = \{\alpha'_i, \kappa'_i, \nu'_i, \nu'_{i+\ell}\}_{i \in [\ell]}$  as the computation for  $M_{j+\ell}$ . We see that because  $e(M_j, \hat{V}_j) = e(M_{j+\ell}, \hat{V}_{j+\ell})$ , we must have that  $\hat{v}_j Q(K) / b_j = \hat{v}_j Q'(K') / (b_j \hat{b}_j)$ . We can multiply this equation on either side by  $b_i \hat{b}_i$  to we see that  $Q(K) \hat{b}_j = Q'(K')$ . Thus, it must be that:

$$\sum_{i \in [\ell]} \alpha_i b_i \hat{b}_j + \sum_{i \in [\ell]} \kappa_i b_i \hat{b}_i \hat{b}_j + \sum_{i \in [\ell]} \nu_i v_i \hat{b}_j / \hat{b}_i + \sum_{i \in [\ell]} \nu_{i+\ell} v_{i+\ell} \hat{b}_j / (b_i \hat{b}_i) = \sum_{i \in [\ell]} \alpha'_i b_i + \sum_{i \in [\ell]} \kappa'_i b_i \hat{b}_i + \sum_{i \in [\ell]} \nu'_i v_i / \hat{b}_i + \sum_{i \in [\ell]} \nu'_{i+\ell} v_{i+\ell} / (b_i \hat{b}_i)$$

We can quickly see that these polynomials will certainly not be equal if we have any  $\alpha_i, \kappa_i, \nu_i, \nu_{i+\ell} \neq 0$  where  $i \neq j$  since in this case, the left equation would have some  $b_i \hat{b}_j$  or  $\hat{b}_j / \hat{b}_i$  term which the right equation does not. From this, we can deduce that for all  $i \neq j$ ,  $\alpha'_i, \kappa'_i, \nu'_i, \nu'_{i+\ell} = 0$ , since we established that for any term where  $i \neq j$ , the left side must be zero and thus including any non-zero indeterminate on the right side of the equation would unbalance the equation.

Thus, we have that only the bases,  $\{B_j, B_{\ell+j}, V_j, V_{\ell+j}\}$  are used in the computation of  $M_j, M_{j+\ell}$  for any  $j$  (critically,  $\{B_i, B_{\ell+i}, V_i, V_{\ell+i}\}_{i \in [\ell] \setminus \{j\}}$  are excluded

from the computation). Thus, we have the following equation for the verification of  $M_j$  and  $M_{j+\ell}$ :

$$\alpha_j b_j \hat{b}_j + \kappa_j b_j \hat{b}_j^2 + \nu_j v_j + \nu_{j+\ell} v_j / (b_j) = \alpha'_j b_j + \kappa'_j b_j \hat{b}_j + \nu'_j v_j / \hat{b}_j + \nu'_{j+\ell} v_j / (b_j \hat{b}_j)$$

From this, we can see that this is only satisfied when  $\alpha_j = \alpha'_j$  and all other values chosen by the adversary are 0. Thus, we can extract  $m_i = \alpha_j = \kappa'_j = \text{dlog}_{B_i}(M_i) = \text{dlog}_{B_{\ell+i}}(M_{i+\ell})$ .

Next, we need to show that even after seeing signatures, the adversary cannot include secrets from the signatures in their messages. Since the adversary cannot create a message without knowing the discrete log using the parameters, and the public key is in  $\mathbb{G}_2$ , we know the message in the first query to the signing oracle is independent of any of the challenger's secret. Now, we'll assume this for query  $n$  and show that for  $n + 1$  this holds (proof by induction).

For query  $j$ , the adversary chooses a message,  $\{\mu_{i,j}\}_{i \in [\ell]}$ , and the challenger signs it, yielding  $Z_j = \prod_{i,j} P^{\mu_{i,j} b_i \hat{b}_i x_i y_j}$ ,  $Y_j = P^{1/y_j}$  in  $\mathbb{G}_1$ . After query  $n$ , the adversary has seen  $\{Z_j, Y_j\}_{j \in [n]}$  in  $\mathbb{G}_1$  along with the public parameters. Let  $K$  be the set of the adversary's choices.  $K = \{\mu_{i,j}, \gamma_j, \alpha_i, \kappa_i, \nu_i, \nu_{i+\ell}\}_{j \in [n], i \in [\ell]}$  where  $\alpha_i, \kappa_i, \nu_i, \nu_{i+\ell}$  are defined similar to their previous definition in this proof,  $\mu_{i,j}$  is the choice of message, and  $\gamma_j$  is the scalar for  $Y_j$ . We can update the polynomial representing any of the adversary's evaluations in  $\mathbb{G}_1$  from the last proof (when the adversary only had  $\text{pp}$ ). We'll split this polynomial ( $R$ ) into the sum of two polynomial:  $Q$  (from the last proof) and  $S$  using signatures (new to this proof).  $R(K) = Q(K_Q) + S(K_S)$  where  $K_Q = \{\alpha_i, \kappa_i, \nu_i, \nu_{i+\ell}\}_{i \in [\ell]}$  and  $K_S = \{\mu_{i,j}, \gamma_j\}_{i \in [\ell], j \in [n]}$ .  $S$  is defined as  $S(K_S) = \sum_{j \in [n]} (\sum_{i \in [\ell]} \mu_{i,j} \hat{b}_i b_i x_i y_j) + \sum_{j \in [n]} \gamma_j / y_j$ . We do not include a direct scalar for exponentiating  $Z_j$  values because any scaling of  $Z_j$  by the adversary can be countered by multiplying with  $\mu_{i,j}$ , as the relation is proportional.

Again, focusing on one verification equation for index  $k$ :

If  $e(M_k, \hat{V}_k) = e(M_{k+\ell}, \hat{V}_{k+\ell})$  we see that  $\hat{b}_k R(K) = R'(K')$  (where  $R(K)$  is the adversary's computation of  $M_k$  and  $R'(K')$  is the computation of  $M_{k+\ell}$ ). Because  $S$  is dependent on  $y_j$  which does not appear in  $Q$ , no term in  $Q$  can be used to cancel these terms out. Thus, it must be that:  $\hat{b}_k S(K_S) = S'(K'_S)$ . This means that:

$$\sum_{j \in [n]} (\sum_{i \in [\ell]} \mu_{i,j} \hat{b}_k b_i \hat{b}_i x_i y_j) + \sum_{j \in [n]} \gamma_j \hat{b}_k / y_j = \sum_{j \in [n]} (\sum_{i \in [\ell]} \mu'_{i,j} b_i \hat{b}_i x_i y_j) + \sum_{j \in [n]} \gamma'_j / y_j$$

Where  $K'_S = \{\mu'_{i,j}, \gamma'_j\}_{i \in [\ell], j \in [n]}$  are the adversary's choices from  $K'$  used in the computation of  $Q'(K'_S)$  used to compute  $M_{k+\ell}$ . We can see that all terms on the left side of the equation include the indeterminate  $\hat{b}_k$  whereas on the right side, only one includes  $\hat{b}_k$ . Thus, it must be that:

$$\sum_{j \in [n]} (\sum_{i \in [\ell]} \mu_{i,j} \hat{b}_k b_i \hat{b}_i x_i y_j) + \sum_{j \in [n]} \gamma_j \hat{b}_k / y_j = (\sum_{i \in [\ell]} \mu'_{i,j} b_i \hat{b}_i x_i y_k) + \gamma'_k / y_k$$

We can see that on the left side, each term includes some  $y_j$  when  $j \neq k$  and thus it must be that:

$$\left(\sum_{i \in [\ell]} \mu_{i,j} \hat{b}_k b_i \hat{b}_i x_i y_k\right) + \gamma_k \hat{b}_k / y_k = \left(\sum_{i \in [\ell]} \mu'_{i,j} \hat{b}_i \hat{b}_k x_i y_k\right) + \gamma'_k / y_k$$

We can see that terms on the left side include either  $\hat{b}_k \hat{b}_i$  or  $\hat{b}_k / y_k$  while indeterminate of this form are absent on the right side. Thus, it must be that  $\forall i \in [\ell], j \in [n], \mu_{i,j} = 0, \gamma_j = 0$ .

