Distributed Asynchronous Remote Key Generation

Mark Manulis and Hugo Nartz

PACY Lab @ RI CODE
Universität der Bundeswehr München, Munich, Germany
mark@manulis.eu, hugo.nartz@unibw.de

Abstract. Asynchronous Remote Key Generation (ARKG) is a primitive introduced by Frymann et al. at ACM CCS 2020. It enables a sender to generate a new public key pk' for a receiver ensuring only it can, at a later time, compute the corresponding private key sk'. These key pairs are indistinguishable from freshly generated ones and can be used in various public-key cryptosystems such as digital signatures and public-key encryption. ARKG has been explored for applications in WebAuthn credential backup and delegation, as well as for enhancing receiver privacy via stealth addresses.

In this paper, we introduce distributed ARKG (dARKG) aiming to provide similar security properties in a distributed setting. Here, a sender generates pk' for a group of n receivers and the corresponding sk' can only be computed by any sub-group of size $t \leq n$. This introduces threshold-based access protection for sk', enabling for instance a set of proxies to jointly access a WebAuthn account or claim blockchain funds.

We construct dARKG using one-round publicly verifiable asymmetric key agreement, called 1PVAKA, a new primitive formalized in this work. Unlike traditional distributed key generation protocols where users interact with one another, 1PVAKA is asynchronous and allows a third party to verify and generate a public key from users' outputs.

We discuss 1PVAKA and dARKG instantiations tailored for use with bilinear groups and demonstrate practicality with implementation and performance analysis for the BLS12-381 curve.

Keywords: Asynchronous Remote Key Generation, Distributed Key Generation, Unlinkability

1 Introduction

Background and Motivation. Asynchronous Remote Key Generation introduced by Frymann et al. [FGK$^+$] serves as a cryptographic encapsulation mechanism for generating asymmetric key pairs between a sender and a receiver. In ARKG, the sender utilizes a receiver's long-term public key pk to derive a fresh public key pk' along with credentials cred. These credentials enable the receiver to later derive the corresponding private key sk' from its own long-term secret key sk. Security-wise, SK-security and PK-unlinkability ensure that only the intended receiver can access sk' and that derived key pairs $(\mathsf{sk}', \mathsf{pk}')$ are indistinguishable from freshly generated ones. Consequently, derived keys can be utilized in various cryptographic applications such as digital signatures or public key encryption, provided they are structurally compatible.

The initial instantiation of ARKG aimed to produce Diffie-Hellman-like key pairs. Subsequent developments addressed other types of keys. The generalized construction outlined in [FGMN23] focuses on pairing-based cryptosystems while the one in [FGM], relying on split-KEMs [BFG$^+$20], supports lattice-based cryptosystems like Dilithium [DLL$^+$18] and Kyber [BDK$^+$18]. Recent papers focus primarily on the original application of ARKG in the backup of WebAuthn credentials [BCH$^+$19]. Wilson [SW] leverages KEMs with key blinding properties derived from lattices to achieve a stronger form of SK-security. Brendel et al. [BCF23] also constructed a post-quantum ARKG scheme based on KEMs and digital signatures, albeit under modified security definitions.

Applications of ARKG. Initially devised to address the backup of hardware authenticators like Yubikeys within the WebAuthn/FIDO2[1] framework, ARKG has found versatile applications. The primitive was used to enable delegation of WebAuthn accounts and the construction of a new class of proxy signatures with unlinkable warrants [FGM22]. This allows an account holder to delegate access rights for multiple accounts to a proxy. The blinding property providing unlinkability in ARKG is also similar to the one used for public-key malleability in mercurial signatures which found application in anonymous credential delegation [CL19,MBG$^+$23].

Beyond credentials delegation, the same techniques used in ARKG contribute to the realization of stealth addresses and signatures in blockchain transactions [FGMN23,FGM], enhancing receiver privacy while allowing fund claiming. In fact, the original scheme from [FGK$^+$] for DL-based key pairs uses techniques that power the stealth addresses on the Monero blockchain [Yu]. The recent post-quantum stealth signature scheme [PTDH] also adopts similar algorithms for the generation of public and private keys.

Motivation for a Distributed Scheme. A derived public key pk' can be used immediately for application tasks, i.e., before the corresponding sk' is derived by the receiver. The reconstruction of sk' is not possible is not possible if the receiver is unavailable. Moreover, compromising the receiver's static private key sk (which is excluded in the ARKG security model) yields sk'.

From this perspective, the restriction to a sole receiver exhibits single points of failure and attacks which may negatively impact the aforementioned applications. For example, if a WebAuthn backup authenticator is compromised, the attacker will gain unconditional access to all accounts on which backup keys were registered. Similarly, in case of stealth addresses, the loss or compromise of a transaction receiver's static private key would prevent them from claiming the funds or allow the attacker to claim them instead.

[1] https://github.com/Yubico/libfido2

We argue that a *distributed* ARKG (dARKG) functionality where the derived secret key sk' can only be reconstructed by a (threshold) set of receivers would help mitigate the aforementioned risks. The distributed setup may also enable additional use cases. Owners of WebAuthn or unlinkable blockchain accounts could delegate access to trusted parties to reconstruct the required ARKG private keys sk' when needed. This power of attorney situation might apply in cases ranging from unavailability due to travel to death of the owner. Additionaly, threshold-based reconstruction of sk' would enable delegation of such accounts with access control policies based on the number of users, which is of particular interest in enterprise scenarios.

1.1 Challenges and Design Rationale

The dARKG primitive must allow a *delegator* to use the long-term public keys of n *proxies* in order to derive a fresh public key pk' and associated credentials. At a later stage, a subset of proxies should be able to reconstruct the corresponding private key sk' (or shares of it) from their long-term keys and credentials. It is desirable for dARKG to be compatible with the already existing functionalities and security requirements [FGK+] of ARKG. Of particular importance is also the need to preserve asynchrony during the generation of pk'.

A sensible approach to build the primitive is to use existing (Non-Interactive) Distributed Key Generation ((NI-)DKG) protocols, e.g. [Gro,KGS]. The public key resulting from such a scheme could be used as the input static public key for the original ARKG. Clearly, interactive DKG protocols are less suitable as they assume online presence of all players and hence stand in conflict with the asynchrony requirement. In contrast, NI-DKG, e.g. [Gro], players only broadcast one message to each other. While this provides asynchrony, in dARKG proxies may not even have any means to communicate with each other at the time the static public key is created. In fact, the dARKG applications previously mentioned assume that each proxy interacts only with the delegator and that these interactions are asynchronous. From this perspective, we need a somewhat different flavour of (NI-)DKG where the delegator, as a dedicated party, asynchronously receives messages from the proxies and computes the resulting static public key pk. Importantly, the delegator must also be able to validate payloads from the proxies to ensure robustness: if pk' was successfully derived, the later computation of sk' by t honest players must also succeed.

Key Agreement for dARKG. In our paper we thus first propose a new one-round publicly verifiable asymmetric key agreement, denoted 1PVAKA. Each user generates a share sh and a "dealing" d for a group of users identified using their long-term public keys $\mathsf{ek}_1, \ldots, \mathsf{ek}_n$. A third party can verify individual dealings and aggregate them into a public static key pk and new dealings $\mathsf{f}_1, \ldots, \mathsf{f}_l$ for the l qualified users. Qualified users can, using their long-term secret keys dk_i and those new dealings, reconstruct the corresponding static key sk.

With dARKG in mind, the main challenge in building 1PVAKA is for the third party (the delegator) to combine users' (proxies) outputs into a static public key, while ensuring future recoverability of the static private key. That is, misbehaviour of asynchronous users must be identifiable during a 1PVAKA key generation session before aggregating a public key pk for the group.

For that purpose, the following natural approach is insufficient. Suppose each participant encrypts its own ephemeral secret key using (multi-recipient) public-key encryption for each other participant and broadcasts these ciphertexts along with the corresponding ephemeral public key (forming dealing $(\mathsf{d}_i)_{i\in[n]}$). A third party then combines the ephemeral public keys into a static group key pk and use the dealings as public information. In this configuration, a single malicious user could send incorrect ciphertexts (unerelated to the public key in their dealing) leading later to unrecoverability of the private key sk. This is unacceptable as the public key pk may be used for many cryptographic operations before sk is required. In our construction of 1PVAKA we therefore use Verifiable Encryption (VE) [CD] to ensure public verifiability of ciphertexts encrypting ephemeral secret keys with respect to their counterpart public keys.

The asymmetric group key agreement presented in [WMS$^+$] does not support *public* verifiability as derived decryption keys are required to verify the derived group public key. Nevertheless, our 1PVAKA construction uses a similar technique related to the homomorphic structure of public/private keys. We also use additively homomorphic encryption (similarly to [Gro]) to compress the payload sent by the delegator to a proxy from $\mathcal{O}(n)$ to $\mathcal{O}(1)$ ciphertexts.

1.2 Our contributions

In this work, we introduce a framework for dARKG, building upon the foundational principles of ARKG outlined in [FGK$^+$]. Essentially, the original scheme can be seen as a specialized instance within our distributed framework.

We establish the syntax of dARKG along with key security properties, adapting the original definitions of SK-security and PK-unlinkability to account for the challenges inherent to the distributed approach. In doing so, we enhance PK-unlinkability to a more robust form, as seen in [SW,BCF23]. Our overarching dARKG construction uses 1PVAKA alongside conventional building blocks such as threshold secret sharing, verifiable public key encryption and KEM.

We treat 1PVAKA as an independent primitive at the frontier between asymmetric group key agreement [WMS$^+$] and non-interactive DKG [Gro]. To produce generic transformations compatible with most asymmetric cryptosystems, we use the blinding Asymmetric Key Generation (AKG) abstraction from [FGMN23]. This extends the applicability of our dARKG scheme to diverse key pair types, including those suitable for pairing-based cryptosystems.

The security properties of both our general 1PVAKA and dARKG constructions are proven in the standard model. We present an instantiation of 1PVAKA using single-message multi-receiver VE for use with the BLS12-381 curve. This curve can be used for authentication in WebAuthn and major blockchains like Bitcoin and Ethereum, as well as for BLS-3 signatures increasingly employed in distributed scenarios. Finally, we provide a Jupyter notebook with detailed benchmarks for an implementation of our instantiation of 1PVAKA. The results are satisfying and highlight the communication costs vs. complexity costs trade-off we aimed for.

1.3 Outline of the paper

We start in Section 2 by recalling the building blocks used in our 1PVAKA and dARKG constructions, including asymmetric key generation and verifiable encryption. We introduce 1PVAKA in Section 3 along with definitions and a construction. Section 4 presents the framework, syntax, security definitions and our general construction for dARKG. In Section 5 we discuss an instantiation for the pairing friendly BLS12-381 elliptic curve and our experimental results.