Thus, after  $n$  queries to the oracle, the adversary must still know the discrete log of the message, and thus by induction, must always know the discrete logs of messages during the game.  $\square$

Recall (see Fig. 2) the security game. After the adversary interacts with the signing oracle for  $\mathbf{pk}$  for a while, it is ready to receive its challenge: a public key  $\mathbf{pk}^b$  that is either in the same equivalence class as  $\mathbf{pk}$  or in the same class as  $\mathbf{pk}_A$  chosen by the adversary; as well as the signature  $\sigma^b$  under  $\mathbf{pk}^b$  on the message  $M$  of the adversary's choice. The adversary may continue making further queries to the signing oracle for the original public key  $\mathbf{pk}$ . After receiving  $(\mathbf{pk}^b, \sigma^b)$ , the adversary may be able to use the fact that, in case  $b = 1$ , the signature  $\sigma^b$  will include the adversary's own secrets. For example, the adversary can incorporate these secrets into the messages they choose to query to the signing oracle. Thus, we need to prove a second lemma to ensure that the adversary does not include elements of previous signatures in their queries after receiving the challenge.

**Lemma 3 (Extraction of  $m_i$  values from valid messages after the challenge).** *In the APKCH hiding game, because the message satisfies  $\forall i \in [\ell], e(M_i, \hat{V}_i) = e(M_{i+\ell}, \hat{V}_{i+\ell})$ , the adversary must know the discrete logs,  $\{m_i\}_{i \in [\ell]}$ , such that  $m_i = \text{dlog}_{B_i}(M_i) = \text{dlog}_{B_{\ell+i}}(M_{\ell+i})$  for  $0 < i \leq \ell$ .*

*Proof of Lemma 3.* We can apply our logic from the proof of Lemma 2 to ensure that when  $b = 0$  the adversary cannot make their messages depend on secrets from the signature since the returned challenge will be distributed identically to other queries. This is because the challenge signature comes from the same oracle that the adversary has already been receiving signatures on, it is just randomized. Thus, fixing  $b = 0$ , if an adversary exists that can break key extraction with the challenge, we can reduce to a reduction that does not receive this challenge, but instead simply queries one more signature, randomizes it, and hands it to this adversary as the challenge.

When  $b = 1$ , the adversary gains a values in  $\mathbb{G}_1$  which are distinct from previous signatures. Specifically, they learn:  $Z_c = P^{\sum_{i \in [\ell]} \mu_{i,c} \rho_c \chi_i y_c}$  and  $Y_c = P^{1/y_c}$  where  $\{\chi_i\}_{i \in [\ell]}$  is the adversary's secret key,  $\{\mu_{i,c}\}_{i \in [\ell]}$  is the adversary's submitted message,  $\rho_c$  is the randomization that the challenger computes, and  $y_c$  is the randomizer used in the signature. As in the proof of Lemma 2, we split the polynomials representing  $M_k$  and  $M_{k+\ell}$  into sums of polynomials  $Q$  and  $S$ . But, we include  $Z_c$  in the computation of  $S$  such that for  $K_S = \{\mu_{i,c}, \gamma_c, \mu_{i,j}, \gamma_j\}_{i \in [\ell], j \in [n]}$ ,  $S(K_S) = \sum_{j \in [n]} (\sum \mu_{i,j} \hat{b}_i \hat{b}_i x_i y_j) + \sum_{j \in [n]} \gamma_j / y_j + (\sum \mu_{i,c} \hat{b}_i \rho_c y_c) + \gamma_c / y_c$ . Again,



because the adversary could simply compute  $\mu'_{i,c} = \chi_i \mu_{i,c}$ , we do not include  $\chi_i$  in this polynomial WLOG.

Again, we see that all terms in  $S(K_S)$  include  $y_c$  or  $y_j$  and thus, no term in  $Q(K_Q)$  can cancel these indeterminate out. Thus it must be that  $S(K_S)\hat{b}_k = S(K'_S)$

$$\sum_{j \in [n]} (\sum \mu_{i,j} \hat{b}_k b_i \hat{b}_i x_i y_j) + \sum_{j \in [n]} \gamma_j \hat{b}_k / y_j + (\sum \mu_{i,c} \hat{b}_k b_i \hat{b}_i \rho_c y_c) + \gamma_c \hat{b}_k / y_c = \sum_{j \in [n]} (\sum \mu'_{i,j} b_i \hat{b}_i x_i y_j) + \sum_{j \in [n]} \gamma'_j / y_j + (\sum \mu'_{i,c} b_i \hat{b}_i \rho_c y_c) + \gamma'_c / y_c$$

Using similar logic from the proof of Lemma 2 we can see that  $\forall i \in [\ell], j \in [n], \mu_{i,j} = 0, \mu_{i,c} = 0, \gamma_j = 0$  and  $\gamma_c = 0$ . The logic from the proof of Lemma 2 holds because it only relies on the presence of  $\hat{b}_k$  and not on  $x_i$  which is the only difference that the challenge signature has.  $\square$

Before we continue, we need to prove that we can extract the secret key from the adversary's public key in the experiment in Fig. 2. We define this property in Def. 20.

**Definition 20 (Key extractability).** *For all parameters  $\lambda, \ell$  and any PPT algorithm,  $\mathcal{A}$ , there exists an efficient extraction algorithm,  $\mathcal{E}_{\text{sk}}$  and negligible function,  $\text{negl}$  such that:*

$$\Pr \left[ \begin{array}{l} \text{VerifyKey}(\text{pp}, \text{pk}^A) = 1 \wedge \\ (\text{sk}^A, \text{pk}^A) \notin \text{KGen}(\text{pp}) \end{array} \middle| \begin{array}{l} \text{pp} \leftarrow \text{Setup}(1^\lambda, 1^\ell); (\text{sk}, \text{pk}) \leftarrow \text{KGen}(\text{pp}); \\ \text{pk}^A \leftarrow \mathcal{A}^{\text{Sign}(\text{sk}, \cdot)}(\text{pp}); \text{sk}^A \leftarrow \mathcal{E}_{\text{sk}}(\text{pk}^A) \end{array} \right] \leq \text{negl}(\lambda)$$

**Lemma 4 (Extraction of the secret key ( $x_i$ ) values from valid public keys).** *The mercurial signature construction in Fig. 3 has key extraction as defined in Def. 20.*

*Proof of Lemma 4.* Intuitively, in the APKCH hiding game, because the public key satisfies  $\forall i \in [\ell], e(V_i, \hat{X}_i) = e(V_{i+\ell}, \hat{X}_{i+\ell})$ , the adversary must know the discrete logs,  $\{x_i\}_{i \in [\ell]}$ , such that  $x_i = \text{dlog}_{\hat{B}_i}(\hat{X}_i) = \text{dlog}_{\hat{B}_{\ell+i}}(\hat{X}_{\ell+i})$  for  $0 < i \leq \ell$ .