2 Preliminaries

We define security of cryptographic primitives through experiments involving Probabilistic Polynomial Time (PPT) adversaries denoted by letters $\mathcal{A}$ or $\mathcal{B}$. Notation $\mathcal{A}^{\mathcal{O}}$ specifies that $\mathcal{A}$ has access to oracle $\mathcal{O}$. Security parameters are denoted by λ. By $\mathsf{negl}(\lambda)$ we denotes any function f such that $f(\lambda) = \lambda^{-\omega(1)}$.

Let $n \in \mathbb{N}$, we denote by $[n]$ the set $\{1, \ldots, n\}$. For $I \subset \mathbb{N}$, we denote by $[\mathsf{x}]_I$ the set $\{\mathsf{x}_i \mid i \in I\}$ when elements x_i are clearly defined from context. For integer $n \in \mathbb{N}$, we abbreviate $[\mathsf{x}]_n := [\mathsf{x}]_{[n]}$.

The syntax and security definitions of the original ARKG scheme are recalled in Appendix A.

2.1 Key Encapsulation

Let $\mathcal{K}$ be an arbitrary key space. A Key Encapsulation Mechanism (KEM) scheme (for key space $\mathcal{K}$) is defined by the following three PPT algorithms:

$\mathtt{KGen}(1^\lambda) \to (\mathsf{dk}, \mathsf{ek})$: The key generation algorithm takes as input the unary representation of the security parameter λ. It outputs an encapsulation key ek and a decapsulation key dk.

$\mathtt{Encaps}(\mathsf{ek}) \to (K, \mathsf{ct})$: The encapsulation algorithm takes as input an encapsulation key ek. It outputs a key $K \in \mathcal{K}$ and a ciphertext ct.

$\mathtt{Decaps}(\mathsf{dk}, \mathsf{ct}) \to K$ or $\perp$: The deterministic decapsulation algorithm takes as input a decapsulation key dk and a ciphertext ct. It outputs a key $K \in \mathcal{K}$ or $\perp$ indicating that the ciphertext is invalid.

Definition 1 (Correctness). *A KEM is correct if for all $\lambda \in \mathbb{N}$,*

$$\Pr\left[\begin{array}{c}(\mathsf{dk}, \mathsf{ek}) \leftarrow \mathtt{KGen}(1^\lambda),\ (K, \mathsf{ct}) \leftarrow \mathtt{Encaps}(\mathsf{ek}),\\ K' \leftarrow \mathtt{Decaps}(\mathsf{dk}, \mathsf{ct}) : K \neq K'\end{array}\right] = \mathsf{negl}(\lambda)$$

where the probability is taken over the random coins of $\mathtt{KGen}$ *and* $\mathtt{Encaps}$*.*

```
Experiment $\mathsf{Exp}^{\mathsf{ANO\text{-}CCA}}_{\mathsf{KEM},\mathcal{A}}(\lambda)$:
$(\mathsf{dk}_b, \mathsf{ek}_b) \leftarrow \mathtt{KGen}(1^\lambda)\ b \in \{0,1\}$
$b \leftarrow\!\!\$\ \{0,1\}$
$(K, \mathsf{ct}) \leftarrow \mathtt{Encaps}(\mathsf{ek}_b)$
$b' \leftarrow \mathcal{A}^{\mathcal{O}_{\mathsf{Decaps}}(\mathsf{dk},\cdot)}(\mathsf{ek}_0, \mathsf{ek}_1, K, \mathsf{ct})$
return $b \overset{?}{=} b'$
```

```
Oracle $\mathcal{O}_{\mathsf{Decaps}}(b, \mathsf{ct}^\star)$:
if $\mathsf{ct}^\star \overset{?}{=} \mathsf{ct}$
    then return $\perp$
else
    return $K := \mathtt{Decaps}(\mathsf{dk}_b, \mathsf{ct}^\star)$
```

Fig. 1: ANO-CCA game for KEM.

We recall the standard notion of anonymity under chosen ciphertext attacks (ANO-CCA) for KEM.

Definition 2 (ANO-CCA Security). *A KEM is said to be* ANO-CCA *secure if for all λ and PPT adversary $\mathcal{A}$, it holds that*

$$\mathsf{Adv}^{\mathsf{ANO\text{-}CCA}}_{\mathsf{KEM},\mathcal{A}}(\lambda) := \left|\Pr\left[\mathsf{Exp}^{\mathsf{ANO\text{-}CCA}}_{\mathsf{KEM},\mathcal{A}}(\lambda) = 1\right] - \frac{1}{2}\right| = \mathsf{negl}(\lambda).$$

where experiment $\mathsf{Exp}^{\mathsf{ANO\text{-}CCA}}_{\mathsf{KEM},\mathcal{A}}$ is defined in Figure 1 and the probability is taken over the random coins of the algorithms.

2.2 Asymmetric Key Generation (AKG)

Asymmetric Key Generation (AKG) is an abstraction introduced in [FGMN23] providing handles for manipulating asymmetric keys. We add to the AKG syntax the convenient key blinding algorithms defined in [SW]. This framework allows us to provide a general construction of dARKG compatible with generic asymmetric cryptosystems. Let $(\mathbb{S}, +)$ and $(\mathbb{P}, \cdot)$ be a secret and public key space respectively with abelian group structures. An AKG scheme for key spaces $\mathbb{S}$ and $\mathbb{P}$ is defined by the following PPT algorithms:

$\mathtt{Setup}(1^\lambda)$: The setup algorithm takes as input a security parameter λ in unary representation. It outputs public parameters of the scheme pp.

$\mathtt{KGen}(\mathsf{pp})$: The key generation algorithm takes as input public parameters pp. It outputs a secret-public key pair $(\mathsf{sk}, \mathsf{pk}) \in \mathbb{S} \times \mathbb{P}$.

$\mathtt{Check}(\mathsf{sk}, \mathsf{pk})$: The check algorithm takes as input public parameters pp, a secret key pk and a public key pk. It returns 1 if $(\mathsf{sk}, \mathsf{pk})$ is in the image of $\mathtt{KGen}$ and 0 otherwise.

If the following two algorithms are available, the scheme is said to be *blinding*:

$\mathtt{BlindPK}(\mathsf{pk}, \tau)$: The public key blinding algorithm takes as input public parameters pp, a public key pk and randomness $\tau \in \{0,1\}^\lambda$. It returns a blinded public key pk_τ.

$\texttt{BlindSK}(\mathsf{sk},\tau)$: The secret key blinding algorithm takes as input public parameters pp, a secret key sk and randomness $\tau \in \{0,1\}^\lambda$. It returns a blinded secret key sk_τ.

Definition 3 (Correctness). *An AKG scheme is correct if it provides valid key pairs compatible under the group laws: for all $\lambda \in \mathbb{N}$, $\mathsf{pp} \leftarrow \texttt{Setup}(1^\lambda)$, $(\mathsf{sk},\mathsf{pk}) \leftarrow \texttt{KGen}(\mathsf{pp})$ and $(\mathsf{sk}',\mathsf{pk}') \leftarrow \texttt{KGen}(\mathsf{pp})$, it holds that*

$$\Pr\big[\texttt{Check}(\mathsf{sk},\mathsf{pk}) \neq 1\big] = \mathsf{negl}(\lambda)$$
$$\Pr\big[\texttt{Check}(\mathsf{sk}+\mathsf{sk}',\mathsf{pk}\cdot\mathsf{pk}') \neq 1\big] = \mathsf{negl}(\lambda).$$

A blinding AKG is correct if additionaly blinded key pairs remain valid,

$$\Pr\left[\begin{matrix}\tau \leftarrow\!\!\$\, \{0,1\}^\lambda\,,\ \mathsf{pk}_\tau \leftarrow \texttt{BlindPK}(\mathsf{pk},\tau),\\ \mathsf{sk}_\tau \leftarrow \texttt{BlindPK}(\mathsf{sk},\tau):\ \texttt{Check}(\mathsf{sk}_\tau,\mathsf{pk}_\tau) \neq 1\end{matrix}\right] = \mathsf{negl}(\lambda).$$

The probabilities are taken over τ and the random coins of `Setup` *and* `KGen`*.*

An AKG scheme commonly used in DL-based cryptosystems takes for public parameters a cyclic group $\mathbb{G}$ of given order $q = q(\lambda)$ and a generator g. In this setting $\mathbb{S} = \mathbb{Z}_q$, $\mathbb{P} = \mathbb{G}$ and the key generation algorithm outputs key pairs $(x, g^x) \in \mathbb{Z}_q \times \mathbb{G}$. Blinding can be achieved using a function $\mathcal{H} : \{0,1\}^\lambda \to \mathbb{Z}_q$ by defining $\mathsf{sk}' := \mathcal{H}(\tau)$ and $\mathsf{pk}_\tau := \mathsf{pk}\cdot g^{\mathsf{sk}'}$, $\mathsf{sk}_\tau := \mathsf{sk}+\mathsf{sk}'$. This is the method employed in [FGK+] to achieve PK-unlinkability in the DL instance of ARKG.

Definition 4 (Security). *We say that the key-secrecy property holds for an AKG scheme or that it is secure if for all $\lambda \in \mathbb{N}$ and PPT adversary $\mathcal{A}$,*

$$\mathsf{Adv}^{\mathsf{secrecy}}_{\mathsf{AKG},\mathcal{A}}(\lambda) := \Pr\left[\begin{matrix}\mathsf{pp} \leftarrow \texttt{Setup}(1^\lambda),\ (\mathsf{sk},\mathsf{pk}) \leftarrow \texttt{KGen}(\mathsf{pp}),\\ \mathsf{sk}^\star \leftarrow \mathcal{A}(\mathsf{pp},\mathsf{pk}) : \texttt{Check}(\mathsf{sk}^\star,\mathsf{pk}) = 1\end{matrix}\right] = \mathsf{negl}(\lambda).$$

A blinding AKG scheme is secure if furthermore blinded public keys are indistinguishable from freshly generated ones, i.e.,

$$\mathsf{Adv}^{\mathsf{blinding}}_{\mathsf{AKG},\mathcal{A}}(\lambda) := \Pr\left[\begin{matrix}\mathsf{pp} \leftarrow \texttt{Setup}(1^\lambda),\ (\_\,,\mathsf{pk}_0) \leftarrow \texttt{KGen}(\mathsf{pp}),\\ (\_\,,\mathsf{pk}_\star) \leftarrow \texttt{KGen}(\mathsf{pp}),\ \tau \leftarrow\!\!\$\, \{0,1\}^\lambda\,,\\ \mathsf{pk}_1 \leftarrow \texttt{BlindPK}(\mathsf{pk}_\star,\tau),\\ b \leftarrow\!\!\$\, \{0,1\}\,,\ b' \leftarrow \mathcal{A}(\mathsf{pk}_b):\ b = b'\end{matrix}\right] = \mathsf{negl}(\lambda).$$

The probabilities are taken over τ and the random coins of the algorithms.

2.3 Verifiable Encryption (VE)

After encrypting a witness w for a statement x in a ciphertext ct under public key ek, verifiable encryption allows a third party to verify that ct indeed encrypts a witness for x in zero-knowledge. We follow the non-interactive version of the definitions in [CD] and [GHM+22] for which the Fiat-Shamir heuristic applies. Let $\mathcal{R}$ be a binary relation, a Verifiable Encryption (VE) scheme for $\mathcal{R}$ is defined by the following PPT algorithms:

$\texttt{KGen}(1^\lambda) \to (\mathsf{dk}, \mathsf{ek})$: The key generation algorithm takes as input a security parameter 1^λ in unary form. It outputs a key pair $(\mathsf{dk}, \mathsf{ek})$.

$\texttt{Enc}(\mathsf{ek}, w) \to \mathsf{ct}$: The encryption algorithm takes as input a public key ek and a witness w. It outputs a ciphertext ct.

$\texttt{Verify}(\mathsf{ct}, \mathsf{ek}, x) \to b$: The verification algorithm takes as input a ciphertext ct, a public key ek and a statement x. It returns a bit b.

$\texttt{Dec}(\mathsf{dk}, \mathsf{ct}) \to w/\bot$: The decryption algorithm takes as input a decryption key dk and a ciphertext ct. It outputs a plaintext w or symbol $\bot$ on error.

Definition 5 (Correctness). *We say a verifiable encryption scheme is correct or complete if for all $\lambda \in \mathbb{N}$ and $(x, w) \in \mathcal{R}$,*

$$\Pr\left[\begin{array}{c} (\mathsf{dk}, \mathsf{ek}) \leftarrow \texttt{KGen}(1^\lambda),\ \mathsf{ct} \leftarrow \texttt{Enc}(\mathsf{ek}, w), \\ b \leftarrow \texttt{Verify}(\mathsf{ct}, \mathsf{ek}, x),\ w' \leftarrow \texttt{Dec}(\mathsf{dk}, \mathsf{ct}) : b \neq 1 \vee (x, w') \notin \mathcal{R} \end{array}\right] = \mathsf{negl}(\lambda).$$

where the probability is taken over the random coins of $\texttt{KGen}$ *and* $\texttt{Enc}$.

Definition 6 (Security). *A verifiable encryption scheme is said to be valid if for all $\lambda \in \mathbb{N}$ and adversaries $\mathcal{A}$,*

$$\mathsf{Adv}^{\mathsf{validity}}_{\mathsf{VE},\mathcal{A}}(\lambda) := \Pr\Big[\mathsf{Exp}^{\mathsf{validity}}_{\mathsf{VE},\mathcal{A}}(\lambda) = 1\Big] = \mathsf{negl}(\lambda).$$

The scheme is Honest-Verifier Zero-Knowledge (HVZK) if for all $\lambda \in \mathbb{N}$, there exists a simulator Sim *such that for all distinguisher $\mathcal{A}$ and $(x, w) \in \mathcal{R}$,*

$$\mathsf{Adv}^{\mathsf{hvzk}}_{\mathsf{VE},\mathcal{A}}(\lambda) := \left|\Pr\Big[\mathsf{Exp}^{\mathsf{hvzk}}_{\mathsf{VE},\mathcal{A}}(\lambda) = 1\Big] - \frac{1}{2}\right| = \mathsf{negl}(\lambda).$$

Experiments $\mathsf{Exp}^{\mathsf{validity}}_{\mathsf{VE},\mathcal{A}}$ and $\mathsf{Exp}^{\mathsf{hvzk}}_{\mathsf{VE},\mathcal{A}}$ are defined in Figure 2. Probabilities are taken over the random coins of algorithms $\texttt{KGen}$ *and* $\texttt{Enc}$.

```
$\mathsf{Exp}^{\mathsf{validity}}_{\mathsf{VE},\mathcal{A}}(\lambda)$
$(\mathsf{dk}, \mathsf{ek}) \leftarrow \texttt{KGen}(1^\lambda)$
$(x, \mathsf{ct}) \leftarrow \mathcal{A}(\mathsf{dk}, \mathsf{ek})$
$b \leftarrow \texttt{Verify}(\mathsf{ct}, \mathsf{ek}, x)$
$w' \leftarrow \texttt{Dec}(\mathsf{dk}, \mathsf{ct})$
return $b == 1 \wedge (x, w') \notin \mathcal{R}$
```

```
$\mathsf{Exp}^{\mathsf{hvzk}}_{\mathsf{VE},\mathcal{A}}(\lambda)$
$(\mathsf{dk}, \mathsf{ek}) \leftarrow \texttt{KGen}(1^\lambda)$
$\mathsf{ct}_0 \leftarrow \texttt{Enc}(\mathsf{ek}, w)$
$\mathsf{ct}_1 \leftarrow \mathsf{Sim}(\mathsf{ek}, x)$
$b \leftarrow\$ \{0, 1\}$
$b' \leftarrow \mathcal{A}(\mathsf{ek}, x, \mathsf{ct}_b)$
return $b == b'$
```

Fig. 2: Security experiments for Verifiable Encryption [CD].

2.4 Threshold Secret Sharing

A Threshold Secret Sharing (TSS) scheme allows a dealer to split a secret into shares such that any subset of shares above a certain threshold reveals the secret. We define TSS following the syntax of [AMN+21] based on [Bei11] by the following PPT algorithms:

$\texttt{Setup}(1^\lambda)$: takes as input a security parameter 1^λ and outputs a description pp of the public parameters.

$\texttt{Share}(\mathsf{pp}, n, t, S)$: takes as input a public parameter, two integers $t \leq n$ and a secret S. It outputs n shares $(s_i)_{i\in[n]}$.

$\texttt{Recon}(\mathsf{pp}, (s_i)_{i\in I})$: takes as input a public parameter and t shares $(s_i)_{i\in I}$ with $I \subseteq [n]$. It either outputs a secret S or $\perp$ on error.

A TSS scheme is correct if for all $\lambda \in \mathbb{N}$, $1 \leq t \leq n$, for all secret S, and $I \subset \mathbb{N}$ with $|I| = t$, it holds that

$$\Pr\left[\begin{array}{c} \mathsf{pp} \leftarrow \texttt{Setup}(1^\lambda),\ (s_i)_{i\in[n]} \leftarrow \texttt{Share}(\mathsf{pp}, n, t, S) : \\ \texttt{Recon}(\mathsf{pp}, (s_i)_{i\in I}) \neq S \end{array}\right] = \mathsf{negl}(\lambda).$$

Security-wise, a TSS scheme offers *privacy* if the following holds

$$\mathsf{Adv}^{\mathsf{privacy}}_{\mathsf{TSS},\mathcal{A}}(\lambda) := \left|\Pr\left[\mathsf{Exp}^{\mathsf{privacy}}_{\mathsf{TSS},\mathcal{A}}(\lambda) = 1\right] - \frac{1}{2}\right| \leq \mathsf{negl}(\lambda)$$

where the $\mathsf{Exp}^{\mathsf{privacy}}_{\mathsf{TSS},\mathcal{A}}(\lambda)$ experiment is defined in Figure 3.

```
$\mathsf{Exp}^{\mathsf{privacy}}_{\mathsf{TSS},\mathcal{A}}(\lambda)$
$(t, n) \leftarrow \mathcal{A}(1^\lambda)$
$\mathsf{pp} \leftarrow \texttt{Setup}(1^\lambda, t, n)$
$m_0, m_1 \leftarrow\!\!\$\ \{0,1\}^k$
$(s_i^0)_{i\in[n]} \leftarrow \texttt{Share}(\mathsf{pp}, m_0)$
$(s_i^1)_{i\in[n]} \leftarrow \texttt{Share}(\mathsf{pp}, m_1)$
$I \leftarrow\!\!\$\ \{J \subset [n] \mid \#J < t\}$
$b \leftarrow\!\!\$\ \{0,1\}$
$b' \leftarrow \mathcal{A}(\mathsf{pp}, m_0, m_1, (s_i^b)_{i\in I})$
return $b \overset{?}{=} b'$
```

Fig. 3: Privacy security experiment for threshold secret sharing.

3 1PVAKA: One-Round Key Agreement for dARKG

Designing a distributed version of ARKG involves selecting a proper communication model between the delegator and the many proxies. Assuming derived secret and public keys are uniquely bound to each other, an efficient non-interactive generalization of ARKG to the distributed setting implies[2] the existence of an efficient multiparty non-interactive key-exchange scheme [BZ]. These schemes are notoriously difficult to construct, especially against adaptive adversaries [Rao,KWZ22]. We therefore focus on the next minimal constraint model allowing a single round between each proxy and the delegator but no communication between proxies.

To accommodate this setup and simultaneously achieve modularity in our dARKG construction, we introduce a one-round publicly verifiable asymmetric key agreement scheme named 1PVAKA. Looking ahead to our dARKG construction, proxies in a group selected by the delegator will asynchronously produce 1PVAKA secret shares and dealings. The delegator then produces a static public key for the group after verifying the dealings.

Our definition of 1PVAKA is close to the ASGKA construction in [WMS+] as well as Groth's NI-DKG scheme in [Gro]. It is, however, publicly verifiable, built generically for an arbitrary AKG scheme and does not require re-sharing or thresholding mechanisms. The entity in charge of selecting the recovery threshold in our construction of dARKG is the delegator, not the proxies.

3.1 Syntax and security definitions of 1PVAKA

Let Δ be an AKG scheme with key spaces $\mathbb{S}$ and $\mathbb{P}$. A one-round publicly verifiable asymmetric key agreement 1PVAKA) for Δ is defined by the following PPT algorithms:

$\texttt{Setup}(1^\lambda, 1^m) \to \mathsf{pp}$: The setup algorithm takes as input a security parameter λ and a maximum user group size m in unary representation. It outputs a description pp of the public parameters of the scheme.

$\texttt{KGen}(\mathsf{pp}) \to (\mathsf{dk}, \mathsf{ek})$: The long-term key generation algorithm takes as input the public parameters pp. It outputs a long-term key pair $(\mathsf{dk}, \mathsf{ek})$.

$\texttt{Deal}([\mathsf{ek}]_n) \to (\mathsf{sh}, \mathsf{d})$: The dealing algorithm takes as input n long-term public keys $[\mathsf{ek}]_n$. It outputs a secret share sh and a public dealing d.

$\texttt{Verify}([\mathsf{ek}]_n, \mathsf{d}) \to b$: The verification algorithms takes as input public keys for a group of users and a dealing d. It outputs a bit b.

$\texttt{CombinePK}([\mathsf{d}]_l) \to (\mathsf{pk}, [\mathsf{f}]_l)$: The public key combination algorithm takes as input dealings of a group of users. It outputs a (group) public key $\mathsf{pk} \in \mathbb{P}$ for the group and new public dealings $[\mathsf{f}]_n$ or $\perp$ on error.