More formally, the adversary sees  $\text{pp} = \{B_i, \hat{B}_i, B_{\ell+i}, \hat{B}_{\ell+i}, V_i, V_{i+\ell}, \hat{V}_i, \hat{V}_{i+\ell}\}_{i \in [\ell]}$ , where  $\forall i \in [\ell], B_i \leftarrow P^{b_i}, \hat{B}_i \leftarrow \hat{P}^{\hat{b}_i}, B_{\ell+i} \leftarrow P^{b_i \hat{b}_i}, \hat{B}_{\ell+i} \leftarrow \hat{P}^{\hat{b}_i \hat{d}_i}, \hat{V}_i = \hat{P}^{\hat{v}_i \hat{b}_i}, \hat{V}_{i+\ell} = \hat{P}^{\hat{v}_i}, V_i = P^{v_i \hat{d}_i}, V_{i+\ell} = P^{v_i}$ .

After  $q$  queries to the signature oracle, the GGM adversary will create the following polynomials in  $\mathbb{G}_2$ :

$$P(K) = \sum \beta_i \hat{b}_i + \sum \beta_{\ell+i} \hat{b}_i \hat{d}_i + \sum \gamma_j y_j + \nu_i \hat{v}_i \hat{b}_i + \sum \nu_i \hat{v}_i$$

Where  $K$  is a set of the adversary's choice of  $\beta_i, \gamma_j$ , and  $\nu_i$ .

Because the adversary's public key verifies, i.e.  $e(V_i, \hat{X}_i) = e(V_{\ell+i}, \hat{X}_{\ell+i})$ , we know that:

$$\hat{d}_i P(K_i) = P(K_{\ell+i})$$

Thus,  $P(K_{\ell+i})$  must contain  $\hat{d}_i$  or  $P(K)$  must be zero, which can be easily detected and rejected. We can see that the only term in  $P(K)$  that includes  $\hat{d}_i$  is  $\beta_{\ell+i}\hat{b}_i\hat{d}_i$ . Thus, the adversary must use this term in the computation of  $P(K_{\ell+i})$ .

We can see that this includes the indeterminate  $\hat{b}_i$ . Thus, the adversary must include the terms  $\beta_i\hat{b}_i$  or  $\nu_i\hat{v}_i\hat{b}_i$  in the computation of  $P(K_i)$ . If the adversary were to include  $\nu_i\hat{v}_i\hat{b}_i$  in the computation of  $P(K_i)$ , we can see that no other term includes  $\hat{d}_i\hat{v}_i\hat{b}_i$ . Thus, there would be no way for the adversary to balance this equation. Therefore, the adversary must only use the term  $\beta_i\hat{b}_i$  for  $P(K_i)$  and  $\beta_{\ell+i}\hat{b}_i\hat{d}_i$  for  $P(K_{\ell+i})$ , meaning we can extract these values as the correct secret key.

After the challenge, the adversary only learns a single additional element in  $\mathbb{G}_2$ ,  $\hat{Y} = \hat{P}^{1/y}$ . The discrete log of this element is independent of any other elements that the adversary saw in  $\mathbb{G}_2$  so it does not help the adversary incorporate it into their computation of any verifying public key.

*Proof of Thm. 3.* In proving Thm. 3, we'll see that the following game will become useful. We will prove that the construction in [18] meets this definition. This game is similar to the public key class-hiding game in [18], but the adversary receives some structured bases as  $\mathbf{B}^l, \mathbf{B}^u$ , and  $\hat{\mathbf{B}}^l$  are related in a DDH type of way. They also effectively can only query the oracle once, which is modeled by having the adversary output the message that they want signed and then calling them with the signature later. Later, our reduction will play the role of the adversary in this experiment.

**Lemma 5.** *For any PPT generic adversary,  $\mathcal{A} = \{\mathcal{A}_1, \mathcal{A}_2\}$ , we have that  $\left| \Pr \left[ \mathbf{Exp}_{\mathcal{A}}^{\text{PKCH-Prf},0} = 1 \right] - \Pr \left[ \mathbf{Exp}_{\mathcal{A}}^{\text{PKCH-Prf},1} = 1 \right] \right|$  is negligible. Where  $\mathbf{Exp}_{\mathcal{A}}^{\text{PKCH-Prf},1}$  is described in Fig. B.1.*

```

{ $b_i, \hat{b}_i, e_i\}_{i \in [\ell]}, \rho, y \leftarrow \mathbb{Z}_p$ 
B1 =  $\{B_i\}_{i \in [\ell]}$  where  $B_i = P^{b_i}$ 
Bu =  $\{B_{\ell+i}\}_{i \in [\ell]}$  where  $B_{\ell+i} = P^{b_i \hat{b}_i}$ 
B1 =  $\{B_i\}_{i \in [\ell]}$  where  $\hat{B}_i = \hat{P}^{\hat{b}_i}$ 
E0 =  $\{E_i^0\}_{i \in [\ell]}$  where  $\hat{E}_i^0 = \hat{P}^{\rho \hat{b}_i}$ 
E1 =  $\{E_i^1\}_{i \in [\ell]}$  where  $\hat{E}_i^1 = \hat{P}^{e_i}$ 
m  $\leftarrow \mathcal{A}_1(\mathbf{B}^1, \mathbf{B}^u, \hat{\mathbf{B}}^1, \hat{\mathbf{E}}^b)$ 
 $Z^0 = (\prod P^{m_i \rho \hat{b}_i})^y, Y^0 = P^{1/y}, \hat{Y}^0 = \hat{P}^{1/y}$ 
 $Z^1 = (\prod P^{m_i e_i})^y, Y^1 = P^{1/y}, \hat{Y}^1 = \hat{P}^{1/y}$ 
 $b' \leftarrow \mathcal{A}_2(Z^b, Y^b, \hat{Y}^b)$ 
return  $b'$ 

```

**Fig. 13.** Experiment  $\text{Exp}_{\mathcal{A}}^{\text{PKCH-Prf}, b}$  used in Proof of Thm. 3.

*Proof of Lemma 5.* We will prove this in the generic group model. In the generic group model, the goal of any adversary is to find a computation that is equivalent to  $e(P, \hat{P})$  when  $b = b'$  but is not equal to  $e(P, \hat{P})$  when  $b = 1 - b'$ .

**Lemma 6.** *If the adversary does not use  $B_{\ell+i}$  values in their distinguishing polynomial, we can reduce to PKCH of the mercurial signature construction in [18].*

*Proof of Lemma 6.* Intuitively, this Lemma holds because  $B_{\ell+i}$  is the only element that differs from the PKCH game and everything else, a reduction can generate. This can be seen by looking at computations that the adversary can create without these elements (disregarding  $Z$  for now):

$$e(P, \hat{P})^\eta \prod e(P, \hat{B}_i)^{\beta_i^l} \prod e(P, \hat{E}_i^b)^{\epsilon_i} \prod e(B_i, \hat{P})^{\alpha_i^l} \prod e(B_i, \hat{B}_j)^{\gamma_{\beta, j}^{\alpha, i}} \prod e(B_i, \hat{E}_j^b)^{\gamma_{\epsilon, j}^{\alpha, i}}$$

This polynomial appears as:

$$Q(K) = \eta + \sum \hat{b}_i * \beta_i^l + \sum e_i * \epsilon_i + \sum b_i * \alpha_i^l + \sum b_i * \hat{b}_j * \gamma_{\beta, j}^{\alpha, i} + \sum b_i * e_j * \gamma_{\epsilon, j}^{\alpha, i}$$

We can create a reduction playing the PKCH game with the CL19 construction which generates these trapdoors,  $b_i$  itself. Because we are assuming that  $B_{\ell+i}$  is never included in any distinguishing polynomial, our reduction does not need to know  $\hat{b}_i$  to create encoding when the adversary exponentiates it. Instead, the reduction simply leaves the discrete log of each  $B_{\ell+i}$  as an indeterminate and maps any encoding of this element to be the same as  $e(B_i, \hat{B}_j)$ . After this adversary finds a distinguishing polynomial, our reduction removes their own trapdoors from the polynomial, leaving:  $Q'(K) = \eta + \sum \hat{b}_i * \beta_i^l + \sum e_i * \epsilon_i + \sum \alpha_i^l + \sum \hat{b}_j * \gamma_{\beta, j}^{\alpha, i} + \sum e_j * \gamma_{\epsilon, j}^{\alpha, i}$

We can see that if  $Q(K)$  is a distinguishing polynomial in the game in Fig. B.1, then  $Q'(K)$  is a distinguishing polynomial in the CL19 game when we replace  $\hat{B}_{\ell+i}$  with elements from the  $\text{pk}_1$  from the CL19 challenger and replace  $\hat{E}_i^b$  with

elements from the  $\text{pk}_2^b$  challenger. Thus, the adversary must include one of these elements ( $B_{\ell+i}$  or  $Z$ ) in their distinguishing polynomial, otherwise, we break the PKCH of the construction in CL19. We have a similar argument to Lemma 6 for proving that the distinguishing polynomial cannot be independent of  $B_{\ell+i}$  as our reduction can generate  $Z^b$  using the CL19 challenger.  $\square$

Thus, using Lemma 6, we find that the distinguishing polynomial must include  $B_{\ell+i}$ .

The only elements that have  $b_i$  in them are  $B_i$  and  $B_{\ell+i}$ . Thus, for the adversary to win, they must be able to cancel out the  $B_{\ell+i}$  that they include in their computation with one of these elements.

The adversary must also include the  $Z$  or  $\hat{E}_i^b$  elements as these are the only elements that differ in the two games.

We can see that (without using  $Z^b$ ) any computation with  $Y$  or  $\hat{Y}$  will only add the indeterminate  $1/y$  to the computation, any these could simply be replaced with  $P$  with the same result. So, we can ignore these for now.

Because the  $Z^b$  values are independent of the  $b_i$  values, they cannot be used to cancel out the given  $B_{\ell+i}$ . These might still be useful if the adversary is able to first cancel out the  $b_i$  present in their computation of  $B_{\ell+i}$ . Thus, if we can first prove that the adversary cannot cancel out the  $b_i$  in their computation without the  $Z^b$  values, there's nothing the adversary can do to use these  $Z^b$  values to obtain an equation that results in  $e(P, \hat{P})$ .

In a similar argument, no computation without  $B_i$  or  $B_{\ell+i}$  can be used to cancel out the  $b_i$  values.

Thus, the part of the adversary's computation that cancels out  $b_i$  appears as the following equation where the adversary can choose to include or remove specific pairings:

$$\prod e(B_i, \hat{P})^{\gamma^{\alpha,i}} \prod e(B_i, \hat{B}_j)^{\gamma_{\beta,j}^{\alpha,i}} \prod e(B_i, \hat{E}_j^b)^{\gamma_{\eta,j}^{\alpha,i}} \prod e(B_{\ell+i}, \hat{P})^{\gamma^{\kappa,i}} \prod e(B_{\ell+i}, \hat{B}_j)^{\gamma_{\beta,j}^{\kappa,i}} \prod e(B_{\ell+i}, \hat{E}_j^b)^{\gamma_{\eta,j}^{\kappa,i}}$$

In order to cancel out an  $b_i$ , there must be some division of elements with this value. For any  $B_i$  or  $B_{\ell+i}$  used in the numerator of the computation, there must be another in the denominator.

$$\text{Thus, } \forall i, \gamma^{\alpha,i} + \gamma_{\beta,j}^{\alpha,i} + \gamma_{\eta,j}^{\alpha,i} + \gamma^{\kappa,i} + \gamma_{\beta,j}^{\kappa,i} + \gamma_{\eta,j}^{\kappa,i} = 0$$

Let's look at the discrete log of this equation:

If  $b = 0$ :

$$\sum \gamma^{\alpha,i} b_i + \sum \gamma_{\beta,j}^{\alpha,i} b_i \hat{b}_j + \sum \gamma_{\eta,j}^{\alpha,i} b_i \rho \hat{b}_j + \sum \gamma^{\kappa,i} b_i \hat{b}_i + \sum \gamma_{\beta,j}^{\kappa,i} b_i \hat{b}_i \hat{b}_j + \sum \gamma_{\eta,j}^{\kappa,i} b_i \hat{b}_i \rho \hat{b}_j$$

If  $b = 1$ :

$$\sum \gamma^{\alpha,i} b_i + \sum \gamma_{\beta,j}^{\alpha,i} b_i \hat{b}_j + \sum \gamma_{\eta,j}^{\alpha,i} b_i e_j + \sum \gamma^{\kappa,i} b_i \hat{b}_i + \sum \gamma_{\beta,j}^{\kappa,i} b_i \hat{b}_i \hat{b}_j + \sum \gamma_{\eta,j}^{\kappa,i} b_i \hat{b}_i e_j$$

We can see that if the adversary sets  $\gamma^{\alpha,i} > 0$ , they have a  $e(P, \hat{P})^{b_i}$  term. Because each other term has  $\hat{b}_i$ , this cannot be removed from the polynomial. Thus, it must be that  $\gamma^{\alpha,i} = 0$ .

Thus, we have the two computations:

If  $b = 0$ :

$$\sum \gamma_{\beta,j}^{\alpha,i} b_i \hat{b}_j + \sum \gamma_{\eta,j}^{\alpha,i} b_i \rho \hat{b}_j + \sum \gamma^{\kappa,i} b_i \hat{b}_i + \sum \gamma_{\beta,j}^{\kappa,i} b_i \hat{b}_i \hat{b}_j + \sum \gamma_{\eta,j}^{\kappa,i} b_i \hat{b}_i \rho \hat{b}_j$$

If  $b = 1$ :

$$\sum \gamma_{\beta,j}^{\alpha,i} b_i \hat{b}_j + \sum \gamma_{\eta,j}^{\alpha,i} b_i e_j + \sum \gamma^{\kappa,i} b_i \hat{b}_i + \sum \gamma_{\beta,j}^{\kappa,i} b_i \hat{b}_i \hat{b}_j + \sum \gamma_{\eta,j}^{\kappa,i} b_i \hat{b}_i e_j$$

It must be that some  $\gamma_{\eta,j}^{\alpha,i}, \gamma_{\eta,j}^{\kappa,i}$  must be non-zero otherwise the equation would always look the same in both games.