[2] Assuming a threshold of $t = 1$, in ARKG suppose $\texttt{DerivePK}$ now takes as input public keys $\mathsf{pk}_1, \ldots, \mathsf{pk}_n$ for a group of users and returns public values $(\mathsf{pk}', \mathsf{cred})$. Running $\texttt{DeriveSK}(\mathsf{sk}_i, \mathsf{cred}) \to \mathsf{sk}'$ yields the unique secret key bound to pk' which is a secret value shared by all users of the group.

$\texttt{CombineSK}(\mathsf{dk}, \mathsf{sh}, \mathsf{f}) \to \mathsf{sk}$ or $\perp$: The secret key combination algorithm takes as input a decryption key dk, a secret share sh and dealing f. It outputs a (group) secret key $\mathsf{sk} \in \mathbb{S}$ or symbol $\perp$ on error.

Definition 7 (Correctness). *A 1PVAKA scheme is correct if for all integers $\lambda, n, m \in \mathbb{N}$ with $n \leq m$, the following holds*

$$\Pr\left[\begin{array}{c} \mathsf{pp} \leftarrow \texttt{Setup}(1^\lambda, 1^m),\ (\mathsf{ek}_i, \mathsf{dk}_i) \leftarrow \texttt{KGen}(\mathsf{pp}) \textbf{ for } i \in [n], \\ (\mathsf{sh}_i, \mathsf{d}_i) \leftarrow \texttt{Deal}([\mathsf{ek}]_n) \textbf{ for } i \in [n] :\ \bigwedge_{i \in [n]} \texttt{Verify}([\mathsf{ek}]_n, \mathsf{d}_i) \neq 1 \end{array}\right] = \mathsf{negl}(\lambda).$$

If the sequence above succeeds, define the associated group public key and dealings $(\mathsf{pk}, [\mathsf{f}]_n) \leftarrow \texttt{CombinePK}([\mathsf{d}]_n)$, it should also hold for all $i \in [n]$ that

$$\Pr\big[\mathsf{sk} \leftarrow \texttt{CombineSK}(\mathsf{dk}_i, \mathsf{sh}_i, \mathsf{f}_i) :\ \mathit{\Delta}.\texttt{Check}(\mathsf{sk}, \mathsf{pk}) \neq 1\big] = \mathsf{negl}(\lambda).$$

We outline below the security concepts pertinent to 1PVAKA. We begin with key secrecy: in a scenario without threshold, a solitary user must be capable of retrieving a secret group key sk from its share sh and its secret key dk. Consequently, we permit exposure of either the share or the long-term secret key to potential adversaries. Our consideration of key secrecy is computational rather than decisional and suffices for our objective of constructing dARKG.

Global variables: list $\mathcal{L}$ and sets $\mathcal{S}$, $\mathcal{E}$ and $\mathcal{K}$ are initialized empty at the start of the security games. They respectively track session material, session identifiers, exposed session shares and exposed long-term keys.

Definition 8 (Key secrecy). *For security parameter λ, integers s, k and a PPT adversary $\mathcal{A}$, we define the $\mathsf{Exp}^{\mathsf{(s,k)\text{-}secrecy}}_{\mathsf{1PVAKA},\mathcal{A}}$ security experiment as follows.*

1. **Setup phase:** *One input λ and m, the challenger sets $\mathsf{pp} \leftarrow \texttt{Setup}(1^\lambda, 1^m)$ and returns pp to $\mathcal{A}$.*
2. **Query phase:** *During the execution of the game, adversary $\mathcal{A}$ can adaptively make the following queries in arbitrary order. Only a single challenge query can be made, the others are polynomially limited.*
   (a) **Key query:** *The challenger generates a key pair $(\mathsf{dk}, \mathsf{ek}) \leftarrow \texttt{KGen}(\mathsf{pp})$, stores dk and gives ek to $\mathcal{A}$.*
   (b) **Deal query:** *$\mathcal{A}$ submits session identifier sid and public keys $[\mathsf{ek}]_n$ to the challenger. If $\mathsf{sid} \in \mathcal{S}$, the session already exists: the challenger fetches $([\mathsf{ek}]'_n, L, i) := \mathcal{L}[\mathsf{sid}]$, checks that $[\mathsf{ek}]'_n = [\mathsf{ek}]_n$ and aborts otherwise. Else if $\mathsf{sid} \notin \mathcal{S}$, it registers the session $\mathcal{S} \leftarrow \mathcal{S} \cup \{\mathsf{sid}\}$ and defines $\mathcal{L}[\mathsf{sid}] := ([\mathsf{ek}]_n, L := \varnothing, i := 1)$. The challenger then computes $(\mathsf{sh}, \mathsf{d}) \leftarrow \texttt{Deal}([\mathsf{ek}]_n)$, gives d to $\mathcal{A}$ and updates $L \leftarrow L \cup \{(i, \mathsf{sh}, \mathsf{d})\}$ as well as $i \leftarrow i + 1$.*
   (c) **Expose share query:** *$\mathcal{A}$ submits a session identifier sid and an index i. The challenger looks for $(i, \mathsf{sh}, \mathsf{d}) \in L$ with $(\_, L, \_) := \mathcal{L}[\mathsf{sid}]$ and returns sh to $\mathcal{A}$ or $\perp$ if no such element exists. The challenger also sets $\mathcal{E} \leftarrow \mathcal{E} \cup \{(\mathsf{sid}, i)\}$.*

(d) **Expose key query:** *$\mathcal{A}$ submits a long-term public key* ek. *The game aborts if the public key was not generated during a key query* (*a*). *Otherwise, the challenger gives the corresponding decryption key* ek *to $\mathcal{A}$ and updates* $\mathcal{K} \leftarrow \mathcal{K} \cup \{\mathsf{ek}\}$.

(e) **Challenge query:** *At some point during the game, $\mathcal{A}$ submits a session identifier* sid *and a challenge secret key* $\mathsf{sk}^\star$. *The challenger fetches* $L = \{(i, \mathsf{sh}_i, \mathsf{d}_i)\}_{i\in[n]}$ *with* $([\mathsf{ek}]_n, L, n+1) := \mathcal{L}[\mathsf{sid}]$ *and aborts if either*

- *a public key in* $[\mathsf{ek}]_n$ *was not generated during phase* (*a*)
- *a dealing in L was not generated during phase* (*b*)
- *more than $l - s$ elements* (sid, i) *belong to* $\mathcal{E}$
- *more than k public keys in* $[\mathsf{ek}]_n$ *belong to* $\mathcal{K}$
- *there exists i such that* $(\mathsf{sid}, i) \in \mathcal{E}$ *and* $\mathsf{ek}_i \in \mathcal{K}$.

The challenger computes $(\mathsf{pk}, [\mathsf{f}]_l) \leftarrow \mathtt{CombinePK}([\mathsf{d}]_n)$, *aborting on error.*

3. **Output phase:** *The challenger outputs the result of* $\mathit{\Delta}.\mathtt{Check}(\mathsf{sk}^\star, \mathsf{pk})$.

We say a 1PVAKA scheme satisfies (s, k)-key-secrecy if for all security parameter λ and all stateful PPT adversaries $\mathcal{A}$, it holds that

$$\mathsf{Adv}^{\mathsf{(s,k)\text{-}secrecy}}_{\mathsf{1PVAKA},\mathcal{A}}(\lambda, m) := \Pr\left[\mathsf{Exp}^{\mathsf{(s,k)\text{-}secrecy}}_{\mathsf{1PVAKA},\mathcal{A}}(\lambda, m) = 1\right] = \mathsf{negl}(\lambda).$$

In the above experiment, session indices are introduced together with virtual user identifier i and reflect our use of 1PVAKA in the context of distributed ARKG. Without those constrains, the adversary could use any dealings for any group of user of arbitrary size in a challenge session. This is irrelevant in the presence of an honest but curious delegator which selects the group of proxies it needs to generate a static public key. These tracking values thus help tighten the reduction for the key-secrecy property in our 1PVAKA construction.

The following definition of *robustness* states that if an honest user participates in a 1PVAKA session, an adversary controlling up to all remaining parties cannot force this user to derive an incorrect secret key sk while producing valid dealings. This definition is relevant in any situation where dealings are asynchronously prepared by the users and collected, verified and processed by a third party (the delegator in our case). Generally, the combined public key pk may be used for an extended period of time before users retrieve their group secret key sk. It is not acceptable to notice only at this later stage that sk is invalid.

Definition 9 (Robustness). *We say a 1PVAKA scheme is robust if for all integers λ, m and all stateful PPT adversaries $\mathcal{A}$, it holds that*

$$\mathsf{Adv}^{\mathsf{robustness}}_{\mathsf{1PVAKA},\mathcal{A}}(\lambda, m) := \Pr\left[\mathsf{Exp}^{\mathsf{robustness}}_{\mathsf{1PVAKA},\mathcal{A}}(\lambda, m) = 1\right] = \mathsf{negl}(\lambda).$$

where experiment $\mathsf{Exp}^{\mathsf{robustness}}_{\mathsf{1PVAKA},\mathcal{A}}$ *is defined in Figure 4 and the probability is taken over the random coins of* $\mathtt{KGen}$ *and* $\mathtt{Deal}$.

3.2 1PVAKA Construction from Verifiable Encryption

Let $\mathit{\Delta}$ be an AKG scheme with key spaces $\mathbb{S}$ and $\mathbb{P}$. Define binary relation $\mathcal{R}$ over $\mathbb{S} \times \mathbb{P}$ by $\mathsf{sk}\mathcal{R}\mathsf{pk}$ if and only if $\mathit{\Delta}.\mathtt{Check}(\mathsf{sk}, \mathsf{pk}) = 1$. Let $\mathit{\Pi}$ be an m-times additively homomorphic verifiable encryption scheme for relation $\mathcal{R}$. That is $\mathit{\Pi}.\mathtt{Enc}(\mathsf{ek}, \mathsf{sk}_1 + \cdots + \mathsf{sk}_m) = \mathsf{ct}_1 + \cdots + \mathsf{ct}_m$ where $\mathsf{ct}_k = \mathit{\Pi}.\mathtt{Enc}(\mathsf{ek}, \mathsf{sk}_k)$.

```
Experiment $\mathsf{Exp}^{\mathsf{robustness}}_{\mathsf{1PVAKA},\mathcal{A}}(\lambda, m)$:
$\mathsf{pp} \leftarrow \mathtt{Setup}(1^\lambda, 1^m)$
$(\mathsf{dk}_1, \mathsf{ek}_1) \leftarrow \mathtt{KGen}(\mathsf{pp})$ // Register honest user
$(\mathsf{ek}_2, \ldots, \mathsf{ek}_n) \leftarrow \mathcal{A}(\mathsf{pp}, \mathsf{ek}_1)$
$(\mathsf{sh}_1, \mathsf{d}_1) \leftarrow \mathtt{Deal}([\mathsf{ek}]_n)$
$(\mathsf{d}_2, \ldots, \mathsf{d}_n) \leftarrow \mathcal{A}(\mathsf{d}_1)$ // Adaptively create dealings
$(\mathsf{pk}, [\mathsf{f}]_n) \leftarrow \mathtt{CombinePK}([\mathsf{d}]_n)$
$\mathsf{sk} \leftarrow \mathtt{CombineSK}(\mathsf{dk}_1, \mathsf{sh}_1, \mathsf{f}_1)$
return $(\neg \mathit{\Delta}.\mathtt{Check}(\mathsf{sk}, \mathsf{pk})) \wedge \bigwedge_{1 \le k \le n} \mathtt{Verify}([\mathsf{ek}]_n, \mathsf{d}_k)$
```

Fig. 4: Robustness security experiment for 1PVAKA. The adversary produces seemingly valid public dealings but the honest user gets an invalid secret key.