These contain either  $\rho$  or  $e_j$  which are not included in the other terms. Thus, they can only be canceled out by each other and thus, the adversary must be able to cancel them out with the following computations:

If  $b = 0$ :

$$\sum \gamma_{\eta,j}^{\alpha,i} b_i \rho \hat{b}_j + \sum \gamma_{\eta,j}^{\kappa,i} b_i \hat{b}_i \rho \hat{b}_j$$

If  $b = 1$ :

$$\sum \gamma_{\eta,j}^{\alpha,i} b_i e_j + \sum \gamma_{\eta,j}^{\kappa,i} b_i \hat{b}_i e_j$$

We can see that the second term in  $b = 0$  contains  $\hat{b}_i \hat{b}_j$ , which the first term does not. Thus, this polynomial must be zeroed out on its own. Similarly for when  $b = 1$ , the second term contains  $\hat{b}_i$  while the first term does not. Thus, these terms must zero out independently of one another. Thus, we can analyze them independently.

Case 1:

If  $b = 0$ :

$$\sum \gamma_{\eta,j}^{\alpha,i} b_i \rho \hat{b}_j$$

If  $b = 1$ :

$$\sum \gamma_{\eta,j}^{\alpha,i} b_i e_j$$

Case 2:

If  $b = 0$ :

$$\sum \gamma_{\eta,j}^{\kappa,i} b_i \hat{b}_i \rho \hat{b}_j$$

If  $b = 1$ :

$$\sum \gamma_{\eta,j}^{\kappa,i} b_i \hat{b}_i e_j$$

We'll deal with case 1 first. We see that any  $b_i$  cannot be cancelled out for any term that includes a different  $b_j$ . Thus, the computation for a fixed  $i$  must be equal to  $e(P, \hat{P})$ . We can see that

Thus, for a given  $i$ ,

If  $b = 0$ :

$$\sum_j \gamma_{\eta,j}^{\alpha,i} b_i \rho \hat{b}_j = e(P, \hat{P})$$

If  $b = 1$ :

$$\sum_j \gamma_{\eta,j}^{\alpha,i} b_i e_j = e(P, \hat{P})$$

We can see that each of these terms have a distinct  $\hat{b}_j$  or  $e_j$  and thus they can only be zero when  $\gamma_{\eta,j}^{\alpha,i} = 0$  for all  $j$ .

Moving on to case 2, we can use the same logic from case 1 to fix an  $i$  value.

For a given  $i$ ,

If  $b = 0$ :

$$\sum_j \gamma_{\eta,j}^{\kappa,i} b_i \hat{b}_i \rho \hat{b}_j$$

If  $b = 1$ :

$$\sum_j \gamma_{\eta,j}^{\kappa,i} b_i \hat{b}_i e_j$$

We can see in the second game that each term has a distinct  $e_j$  in them and thus will never cancel out. In the first game, we see that either there is a distinct  $\hat{b}_j$  in the term, or a  $(\hat{b}_i)^2$  when  $i = j$ . In both cases, the polynomial does not zero out.

Thus, the adversary cannot zero out one polynomial in one game without zeroing out the other game and due to Lemma 1, the adversary has no chance at distinguishing in the generic group model.  $\square$

*Proof of Thm. 3 using Lemma 5.* We can now reduce to Lem. 5 to ensure that our scheme in Fig. 3 satisfies APKCH.

To do this, we'll create three hybrids. The first hybrid is where the challenge signature ( $\sigma^b$ ) returned in the APKCH game (Fig. 2) is the same one used in the signature oracle in this game ( $\text{pk}_0$ ). The second hybrid is where the challenge signature returned is from a new random key, independent of  $\text{pk}_0$  and  $\text{pk}_A$ . The third hybrid is where the returned challenge secret key is from a randomization of the adversary's public key  $\text{pk}_A$ .

We'll first prove that the first two hybrids are indistinguishable. The reduction will generate the  $\text{sk}$  used in (Fig. 2). To first prove that this reduction secret key is indistinguishable from random in the challenge key/signature, the reduction will specially craft the public parameters from the challenger in the  $\mathbf{Exp}^{\text{PKCH-Prf}}$  experiment. The reduction uses  $P^{x_i^1}$  given from the challenger in experiment  $\mathbf{Exp}^{\text{PKCH-Prf}}$  to construct the public parameters. This is done by treating  $x_i^1$  as  $\hat{b}_i$  in the parameters, i.e., the reduction samples  $b_i, \hat{v}_r, v_i \leftarrow \mathbb{Z}_p$  and computes:  $B_i \leftarrow P^{b_i}$ ,  $\hat{B}_i \leftarrow \hat{X}_i^1$ ,  $B_{\ell+i} = (X_i^1)^{b_i}$ ,  $\hat{B}_{\ell+i} = (\hat{X}_i^1)^{b_i} \forall i \in [\ell]$ . The reduction then computes the verification bases as:  $\forall i \in [\ell], \hat{V}_i = (X_i^1)^{\hat{v}_i}, \hat{V}_{i+\ell} = P^{\hat{v}_i}$ . We can see that while the reduction does not receive any element like  $P^{1/\hat{b}_i}$ , the reduction can still create verification bases that are distributed identically to an honestly generated scheme as  $\text{dlog}_{\hat{V}_i}(\hat{V}_{i+\ell}) = 1/(x_i^1) = 1/(\hat{b}_i)$  exactly like

Setup would generate normally. The reduction then computes the rest of the verification bases as follows:  $\forall i \in [\ell], V_i = P^{v_i}, V_{i+\ell} = P^{v_i/b_i}$ . This ensures that the verification bases,  $\{V_i\}$ , are computed correctly as well.

The reduction then generates its challenge key pair,  $(\text{sk}, \text{pk})$ , and gives the public key and parameters to the adversary.

The reduction then answers signature queries with the public parameters and challenge secret key.

The adversary then submits an adversarial public key and signature. The reduction then “presigns” the lower half of the message  $\{M_i\}_{i \in [\ell]}$  with its challenge secret key (computing  $M' = \{M_i^{x_i}\}$ ) and passes this message to the challenger in the proof experiment,  $\mathbf{Exp}^{\text{PKCH-Prf}}$ . If the challenger’s bit is  $b = 0$ , the challenger exponentiates this message  $M'$  with its secret key and forms a signature. We can see that because the reduction presigned the message with the reduction’s secret key, this is distributed identically to a signature from the reduction’s secret key as we treat the challenger’s secret key as the  $\hat{b}_i$  values in the parameters. If the challenger’s bit,  $b$ , is 1, then this signature appears to be from a random secret key. To ensure the returned public key from the proof challenger verifies, we exponentiate it with  $b_i$  and concatenate the resulting  $\ell$  dimension vector to the end of the returned public key.

We then have a second reduction that operates exactly like the last, but instead of using the reduction’s secret key to presign the message to query to the challenger, it uses the adversary’s extracted secret key. Our reduction must extract the secret key from the adversary. To do this, we leverage Lemma 4 to show that if the public key verifies, then we can extract the secret key in the GGM (thus ensuring that the extractor  $\mathcal{E}_{\text{pk}}$  is efficient).

We formalize this reduction in Fig. 15. This reduction is split into the two algorithms necessary for the  $\mathbf{Exp}^{\text{PKCH-Prf}}$  experiment and takes in a generic hybrid distinguisher,  $\mathcal{A}$ , along with the input from the experiment. We use  $i = 1$  to denote the reduction that distinguishes between hybrids 1 and 2 and  $i = 2$  to denote the reduction that distinguishes between hybrids 2 and 3. We can see that  $\text{pk}'$  is a valid public key because  $(\hat{X}'_i)^{b_i} = \hat{X}'_{i+\ell}$ , which verifies when substituted into the verifying pairing equation:  $e(V_i, \hat{X}'_i) = e(V_{i+\ell}, \hat{X}'_{i+\ell})$ . Because we used the lower half of the message to pass to the PKCH-Prf challenger after exponentiating it with  $\text{sk}$ , this message has the form  $\mathbf{M}' = \{B_i^{x_i}\}_{i \in [\ell]}$ . If the PKCH-Prf’s secret bit is  $b = 0$ , then they exponentiate with their secret key and form the signature  $Z, Y, \hat{Y}$ . Because we’ve included the challenger’s public key elements in our public parameters, the challenger’s secret key is effectively equal to  $\{\hat{b}_i\}_{i \in [\ell]}$  for our reduction’s public parameters. Thus, the computation of  $Z$  will be:  $(\prod (B_i^{x_i})^{\hat{b}_i})^y = (\prod (B_{i+\ell}^{x_i})^y)$ , which is distributed identically to a signature from the reduction’s secret key. If  $b = 1$ , this  $Z$  value will instead be:  $(\prod (B_i^{x_i})^{x_{2i}})^y$ . This key effectively looks random to the hybrid distinguishing adversary, thus appearing as the second hybrid. While this signature and public key may not appear to verify as it does not depend on  $\hat{b}_i$ , we see that

if this signature verifies in the [18] scheme, it will also verify in our scheme as the construction in [18] implies that  $e(Z, \hat{Y}) = \prod_{i \in [\ell]} e(M'_i, \hat{X}_i) = 1$ . We see that our verification is instead  $e(Z, \hat{Y}) = \prod_{i \in [\ell]} e(M_i, \hat{X}'_i) = 1$  where  $\hat{X}'_i = \hat{X}_i^{x_i}$  and  $M'_i = M_i^{x_i}$  (or  $\hat{X}'_i = \hat{X}_i^{x_i, \mathcal{A}}$  and  $M'_i = M_i^{x_i, \mathcal{A}}$  if  $i = 2$ ). Thus, because of the properties of bilinear pairings, the verification of the signature holds in our scheme and thus the adversary's view is identical to the real game.

**Fig. 14.** *Hybrid<sub>i</sub>*

```

Hybridi(1λ)
1: pp ← Setup(1λ)
2: (sk, pk) = KGen(pp)
3: (pkA, σA, M, stA) ← A1(pp, pk)
4: skA ← EskA(pkA)
5: ρ ←$ Zp
6: if i = 0,
7:   pk† = ConvertPK(pk, ρ)
8:   σ = Sign(ConvertSK(sk, ρ), M)
9: if i = 1,
10:  (sk†, pk†) = KGen(pp)
11:  σ = Sign(sk†, M)
12: if i = 2,
13:  pk† = ConvertPK(pkA, ρ)
14:  σ = Sign(ConvertPK(skA, ρ), M)
15: b' ← A2(stA, pk†, σ)
16: return b'

```

Thus, because the first hybrid is identical to the APKCH game when  $b = 0$  and the third hybrid is indistinguishable from the APKCH game when  $b = 1$ , we've proven that these two cases are indistinguishable.  $\square$

## B.2 Proofs of the extendable construction in Fig. 10

To show that any property of a non-extended scheme still holds for an extended scheme, observe that **ExtendSetup** will exponentiate each element in the public parameters with random scalars and that the resulting public parameters have the same distribution as the extended ones. Thus, our adversary cannot distinguish between these two cases (where **ExtendSetup** is called and when **Setup** is called). Thus, a reduction to any game on our original construction can be reduced by our extended construction by simply a reduction that simply ignores the **pp'** given by an adversary and instead using the **pp** from the challenger. This works because the **ExtendSetup** scheme effectively chooses a random set of valid public parameters and thus the one from the challenger is just as likely as any other set of public parameters. This works because the structure of public parameters can be verified by performing the following checks:  $\forall i \in [\ell], e(V_i, \hat{B}_i) = e(V_{\ell+i}, \hat{B}_{\ell+i}), e(B_i, \hat{V}_i) = e(B_{\ell+i}, \hat{V}_{\ell+i})$  and



**Fig. 15.** APKCH proof reduction

$\mathcal{R}_1^{i,\mathcal{A}}(\mathbf{B}^1, \mathbf{B}^u, \hat{\mathbf{B}}^1, \hat{\mathbf{E}}^b) \rightarrow (\text{st}, \mathbf{M}')$   
1:  $\mathbf{d} = \{\hat{d}_i\}_{i \in [\ell]} \leftarrow_{\$} \mathbb{Z}_p$   
2:  $\mathbf{r} = \{\hat{v}_i\}_{i \in [\ell]} \leftarrow_{\$} \mathbb{Z}_p$   
3:  $\mathbf{o} = \{v_i\}_{i \in [\ell]} \leftarrow_{\$} \mathbb{Z}_p$   
4:  $\mathbf{pp} = \{D_i, B_i, B_{\ell+i}, \hat{B}_i, \hat{B}_{\ell+i}, V_i, V_{i+\ell}, \hat{V}_i, \hat{V}_{i+\ell}\}_{i \in [\ell]}$  where  $B_i, B_{\ell+i}, \hat{B}_i$  are drawn from the reduction's input, and  $D_i = P^{\hat{d}_i}$ ,  $\hat{B}_{\ell+i} = \hat{B}_i^{\hat{d}_i}$ ,  $V_i = P^{v_i \hat{d}_i}$ ,  $V_{i+\ell} = P^{v_i \hat{d}_i}$ ,  $\hat{V}_i = \hat{B}_i^{v_i}$ , and  $\hat{V}_{i+\ell} = P^{v_i}$   
5:  $(\text{sk}, \text{pk}) = \text{KGen}(\mathbf{pp})$  such that  $\text{sk} = \{x_i\}_{i \in [\ell]}$  and  $\text{pk} = \{\hat{B}_i^{x_i}\}_{i \in [\ell]} \parallel \{\hat{B}_{\ell+i}^{x_i}\}_{i \in [\ell]}$   
6:  $(\text{pk}^{\mathcal{A}}, \sigma^{\mathcal{A}}, \mathbf{M}, \text{st}_{\mathcal{A}}) \leftarrow \mathcal{A}_1(\mathbf{pp}, \text{pk})$   
7:  $\text{sk}^{\mathcal{A}} \leftarrow \mathcal{E}_{\text{sk}}^{\mathcal{A}}(\text{pk}_{\mathcal{A}})$   
8: **if**  $i = 1$ ,  $\mathbf{M}' = \{M_i^{x_i}\}_{i \in [\ell]}$   
9: **if**  $i = 2$ ,  $\mathbf{M}' = \{M_i^{x_i^{\mathcal{A}}}\}_{i \in [\ell]}$   
10: **return**  $(\text{st}, \mathbf{M}')$