Definition 10 (Transformation from VE to 1PVAKA). *A generic construction of 1PVAKA using the above AKG and VE is obtained as follows:*

Setup: *Generate and output public parameters* $\mathsf{pp}_\mathit{\Delta}$ *and* $\mathsf{pp}_\mathit{\Pi}$ *for* $\mathit{\Delta}$ *and* $\mathit{\Pi}$.

Key generation: *Generate and output* $(\mathsf{dk}, \mathsf{ek}) \leftarrow \mathit{\Pi}.\mathsf{KGen}(\mathsf{pp}_\mathit{\Pi})$.

Deal: *On input* $[\mathsf{ek}]_n$*, sample a key pair* $(\mathsf{sk}, \mathsf{pk}) \leftarrow \mathit{\Delta}.\mathsf{KGen}(\mathsf{pp}_\mathit{\Delta})$*. Set* $\mathsf{sh} := \mathsf{sk}$*,* $[\mathsf{ct}]_n \leftarrow (\mathit{\Pi}.\mathtt{Enc}(\mathsf{ek}_j, \mathsf{sk}))_{j \in [n]}$ *and* $\mathsf{d} := (\mathsf{pk}, [\mathsf{ct}]_n)$*. Output* $(\mathsf{sh}, \mathsf{d})$.

Verify: *To verify dealing* $\mathsf{d} =: (\mathsf{pk}, [\mathsf{ct}]_n)$*, verify individual ciphertexts against statement* pk *and output* 0 *if one fails. Otherwise, output* 1.

Combine public keys: *For input dealings* $[\mathsf{d}]_l$ *parse* $\mathsf{d}_i =: (\mathsf{pk}_i, [\mathsf{ct}_i]_l)$*. Define* $\mathsf{pk} := \mathsf{pk}_1 \cdots \mathsf{pk}_l$ *and set* $\mathsf{f}_k := \sum_{i \neq k} (\mathsf{ct}_i)_k$*. Output values* $(\mathsf{pk}, [\mathsf{f}]_l)$.

Combine secret keys: *For input decryption key* dk*, share* sh *and dealing* f*, decrypt* f *using* dk *to obtain partial secret key* $\mathsf{sk}_{\mathsf{part}}$*. Output combined secret key* $\mathsf{sk} := \mathsf{sh} + \mathsf{sk}_{\mathsf{part}}$.

Correctness of the transformation follows from that of $\mathit{\Delta}$ and $\mathit{\Pi}$ as well as the m-times homomorphic property of $\mathit{\Pi}$.

Lemma 1 (Key secrecy of 1PVAKA). *Assuming* $\mathit{\Delta}$ *is secure and* $\mathit{\Pi}$ *is HVZK, the 1PVAKA scheme defined above provides* $(1,1)$*-key secrecy. More precisely, for all* λ, m *and for every adversary* $\mathcal{A}$ *against the* $\mathsf{Adv}^{\mathsf{(1,1)\text{-}secrecy}}_{\mathsf{1PVAKA},\mathcal{A}}$ *experiment, there exists an adversary* $\mathcal{B} = (\mathcal{B}_1, \mathcal{B}_2)$ *such that*

$$\mathsf{Adv}^{\mathsf{(1,1)\text{-}secrecy}}_{\mathsf{1PVAKA},\mathcal{A}}(\lambda, m) \le q \left(m \mathsf{Adv}^{\mathsf{hvzk}}_{\mathsf{VE},\mathcal{B}_1}(\lambda) + \mathsf{Adv}^{\mathsf{ks}}_{\mathsf{AKG},\mathcal{B}_2}(\lambda) \right)$$

where q *is the number of sessions started in* $\mathsf{Exp}^{\mathsf{(s,k)\text{-}secrecy}}_{\mathsf{1PVAKA},\mathcal{A}}$.

Proof. Let us start by explaining the choice of values for s and k. If more than 2 long-term keys are available to the adversary, this one can trivially compute the session secret key sk by deciphering two distinct dealings and win the game, so $k \in \{0,1\}$. Up to $n-1 \leq m-1$ shares can be exposed to the adversary without reducing the hidden randomness of sk in $\mathtt{CombineSK}$. This is because the last share acts as a one-time-pad by addition: $\mathsf{sk} = \mathsf{sh}_1 + \cdots + \mathsf{sh}_l$.

Let us consider an attacker $\mathcal{A}$ for experiment $\mathsf{Exp}^{\mathsf{(s,k)\text{-}secrecy}}_{\mathsf{1PVAKA},\mathcal{A}}$. We build an attacker $\mathcal{B}_2$ for the $\mathsf{Exp}^{\mathsf{ks}}_{\mathsf{AKG},\mathcal{B}_2}$ experiment via game-hopping.

$\underline{\mathsf{Game}_0(\lambda, m)}$: this game is defined exactly by $\mathsf{Exp}^{\mathsf{(s,k)\text{-}secrecy}}_{\mathsf{1PVAKA},\mathcal{A}}$, thus

$$\mathsf{Adv}^{\mathsf{(s,k)\text{-}secrecy}}_{\mathsf{1PVAKA},\mathcal{A}}(\lambda, m) = \Pr[\mathsf{Game}_0(\lambda, m) = 1].$$

$\underline{\mathsf{Game}_1(\lambda, m)}$: this game is defined as $\mathsf{Game}_0(\lambda, m)$ with additional value samples at the start

$$\mathsf{sid}^\star \leftarrow\!\!\$\, [q];\;\; \mathsf{pk}^\star \leftarrow \mathbb{P}.$$

This session identifier is a guess by the reduction of the session that the adversary will choose to attack in the challenge query. The public key is sampled as challenge for the AKG key-secrecy experiment. The probability for the game to correctly guess sid is bound by $1/q$, thus

$$\Pr[\mathsf{Game}_0(\lambda, m) = 1] \leq q \Pr[\mathsf{Game}_1(\lambda, m) = 1].$$

$\underline{\mathsf{Game}_2(\lambda, m)}$: in this game, the oracle queries made by adversary $\mathcal{A}$ are answered honestly except for queries to $\mathsf{O}_{\mathsf{deal}}$ for the predicted session sid. In that session, no corruption by $\mathcal{A}$ will take place according to the conditions of success. It can thus be run entirely by the game with relevant shares being exposed to the adversary.

We assume that $n - s$ user shares are revealed and the reduction guesses the index of the $s \in \{1,2\}$ non-revealed ones: i_0 (and i_1 if $s = 2$). This adds a factor of s/m to the reduction.

The deal queries are executed honestly for $k \in [n] \backslash \{i_0\}$ who receive shares sh_j and dealings d_j containing public keys pk_j. The game sets the following public key in the dealing of user i_0 to embed the challenge:

$$\mathsf{pk}_{i_0} := \mathsf{pk}^\star \cdot (\prod_{j \neq i} \mathsf{pk}_j)^{-1}.$$

In the above expression we assume the dealing is ran with i_0 last, so $i_0 = n$. We can make this assumption without loss of generality as all players are honest in this session.

During the oracle call to $\mathsf{O}_{\mathsf{deal}}$ for i_0, the game cannot run the verifiable encryption algorithm as it does not know the secret key $\mathsf{sk}^\star$ which acts as witness. However, according to the Honest Verifier Zero-Knowledge security assumption on Π, there exists a simulator Sim producing ciphertexts for input $\mathsf{pk}^\star$ indistinguishable from the real outputs of the encryption algorithm.

Such ciphertexts ($n-1 \leq m-1$ of them) are used by the game to simulate the encryption of $\mathsf{sk}^\star$. The advantage of an adversary in distinguishing this particular execution from the real one is thus bound by that of an adversary $\mathcal{B}_1$ against the experiment $\mathsf{Exp}^{\mathsf{hvzk}}_{\mathsf{VE},\mathcal{B}_1}$.

Finally, the adversary can expose the long-term key associated to either i_0 or i_1 if $k = 1$. With a minimum probability of 1/2, it will guess i_0 and will be able to test the aforementioned simulated ciphertexts and distinguish the two games. Summing up these results we obtain

$$|\Pr[\mathsf{Game}_1(\lambda, m) = 1] - \Pr[\mathsf{Game}_2(\lambda, m) = 1]| \leq \frac{2^k m}{s} \mathsf{Adv}^{\mathsf{hvzk}}_{\mathsf{VE},\mathcal{B}_1}(\lambda).$$

And $\frac{2^k m}{s} = m$ in both cases since $k = s - 1$ and $s \in \{1, 2\}$.

In this last game, the output public key of the challenge oracle is $\mathsf{pk}^\star$. As such, adversary $\mathcal{A}$ replies with a candidate secret key $\mathsf{sk}^\star$ for the AKG Key-Secrecy challenge public key. The advantage of an adversary in this game is thus bound by that of an adversary $\mathcal{B}_2$ against experiment $\mathsf{Exp}^{\mathsf{ks}}_{\mathsf{AKG},\mathcal{B}_2}$. Thus

$$\Pr[\mathsf{Game}_2(\lambda, m) = 1] \leq \mathsf{Adv}^{\mathsf{ks}}_{\mathsf{AKG},\mathcal{B}_2}(\lambda).$$

□

Lemma 2 (Robustness of 1PVAKA). *If the VE scheme Π is valid, then the 1PVAKA scheme obtained via our transformation is robust. More precisely, for all $\lambda, m \in \mathbb{N}$, there exists an adversary $\mathcal{B}$ against the $\mathsf{Exp}^{\mathsf{validity}}_{\mathsf{VE},\mathcal{B}}$ experiment such that $\mathsf{Adv}^{\mathsf{robustness}}_{\mathsf{1PVAKA},\mathcal{A}}(\lambda, m) \leq \mathsf{Adv}^{\mathsf{validity}}_{\mathsf{VE},\mathcal{B}}(\lambda)$.*

Proof. Let us consider an attacker $\mathcal{A}$ against experiment $\mathsf{Exp}^{\mathsf{robustness}}_{\mathsf{1PVAKA},\mathcal{A}}$. We build an adversary $\mathcal{B}$ against the $\mathsf{Exp}^{\mathsf{validity}}_{\mathsf{VE},\mathcal{A}}$ experiment.