---

$\mathcal{R}_2^{i,\mathcal{A}}(\text{st}, \sigma) \rightarrow b$   
11: **if**  $i = 1$ ,  $\text{pk}' = \{(\hat{E}_i^b)^{x_i}\}_{i \in [\ell]}$   
12: **if**  $i = 2$ ,  $\text{pk}' = \{(\hat{E}_i^b)^{x_i^{\mathcal{A}}}\}_{i \in [\ell]}$   
13:  $\text{pk}^\dagger = \text{pk}' \parallel (\text{pk}')^{\hat{d}_i}$   
14:  $b' \leftarrow \mathcal{A}_2(\text{st}_{\mathcal{A}}, \text{pk}^\dagger, \sigma)$   
15: **return**  $b'$

$e(B_i, \hat{B}_i) = e(B_{\ell+i}, \hat{P})$ . These checks ensure that the public parameters are structured correctly.

When calling `FinalizeSetup`, for any of the games, we can create a reduction that simply uses  $td$  to exponentiate elements from the challenger with  $b_i$  and  $\hat{b}_i$  or exponentiate elements with  $1/b_i$  and  $1/\hat{b}_i$  from the adversary. This poses a non-trivial challenge as the adversary submits a signature which already has these  $\hat{b}_i$  or  $b_i$  values in it which our reduction cannot do a pair-wise exponentiation to fix. But fortunately, if we simply update the verification bases of the public key with  $1/\hat{b}_i$  or  $1/b_i$ , we can see that it verified with the unfinalized public parameters and the rest of the public key, signature, and message are distributed correctly. This makes our reduction valid for the unforgeability game, but in the APKCH game, the challenger will then return a randomized public key and signature. We can see that after receiving the public key from the APKCH challenger, this reduction can simply exponentiate it with  $b_i$  and  $\hat{b}_i$  to make it look right for the adversary and thus, our reduction works for any of the games.

Thus, because we can replace the output of `ExtendSetup` with fresh parameters, and also, we can translate messages and public keys between the schemes after using `FinalizeSetup` to win the security games with our reduction, our proofs hold when using `Setupext` instead of `Setup`  $\square$

### B.3 DAC proof

*Proof of Thm. 7 (Unforgeability).* From the game in Def. 12, we know that if the adversary wins with non-negligible probability, then one of three cases occurred, either (1)  $\text{sk}_0 \neq \text{sk}_{\text{root}}$ , (2)  $\forall i \in [L'], (\text{sk}_i, i) \notin \text{DEL}_{\mathcal{A}}$ , or (3)  $\exists i \in [L']$ , s.t.  $\text{sk}_i \in \text{SK}_{DL}$ . We'll start with the first case, where  $\text{sk}_0 \neq \text{sk}_{\text{root}}$ . We can see that our construction checks to ensure that the first public key is the root's key so this cannot occur. In the second case, we have that  $\forall i \in [L'], (\text{sk}_i, i) \notin \text{DEL}_{\mathcal{A}}$ . In this case, we have that no key in the credential chain was actually delegated to that adversary. In this case, we can either reduce to the unforgeability of the scheme, or recover a secret key of an honest user through the adversary's proof of knowledge of the end user. In the third case, we have that  $\exists i \in [L']$ , s.t.  $\text{sk}_i \in \text{SK}_{DL}$ . Because the TRA reveals the secret keys of any blacklisted user, and recognizing public keys succeeds with probability 1, we see that this cannot happen. Thus, our DAC scheme is unforgeable.

*Proof of Thm. 8 (Anonymity).* We can see in the anonymity game that the only difference is between  $b = 0$  and  $b = 1$  is when the adversary receives the anonymity challenge. We will construct hybrids that replace the non-root public keys with random public keys starting from the lowest level and working up. We can create hybrids that replace public keys and signatures from the bottom up to prove that either credential that the challenger picks is indistinguishable from a random chain.

**Definition 21** (*Hybrid<sub>b,j</sub>*).

- 1:  $\text{pp} \leftarrow \text{Setup}(1^\lambda, 1^L)$
- 2:  $(\text{tsk}, \text{tpk}) \leftarrow \text{Setup}(\text{pp})$
- 3:  $(st, \text{pk}_{\text{root}}, \text{sk}_0, \text{cred}_0, \text{sk}_1, \text{cred}_1, L') \leftarrow \mathcal{A}_0^{\mathcal{O}^{\text{RegisterUser}(\cdot)}, \mathcal{O}^{\text{RevokeUser}(\cdot)}}(\text{pp})$
- 4:  $\forall i \in \{0, 1\}$ , if  $(\text{Prove}(\text{pp}, \text{sk}_i, \text{cred}_i, L') \leftrightarrow \text{Verify}(\text{pk}_{\text{root}}, L', DL)) \neq 1$ , **return**  $\perp$
- 5:  $\{(\text{pk}_i, \sigma_i, \text{tok}_i)\} \leftarrow \text{cred}_b$
- 6:  $\text{sk}_{j-1} = \mathcal{E}_{\text{pk}}(\text{pk}_{j-1})$
- 7:  $\forall i \in [L' - j]$ ,
- 8:  $(\text{sk}_{j+i-1}, \text{pk}_{j+i-1}^*) \leftarrow \mathcal{PK}(\text{pp})$ ,
- 9:  $\text{tok}_{j+i-1}^* \leftarrow \text{RegisterUser}(\text{pp}, \text{tsk}, \text{pk}_{j+i-1}^*)$
- 10:  $\sigma_{j+i-1}^* \leftarrow \text{Sign}(\text{pp}, \text{sk}_{j+i-2}, \text{pk}_{j+i-1}^*)$
- 11:  $\text{cred}^* = \{(\text{pk}_i, \sigma_i, \text{tok}_i)\}_{i \in [j]} \parallel \{(\text{pk}_{j+i-1}^*, \sigma_{j+i-1}^*, \text{tok}_{j+i-1}^*)\}$
- 12:  $\text{Prove}(\text{pp}, \text{sk}_{L'}^*, \text{cred}^*, L') \leftrightarrow \mathcal{A}_1(st) \rightarrow b'$

To create these hybrids, the challenger in these hybrids will extract the secret keys from the public keys in the chains. We construct our reduction from an adversary that can distinguish hybrid  $j$  and  $j + 1$  to APKCH in Def. 22.