The challenger generates key pair $(\mathsf{ek}, \mathsf{dk})$ and sends it to $\mathcal{B}$. Set the challenge key pair as user $\mathcal{U}_1$'s long-term 1PVAKA key pair. Adversary $\mathcal{B}$ now starts using $\mathcal{A}$ against the $\mathsf{robustness}$ experiment. On line 4 of $\mathsf{Exp}^{\mathsf{robustness}}_{\mathsf{1PVAKA},\mathcal{A}}$, a share and a dealing are obtained for $\mathcal{U}_1$:

$$(\mathsf{sh}_1, \mathsf{d}_1) \leftarrow \mathtt{Deal}(\mathsf{ek}_1, \ldots, \mathsf{ek}_n).$$

In the $\mathsf{Exp}^{\mathsf{validity}}_{\mathsf{VE},\mathcal{A}}$ experiment against the challenger, we simulate $\mathcal{P}^*$ exactly as $\mathcal{U}_1$, using $\mathcal{B}$ as intermediary.

Assuming $\mathcal{A}$ wins the $\mathsf{robustness}$ experiment, user $\mathcal{U}_1$ has reconstructed a secret key sk which is not valid for public key pk. By the correctness of AKG, this means that for some k, one of the qualified ciphertexts $(\mathsf{ct}_{1,k})$ does not decrypt to the secret key corresponding to pk. Since $\mathtt{Verify}([\mathsf{ek}]_n, \mathsf{d}_k) = 1$ for all $k \in [n]$, adversary $\mathcal{B}$ won against the $\mathsf{validity}$ experiment by producing a ciphertext $\mathsf{ct}_{1,i}$ that decrypts to a statement which does not correspond to the public value provided to the verifier. □

4 Distributed Asynchronous Remote Key Generation

In this section we present the syntax and security properties of distributed ARKG (dARKG) as well as its generic construction from 1PVAKA. Throughout the entire section we consider the universe $\mathcal{U} = \{\mathcal{U}_i\}_{i \in [N]}$ of all users acting as *proxies* and an additional user $\mathcal{D}$ called *delegator* which may or may not be part of $\mathcal{U}$. We choose to work with a single delegator rather than many because delegators only execute algorithms on public inputs.

4.1 Syntax of dARKG

Let $\mathit{\Delta}$ be an AKG scheme with key spaces $\mathbb{S}$ and $\mathbb{P}$. A dARKG scheme for $\mathit{\Delta}$ is defined by the following PPT algorithms:

$\mathtt{Setup}(1^\lambda, 1^m) \to \mathsf{pp}$: The setup algorithm takes as input a security parameter λ as well as a maximum group size m in unary representation. It outputs a description pp of the public parameters available to the following algorithms.

$\mathtt{LTKGen}(\mathsf{pp}) \to (\mathsf{dk}, \mathsf{ek})$: The long-term key generation algorithm takes as input public parameters pp and outputs a long-term key pair $(\mathsf{dk}, \mathsf{ek})$. The public key ek is publicly available using the generating proxy's index. Running the algorithm multiple times updates this record.

$\mathtt{KGenProxy}([\mathsf{ek}]_n) \to (\mathsf{sh}, \mathsf{d})$: The key generation algorithm for proxies takes as input a set of proxy public keys $[\mathsf{ek}]_n$. It outputs a share sh and a dealing d.

$\mathtt{KGenDeleg}([\mathsf{ek}]_n, [\mathsf{d}]_n) \to (\mathsf{qual}, \mathsf{pk}, [\mathsf{f}]_l)/\bot$: The key generation algorithm for the delegator takes as input a set of proxy public keys $[\mathsf{ek}]_n$ and corresponding dealings $[\mathsf{d}]_n$. It outputs a static public key $\mathsf{pk} \in \mathbb{P}$, a set of qualified proxies $\mathsf{qual} \subset [n]$ of size $l \leq n$ and l new dealings.

$\mathtt{DerivePK}(\mathsf{pk}, \mathsf{aux}) \to (\mathsf{pk}^\star, \mathsf{cred})$: The public key derivation algorithm takes as input public a public key $\mathsf{pk} \in \mathbb{P}$ and auxiliary information. It outputs a derived public key $\mathsf{pk}^\star \in \mathbb{P}$ and credentials cred.

$\mathtt{ShareDPK}(\mathsf{pk}^\star, \mathsf{cred}, t, [\mathsf{ek}]_l, [\mathsf{f}]_l) \to (\mathsf{pk}', [\mathsf{cred}']_l)$: The derived public key sharing algorithm takes as input a public key $\mathsf{pk}^\star$, credentials cred, a threshold t, a set of proxy indices public keys $[\mathsf{ek}]_l$ and dealings $[\mathsf{f}]_l$. It outputs a derived public key pk' and l new credentials or symbol $\bot$ on failure.

$\mathtt{DeriveSKSh}(\mathsf{dk}, \mathsf{sh}, \mathsf{cred}') \to \mathsf{sh}'$: The secret key share derivation algorithm takes as input a long-term decryption key dk, a share sh and credentials cred'. It outputs a secret share sh'.

$\mathtt{DeriveSK}\langle(\mathcal{U}_i)_{i\in J})\rangle \to \mathsf{sk}'$ or $\bot$: The secret key derivation protocol between $t = |J|$ proxies of $\mathcal{U}$. Proxy $\mathcal{U}_i$'s secret input correspond to a secret share sh'_i. It outputs a derived secret key $\mathsf{sk}' \in \mathbb{S}$ or $\bot$ on failure.

In practice, proxies in $\mathcal{U}$ provide long-term keys available to the delegator. To derive a public key pk', the later selects public keys $[\mathsf{ek}]_n$ and asks the proxies to generate their share and dealing. A static public key pk bound to that group can be generated by the delegator using these dealings. This key is then used similarly to the ARKG construction with an added thresholding step.

Definition 11 (Correctness). *Let integers $\lambda, m, n, t \in \mathbb{N}$ be such that $t \leq n \leq m$ and let $I \subset [N]$ be of size n which we index as $[n]$ for convenience. Let* $\mathsf{pp} \leftarrow \mathtt{Setup}(1^\lambda, 1^m)$*. A dARKG scheme is correct if for all such selection of values, the execution sequence in Figure 5 yields* $\mathit{\Delta}.\mathtt{Check}(\mathsf{sk}', \mathsf{pk}') = 1$.

$$\left.\begin{array}{l}(\mathsf{dk}_i, \mathsf{ek}_i) \leftarrow \mathtt{LTKGen}(\mathsf{pp})\ \forall i \in I\end{array}\right\}\quad(1)$$

$$\left.\begin{array}{lr}(\mathsf{sh}_i, \mathsf{d}_i) \leftarrow \mathtt{KGenProxy}([\mathsf{ek}]_n)\ \forall i \in I & \\ & \mathtt{KGenDeleg}([\mathsf{ek}]_n, [\mathsf{d}]_n) \rightarrow (\mathsf{qual}, \mathsf{pk}, [\mathsf{f}]_l)\end{array}\right\}\quad(2)$$

$$\left.\begin{array}{r}\mathtt{DerivePK}(\mathsf{pk}) \rightarrow (\mathsf{pk}^\star, \mathsf{cred}) \\ \mathtt{ShareDPK}(\mathsf{pk}^\star, \mathsf{cred}, t, I, [\mathsf{f}]_l) \rightarrow (\mathsf{pk}', [\mathsf{cred}']_l)\end{array}\right\}\quad(3)$$

$$\left.\begin{array}{l}\mathsf{sh}'_i \leftarrow \mathtt{DeriveSKSh}(\mathsf{dk}_i, \mathsf{sh}_i, \mathsf{cred}'_i)\ \forall i \in \mathsf{qual} \\ \text{Pick a subset } J \subset \mathsf{qual} \text{ of size } t \text{ at random} \\ \mathsf{sk}' \leftarrow \mathtt{DeriveSKRec}\langle(\mathcal{U}_i)_{i \in J})\rangle\end{array}\right\}\quad(4)$$

Fig. 5: Correct execution flow of dARKG. Operations on the left are performed by the proxies, those on the right by the delegator. Phase (1) corresponds to registration of long-term public keys. (2) is a static group key generation session for proxies in I. In (3), the delegator derives a key pk' for group I. (4) corresponds to secret share generation and recovery of the derived secret key sk'.

A session roughly corresponds to algorithm `KGen` in the original ARKG syntax [FGK+]. It consists in each proxy of $[\mathsf{ek}]_n$ asynchronously executing `KGenProxy` on input $[\mathsf{ek}]_n$ and sending their dealing to the delegator. The delegator does not necessarily wait for all the proxies to respond (at least t) and at some point executes `KGenDeleg` using some or all of these dealings. At this point, a static public key pk is generated for the group (or a qualified subgroup) along with public recovery information in the form of dealings.

The delegator can then run `DerivePK` and potentially `ShareDPK` (to add a threshold to the recovery) on those outputs to produce a derived public key pk' and credentials for the qualified proxies. When a qualified proxy gains access to its credential, it is able to compute a secret key share via `DeriveSKSh` which can be used either directly as a distributed secret key or to reconstruct the derived secret key sk' in protocol `DeriveSK`. Figure 5 illustrates the flow of execution of dARKG.

Let us compare our definition of dARKG to that of ARKG. Algorithm `LTKGen` is a new requirement in the distributed setting allowing proxies to publish long term public keys. Algorithms `KGenProxy` and `KGenDeleg` are to dARKG what the key generation algorithm `KGen` is to ARKG. These are used to generate a public key pk associated with a group of proxies rather than a single user. As the name implies, dealings are used in the same way as in a 1PVAKA scheme, i.e., to reconstruct a group secret key sk corresponding to pk at a later time.

The public key derivation algorithm `DerivePK` is kept identical mainly for convenient retro-compatibility. Although defined separately, algorithm `ShareDPK` may be thought of as being part of the public key derivation process in the distributed setting as its purpose is to add a threshold to the derived secret key.

Finally, `DeriveSKSh` and `DeriveSKRec` correspond to a distributed version of `DeriveSK` in ARKG. The first one outputs shares of the derived secret key while the second one uses those shares to reconstruct the derived secret key. Technically, protocol DeriveSKRec is present in the syntax first and foremost to define a threshold functionality. It is used to define the correctness and security of the scheme with respect to a threshold cryptosystem. The shares output by DeriveSKSh might for instance be used directly in a threshold signature scheme such as BLS without ever using DeriveSKRec. However, DeriveSKRec may as well be a concrete MPC protocol with secret output sk'.

4.2 Security properties of dARKG

We adapt the security notions for ARKG schemes [FGK+] to the distributed setting.

We assume that all proxies in $\mathcal{U}$ have generated long-term key pairs $(\mathsf{dk}_i, \mathsf{ek}_i)$ with the public keys available to the delegator. The security games execute algorithms KGenProxy, KGenDeleg, DerivePK and ShareDPK on behalf of honest proxies and the delegator. This corresponds to an honest but curious delegator and implies authenticated channels between proxy and delegator. Share exposure and proxy corruption is handled through calls to the oracle defined below.
Global variables: Set $\mathcal{K}$ contains exposed long-term keys. Lists $\mathcal{L}$, $\mathcal{L}_{\mathcal{C}}$, $\mathcal{L}_{\mathcal{E}}$, $\mathcal{L}_{\mathcal{E}'}$ respectively contain static key sessions data, corrupted shares, exposed shares and exposed derived shares. Three other lists $\mathcal{L}_{\mathsf{pk}}$, $\mathcal{L}_{\mathsf{pk}'}$ and $\mathcal{L}_{\mathsf{sk}'}$ are used similarly to the original ARKG definitions.

$\mathsf{O}_{\mathsf{KGenProx}}([\mathsf{ek}]_n, \mathsf{sid}, \mathsf{res}^\star)$: This oracle takes as input proxy public keys $[\mathsf{ek}]_n$, session identifier $\mathsf{sid} \in \mathbb{N}$ and optional variable $\mathsf{res}^\star$. It fetches a session tuple of the form $([\mathsf{ek}]_n, L := \{(i, \mathsf{sh}_i, \mathsf{d}_i)\}_{i\in[k]}, k) := \mathcal{L}[\mathsf{sid}]$ or initializes it with $k = 0$ and $L := \varnothing$. If $\mathsf{res}^\star$ is adversarially provided as $(\mathsf{sh}, \mathsf{d})$, the oracle sets $\mathcal{L}_{\mathcal{C}}[\mathsf{sid}] \leftarrow \mathcal{L}_{\mathcal{C}}[\mathsf{sid}] \cup \{k+1\}$, otherwise it honestly executes a dealing $(\mathsf{sh}, \mathsf{d}) \leftarrow \mathtt{KGenProxy}(\mathsf{pp}, [\mathsf{ek}]_n)$. The oracle updates $L \leftarrow L \cup \{(k+1, \mathsf{sh}, \mathsf{d})\}$ and $k \leftarrow k+1$ in $\mathcal{L}[\mathsf{sid}]$. It returns d to the adversary.

$\mathsf{O}_{\mathsf{LTKExpose}}(\mathsf{ek})$: This oracle takes as input a proxy public key ek. It adds ek to set $\mathcal{K}$ and returns the corresponding secret key dk.

$\mathsf{O}_{\mathsf{ShExpose}}(\mathsf{sid}, k)$: This oracle takes as input session identifier sid and integer k. It fetches $(k, \mathsf{sh}_k, \mathsf{d}_k) \in L$ for L in $\mathcal{L}[\mathsf{sid}]$ and aborts if not found. The oracle updates $\mathcal{L}_{\mathcal{E}}[\mathsf{sid}] \leftarrow \mathcal{L}_{\mathcal{E}}[\mathsf{sid}] \cup \{k\}$ and returns sh_k.

$\mathsf{O}_{\mathsf{Sh'Expose}}(\mathsf{sid}, k, [\mathsf{cred}']_l)$: This oracle takes as input a session identifier sid, a player index k and credentials $[\mathsf{cred}']_l$. It fetches $(\mathsf{qual}, \mathsf{pk}', [\mathsf{cred}']_l)$ in $\mathcal{L}_{\mathsf{pk}'}[\mathsf{sid}]$ (see $\mathsf{O}_{\mathsf{pk}'}$ below) and $([\mathsf{ek}]_n, L := \{(i, \mathsf{sh}_i, \mathsf{d}_i)\}_{i\in[n]}, n) := \mathcal{L}[\mathsf{sid}]$ aborting if not found. It executes $\mathsf{sh}' \leftarrow \mathtt{DeriveSKSh}(\mathsf{pp}, \mathsf{dk}_k, \mathsf{sh}_k, \mathsf{cred}'_k)$. The oracle updates $\mathcal{L}_{\mathcal{E}'}[\mathsf{sid}] \leftarrow \mathcal{L}_{\mathcal{E}'}[\mathsf{sid}] \cup \{(k, [\mathsf{cred}']_l)\}$ and returns sh to the adversary.

$\mathsf{O}_{\mathsf{check}}(\mathsf{sid})$: This oracle takes as input a session identifier sid. It fetches $[\mathsf{ek}]_n, L := \{(i, \mathsf{sh}_i, \mathsf{d}_i)\}_{i\in[n]}, n) = \mathcal{L}[\mathsf{sid}]$ or aborts if such an element does not exists. In this session, if one proxy is corrupted ($(\mathsf{sid}, i) \in \mathcal{C}$), more than $n - s$ shares have been exposed or more than k long-term keys have been exposed, the oracle returns 0. Additionally, if a proxy has its long-term key and share for that session exposed, the oracle also returns 0. Otherwise, it returns 1.

Oracle $\mathsf{O}_{\mathsf{KGenProx}}$ allows the adversary to simulate honest and malicious users participating in a static key generation session and identified by variable k. Oracles $\mathsf{O}_{\mathsf{LTKExpose}}$ and $\mathsf{O}_{\mathsf{ShExpose}}$ are similar to the exposure queries defined for 1PVAKA security. The last oracle assumes a session has already been used to derive a public key and public credentials $[\mathsf{cred}']_l$. It exposes the corresponding derived secret share for a selected user (chosen by its index in $\mathcal{L}$). The last oracle checks if a static key generation session can serve as a non-trivial challenge, i.e. the adversary does not know the corresponding secret key.

For brevity we define $\mathsf{O}_{\mathsf{static}} := (\mathsf{O}_{\mathsf{KGenProx}}, \mathsf{O}_{\mathsf{LTKExpose}}, \mathsf{O}_{\mathsf{ShExpose}})$. The following oracles extend those used in the security definitions of ARKG. Sessions can be seen as generalization of static public keys.

$\mathsf{O}_{\mathsf{pk}'}(\mathsf{sid}, t)$: This oracle takes as input a session identifier $\mathsf{sid} \in \mathcal{S}$ and a threshold t. It fetches $([\mathsf{ek}]_n, L := \{(i, \mathsf{sh}_i, \mathsf{d}_i)\}_{i\in[n]}, n) := \mathcal{L}[\mathsf{sid}]$, aborting if not found or malformed. Key generation for session sid is completed along key derivation:

$$\begin{aligned}(\mathsf{qual}, \mathsf{pk}, [\mathsf{f}]_l) &\leftarrow \mathtt{KGenDeleg}(\mathsf{pp}, [\mathsf{ek}]_n, [\mathsf{d}]_n)\\ (\mathsf{pk}^\star, \mathsf{cred}) &\leftarrow \mathtt{DerivePK}(\mathsf{pp}, \mathsf{pk})\\ (\mathsf{pk}', [\mathsf{cred}']_l) &\leftarrow \mathtt{ShareDPK}(\mathsf{pp}, \mathsf{pk}^\star, \mathsf{cred}, t, [\mathsf{ek}]_n, [\mathsf{f}]_l)\end{aligned}$$

or aborts on failure. It updates $\mathcal{L}_{\mathsf{pk}'}[\mathsf{sid}] \leftarrow \mathcal{L}_{\mathsf{pk}'}[\mathsf{sid}] \cup \{(\mathsf{qual}, \mathsf{pk}', [\mathsf{cred}']_l)\}$ and outputs $(\mathsf{pk}', [\mathsf{cred}']_l)$.

$\mathsf{O}^b_{\mathsf{pk}'}(\mathsf{sid}, t)$: This oracle takes as input a session identifier and a threshold t. The oracle aborts if $\mathsf{O}_{\mathsf{check}}(\mathsf{sid}) \neq 1$. Otherwise it executes $(\mathsf{pk}'_0, [\mathsf{cred}']_l) \leftarrow \mathsf{O}_{\mathsf{pk}'}(\mathsf{sid}, t)$, aborts on failure, and samples a fresh public key $\mathsf{pk}'_1 \leftarrow\!\!\$\ \mathbb{P}$. Then it returns $(\mathsf{pk}_b, [\mathsf{cred}']_l)$.

$\mathsf{O}_{\mathsf{sk}'}(\mathsf{sid}, [\mathsf{cred}']_l)$: This oracle takes as input identifiers sid as well as credentials. It executes $\mathsf{sh}_k \leftarrow \mathsf{O}_{\mathsf{Sh'Expose}}(\mathsf{sid}, k, [\mathsf{cred}']_l)$ for $k \in [l]$. The oracle updates $\mathcal{L}_{\mathsf{sk}'}[\mathsf{sid}] \leftarrow \mathcal{L}_{\mathsf{sk}'}[\mathsf{sid}] \cup \{[\mathsf{cred}']_l\}$ and returns $(\mathsf{sh}_k)_{k\in[l]}$.

Informally, oracle $\mathsf{O}_{\mathsf{pk}'}$ corresponds to a group key generation session and a public key derivation using existing shares and dealings identified with sid, some of which might be compromised. Oracle $\mathsf{O}_{\mathsf{sk}'}$ essentially completes such a session and outputs derived shares for and credentials allowing reconstruction of sk'.

PK-unlinkability states that honestly derived public keys are indistinguishable from freshly sampled ones. In our definition of oracle $\mathsf{O}_{\mathsf{pk}'}$, credentials for the requested session are returned along with a derived public key. This is stronger than the definition in [FGK+,FGMN23] and more aligned with [SW,BCF23]. Note that the original ARKG scheme in [FGK+] satisfies this stronger property as its credentials can be similarly simulated.

Definition 12 (PK-unlinkability). *A dARKG scheme provides PK-unlinkability if for any PPT adversary $\mathcal{A}$, the following advantage is negligible in λ:*

$$\mathsf{Adv}^{\mathsf{pku}}_{\mathsf{dARKG},\mathcal{A}}(\lambda, m) = \Pr\left[\mathsf{Exp}^{\mathsf{pku}}_{\mathsf{dARKG},\mathcal{A}}(\lambda, m) = 1\right]$$

where the pku *experiment is defined in Figure 6.*

In the distributed model, SK-secrecy prevents an adversary from producing a key pair $(\mathsf{sk}^\star, \mathsf{pk}^\star)$ and credentials $[\mathsf{cred}^\star]_l$ for a challenge group key pk. Such an attacker deceives a group of proxies into deriving a common secret key sk' or shares for key $\mathsf{pk}^\star$ when it is already in possession of a secret key $\mathsf{sk}^\star$ for $\mathsf{pk}^\star$. Public key $\mathsf{pk}^\star$ and credentials $[\mathsf{cred}^\star]_l$ may or may not have been derived from pk. These scenarios yield four flavors of SK-secrecy as in [FGK+]; namely, mwKS, hwKS, msKS and hsKS corresponding to malicious/honest and weak/strong variants.