After we've proven that *Hybrid<sub>0,0</sub>* is indistinguishable from *Hybrid<sub>0,L</sub>*, we can again create hybrids that instead use the public keys in  $\text{cred}_1$  in the anonymity challenge, thus proving that *Hybrid<sub>1,0</sub>* is indistinguishable from *Hybrid<sub>1,L</sub>*. We

then observe that  $Hybrid_{0,L}$  is identical to  $Hybrid_{1,L}$  as all keys have been replaced with random ones. Thus, at either end of our hybrids, we'll have either case of the anonymity challenge and thus prove that they are indistinguishable.

**Definition 22 (Reduction of distinguishing  $Hybrid_{b,j}$  from  $Hybrid_{b,j+1}$  to APKCH).**

- 
- $\mathcal{R}_1^{\text{O}^{\text{Sign}}(\text{sk}, \cdot)}(\text{pk}, \text{pp}) \rightarrow (\text{pk}^*, \sigma^*, M^*, \text{st})$
- 1: If  $j = L - 1$ , the reduction interacts with  $\text{Setup}^{\text{ext}}$  in the APKCH game by first supplying  $\perp$  (as  $\text{pp}_0$  on line Line 1 of  $\text{Setup}^{\text{ext}}$  in Fig. 7) to make the challenger generate the parameters using  $\text{Setup}$ . The reduction receives the public parameters from the challenger which they label  $\text{pp}'_L$  (these parameters are generated by the challenger on line Line 3 of Fig. 7 in  $\text{Setup}^{\text{ext}}$  and in that function they are labeled  $\text{pp}_1$ ). The reduction then generates  $(\text{pp}'_{L-1}, \text{td}'_{L-1}) \leftarrow \text{ExtendSetup}(\text{pp}'_L)$  and returns  $\text{td}'_{L-1}$  to the challenger (on line Line 6 of  $\text{Setup}^{\text{ext}}$  as  $\text{td}'$ ) which allows the challenger to finalize their setup parameters. The challenger then returns the finalized parameters, which the reduction labels  $\text{pp}_L$ . The reduction then generates  $\forall i \in [L-1], (\text{pp}'_{i-1}, \text{td}'_{i-1}) \leftarrow \text{ExtendSetup}(\text{pp}'_i)$  and  $\forall i \in [L-1] \setminus \{L\}, (\text{pp}_i, \text{td}_i) \leftarrow \text{FinalizeSetup}(\text{pp}'_i, \text{td}'_i, \text{td}'_{i-1})$ . The reduction then computes  $\text{pp}_{\text{DAC}} = \{\text{pp}_i\}_{i \in [L]}$ .
  - 2: If  $j \neq L - 1$ , the reduction generates  $(\text{pp}'_L, \text{td}'_L) = \text{Setup}(1^\lambda, 1^{\ell=2})$  and then calls  $\forall i \in [L-j], (\text{pp}'_{j+i}, \text{td}'_{j+i}) \leftarrow \text{ExtendSetup}(\text{pp}'_{j+i+1})$ . The reduction then interacts with  $\text{Setup}^{\text{ext}}$  in the APKCH game by supplying  $\text{pp}'_{j+1}$  as the parameters labeled  $\text{pp}_0$  in  $\text{Setup}^{\text{ext}}$  in Fig. 7. The reduction labels these parameters as  $\text{pp}'_j$ . The challenger then extends this computing  $(\text{pp}'_{j-1}, \text{td}'_{j-1}) = \text{ExtendSetup}(\text{pp}'_j)$  and returns  $\text{td}'_{j-1}$  to the challenger (labeled  $\text{td}'$  in Fig. 7). The challenger then returns public parameters which the reduction labels as  $\text{pp}_j$ . The reduction then finishes initializing the scheme by computing  $\forall i \in [j-2], (\text{pp}'_i, \text{td}'_i) \leftarrow \text{ExtendSetup}(\text{pp}'_{i+1})$ . The reduction then calls  $\forall i \in [j-1], (\text{pp}_i, \text{td}_i) \leftarrow \text{FinalizeSetup}(\text{pp}'_i, \text{td}'_i, \text{td}'_{i-1})$  to finish the parameters,  $\text{pp}_{\text{DAC}} = \{\text{pp}_i\}_{i \in [L]}$ .
  - 3:  $(\text{tsk}, \text{tpk}) \leftarrow \text{TKeyGen}(\text{pp}_{\text{DAC}})$
  - 4:  $(\text{st}, \text{pk}_{\text{root}}, \text{sk}_0, \text{cred}_0, \text{sk}_1, \text{cred}_1, L') \leftarrow \mathcal{A}_0^{\text{O}^{\text{RegisterUser}}(\cdot), \text{O}^{\text{RevokeUser}}(\cdot)}(\text{pp}_{\text{DAC}})$
  - 5:  $\forall i \in \{0, 1\}$ , if  $(\text{Prove}(\text{pp}, \text{sk}_i, \text{cred}_i, L') \leftrightarrow \text{Verify}(\text{pk}_{\text{root}}, L', \text{tpk})) \neq (1, *)$ , return  $\perp$
  - 6:  $\{(\text{pk}_i, \sigma_i, \text{tok}_i)\} \leftarrow \text{cred}_b$   
 In the next line, the reduction returns their public key  $\text{pk}_j$ , signature  $\sigma_j$ , and message,  $\text{pk}_{j+1}$  to the APKCH challenger.
  - 7: return  $(\text{pk}_j, \sigma_j, \text{pk}_{j+1}, \text{st} = (\text{tsk}, \text{pp}_{\text{DAC}}))$
- 

$\mathcal{R}_2^{\text{O}^{\text{Sign}}(\text{sk}, \cdot)}(\text{pp}, \text{st}, \text{pk}^{b^*}, \sigma^{b^*}) \rightarrow b'$

Here, the reduction has received  $\text{pk}^{b^*}, \sigma^{b^*}$  from the APKCH challenger which is either a randomization of  $\text{pk}_j, \sigma_j$  or a new signature form an unrelated key.

- 8:  $\text{tok}^* \leftarrow \text{RegisterUser}(\text{pp}, \text{tsk}, \text{pk}^{b^*})$
- 9:  $\text{cred}^* = \{(\text{pk}_i, \sigma_i, \text{tok}_i)\}_{i \in [j]}$

$$\begin{aligned}
& \|\{(\mathbf{pk}^{b^*}, \sigma^{b^*}, tok^{b^*})\} \\
& \|\{(\mathbf{pk}_{j+i}, \sigma_{j+i}, tok_{j+i})\}_{i \in [L'-j]} \\
10: \text{ Prove}(\text{pp}_{\text{DAC}}, \mathbf{sk}_{L'}^*, \text{cred}^*, L') \leftrightarrow \mathcal{A}_1(st) \rightarrow b'
\end{aligned}$$

*Analysis.* We can see that if the APKCH challenger's bit,  $b$ , is 0, this appears identical to  $\text{Hybrid}_{b,j}$ . This is because the challenger returns a  $\mathbf{pk}_b$  in the same equivalence class as  $\mathbf{pk}_j$ . Because our scheme achieves perfect origin-hiding, this means that this appears exactly as if a real user had randomized it. If  $b$  is 1, then  $\mathbf{pk}_b$  is a random public key, and we appear exactly as  $\text{Hybrid}_{b,j+1}$ . This is because the  $\mathbf{pk}_b$  returned is now a random public key.

For issuing, we can see that the proof is similar, except that the issuer produces a credential chain on a message from the adversary as long as that message verifies.