Definition 13 (SK-secrecy). *A dARKG scheme provides SK-secrecy if for any PPT adversary $\mathcal{A}$, the following advantage is negligible in λ:*

$$\mathsf{Adv}^{\mathsf{sk\text{-}secrecy}}_{\mathsf{dARKG},\mathcal{A}}(\lambda, m) = \Pr\left[\mathsf{Exp}^{\mathsf{sk\text{-}secrecy}}_{\mathsf{dARKG},\mathcal{A}}(\lambda, m) = 1\right]$$

where the $\mathsf{Exp}^{\mathsf{sk\text{-}secrecy}}_{\mathsf{dARKG},\mathcal{A}}$ experiment is defined in Figure 6.

```
$\mathsf{Exp}^{\mathsf{sk\text{-}secrecy}}_{\mathsf{dARKG},\mathcal{A}}(\lambda, m)$
$\mathsf{pp} \leftarrow \mathtt{Setup}(1^\lambda, 1^m)$
$(\mathsf{sid}, t) \leftarrow \mathcal{A}^{\mathsf{O}_{\mathsf{static}}}(\mathsf{pp})$
if $\neg\mathsf{O}_{\mathsf{check}}(\mathsf{sid})$ then return $0$
$\mathsf{O} := (\mathsf{O}_{\mathsf{pk}'}(\mathsf{sid}, t), \mathsf{O}_{\mathsf{Sh'Expose}}(\mathsf{sid}, \cdot, \cdot), \boxed{\mathsf{O}_{\mathsf{sk}'}(\mathsf{sid}, \cdot)})$
$(\mathsf{sk}^\star, \mathsf{pk}^\star, [\mathsf{cred}^\star]_l) \leftarrow \mathcal{A}^{\mathsf{O}}$
if $|\{k | (k, [\mathsf{cred}^\star]_l) \in \mathcal{L}_{\mathcal{E}'}[\mathsf{sid}]\}| > t$ then return $0$
$\mathcal{V} \leftarrow\!\!\$$ subset of $\mathsf{qual}$ of size $t$
for $i \in \mathcal{V}$ do $\mathsf{sh}'_i \leftarrow \mathtt{DeriveSKSh}(\mathsf{pp}, \mathsf{dk}_i, \mathsf{sh}_i, \mathsf{cred}^*_i)$
$\mathsf{sk}' \leftarrow \mathtt{DeriveSK}\langle[\mathsf{sh}]_t\rangle(\mathsf{pp})$
return $(\mathit{\Delta}.\mathtt{Check}(\mathsf{sk}^\star, \mathsf{pk}^\star) \overset{?}{=} 1) \land (\mathit{\Delta}.\mathtt{Check}(\mathsf{sk}', \mathsf{pk}^\star) \overset{?}{=} 1)$
    $\boxed{\land[\mathsf{cred}^\star]_l \notin \mathcal{L}_{\mathsf{sk}'}[\mathsf{sid}]}$ $\boxed{\land(\mathsf{qual}, \mathsf{pk}^\star, [\mathsf{cred}^\star]_l) \in \mathcal{L}_{\mathsf{pk}'}[\mathsf{sid}]}$
```

```
$\mathsf{Exp}^{\mathsf{pku}}_{\mathsf{dARKG},\mathcal{A}}(\lambda, m)$
$\mathsf{pp} \leftarrow \mathtt{Setup}(1^\lambda, 1^m)$
$\mathsf{sid} \leftarrow \mathcal{A}^{\mathsf{O}_{\mathsf{static}}}(\mathsf{pp})$
$b \leftarrow\!\!\$ \{0, 1\}$
$b' \leftarrow \mathcal{A}^{\mathsf{O}^b_{\mathsf{pk}'}(\mathsf{sid}, \cdot)}$
return $(b \overset{?}{=} b')$
```

Fig. 6: SK-secrecy and PK-unlinkability experiments for dARKG. dashed boxes give strong variants (msKS, hsKS) while dotted boxes give honest variants (hwKS, hsKS).

4.3 Generic construction of dARKG using 1PVAKA

Let $\mathit{\Delta}$ be a blinding AKG with key spaces $\mathbb{S}$ and $\mathbb{P}$. We use a key-encapsulation mechanism $\mathit{\Lambda} = (\mathtt{KGen}, \mathtt{Encaps}, \mathtt{Decaps})$ with output key space X as defined in [BDK+18]. We work under the simplifying assumption that $\mathit{\Lambda}.\mathtt{KGen} = \mathit{\Delta}.\mathtt{KGen}$ which holds for ARKG instantiations presented in [FGK+,FGMN23,FGM]. In the general case, if $\mathit{\Lambda}.\mathtt{KGen}$ outputs asymmetric keys for an AKG scheme with key spaces $\mathbb{S}'$ and $\mathbb{P}'$, one may use the combined blinding AKG with spaces $\mathbb{S} \times \mathbb{S}'$ and $\mathbb{P} \times \mathbb{P}'$. Another approach is to assume that the key pairs output by $\mathtt{KGen}$

and $(\texttt{DerivePK}, \texttt{DeriveSK})$ need not be from the same distribution. This method is used in [SW] and allows $\texttt{DerivePK}$ to take as input two keys: one for λ and one for Δ.

We use a message authentication code $\mathsf{MAC} = (\texttt{Tag}, \texttt{Verify})$ with key space K and Shamir's [Sha79] secret sharing scheme $\texttt{TSS} = (\texttt{Share}, \texttt{Verify}, \texttt{Recon})$. We also require a cryptographic hash function $\mathcal{H} : X \to \{0,1\}^\lambda$ and a key derivation function $\mathsf{KDF} : X \to K$. Lastly, we use a 1PVAKA scheme for Δ denoted Γ.

Definition 14 (Generic construction for dARKG). *Our generic construction of 1PVAKA using the above setting is introduced below. The core algorithms* $\texttt{DerivePK}$*,* $\texttt{ShareDPK}$ *and* $\texttt{DeriveSKSh}$ *are defined in Figure 7.*

Setup: *Generate and output public parameters for Δ, Π,* $\texttt{TSS}$ *as well as a description of $\mathcal{H}$ and* KDF*.*
Long-term key generation: *Generate a key pair $(\mathsf{dk}_\Lambda, \mathsf{ek}_\Lambda)$ for KEM Λ and $(\mathsf{dk}_\Pi, \mathsf{ek}_\Pi)$ for 1PVAKA Π. Output $\mathsf{dk} := (\mathsf{dk}_\Lambda, \mathsf{dk}_\Pi)$ and $\mathsf{ek} := (\mathsf{ek}_\Lambda, \mathsf{ek}_\Pi)$.*
Key generation (proxy): *On input $[\mathsf{ek}]_n$, execute $\Gamma.\texttt{Deal}([\mathsf{ek}_\Pi]_n)$ and output the result $(\mathsf{sh}, \mathsf{d})$.*
Key generation (delegator): *On input $([\mathsf{ek}]_n, [\mathsf{d}]_n)$, for each dealing d_i that verifies under $\Gamma.\texttt{Verify}$, add i to set qual. Then run $\Gamma.\texttt{CombinePK}([\mathsf{d}]_{\mathsf{qual}})$ and return $(\mathsf{qual}, \mathsf{pk}, [\mathsf{f}]_{\mathsf{qual}})$.*
Public key derivation: *See* $\texttt{DerivePK}$ *and* $\texttt{ShareDPK}$ *in Figure 7.*
Secret key share derivation: *See* $\texttt{DeriveSKSh}$ *in Figure 7.*
Secret key derivation: *This protocol takes as input t shares $\mathsf{sh}'_i = (\mathsf{sk}^\star, s_i)$. Algorithm* $\texttt{TSS.Recon}$ *is executed on shares s_i to reconstruct τ'. This randomness is used to derive secret key $\mathsf{sk}' \leftarrow \texttt{BlindSK}(\mathsf{sk}^\star, \tau')$.*

Correctness follows from that of the building blocks present in the construction.

Lemma 3 (SK-secrecy of dARKG). *Assume that the 1PVAKA scheme Γ provides (1,1)-key-secrecy, that* $\texttt{TSS}$ *offers secrecy and that the VE scheme Π is HVZK. The dARKG scheme from Figure 7 provides* hwKS *flavour of SK-secrecy. More precisely, for every adversary $\mathcal{A}$ against the $\mathsf{Exp}^{\mathsf{sk\text{-}secrecy}}_{\mathsf{dARKG},\mathcal{A}}$ experiment, there exist adversary $\mathcal{B} := (\mathcal{B}_1, \mathcal{B}_2, \mathcal{B}_3)$ such that*

$$\mathsf{Adv}^{\mathsf{sk\text{-}secrecy}}_{\mathsf{dARKG},\mathcal{A}}(\lambda, m) \leq q_1 q_2 \left(m^2 \mathsf{Adv}^{\mathsf{hvzk}}_{\mathsf{VE},\mathcal{B}_1}(\lambda) + \mathsf{Adv}^{\mathsf{privacy}}_{\mathsf{TSS},\mathcal{B}_2}(\lambda) + \mathsf{Adv}^{(1,1)\text{-}\mathsf{secrecy}}_{\mathsf{1PVAKA},\mathcal{B}_3}(\lambda, m) \right)$$

where q_1 and q_2 are the number of sessions started in $\mathsf{O}_{\mathsf{static}}$ and $\mathsf{O}_{\mathsf{pk}'}$ respectively.

The term $m^2 \mathsf{Adv}^{\mathsf{hvzk}}_{\mathsf{VE},\mathcal{B}_1}(\lambda)$ above can be dropped if one considers the weaker, original version of $\mathsf{O}_{\mathsf{pk}'}$ which does not return credentials. Here we are in line with the recent work and definitions of [BCF23].

Proof. Let us consider an attacker $\mathcal{A}$ for experiment experiment ks of DARKG. We build an attacker $\mathcal{B}$ for the SK-security experiment of the 1PVAKA scheme using game-hopping.

```
$\mathtt{DerivePK}(\mathsf{pk}, \mathsf{aux})$:
1 : $(\mathsf{ct}, K) \leftarrow \Lambda.\mathtt{Encaps}(\mathsf{pk})$
2 : $\tau \quad \leftarrow \mathcal{H}(K)$
3 : $mk \leftarrow \mathsf{KDF}(K)$
4 : $\mathsf{pk}^\star \leftarrow \Delta.\mathtt{BlindPK}(\mathsf{pk}, \tau)$
5 : $\mu \leftarrow \mathtt{MAC.Tag}(mk, (\mathsf{ct}, \mathsf{aux}))$
6 : $\mathsf{cred} \leftarrow (\mathsf{ct}, \mathsf{aux}, \mu)$
7 : $\mathbf{return}\ (\mathsf{pk}^\star, \mathsf{cred})$
```

```
$\mathtt{ShareDPK}(\mathsf{pk}^\star, \mathsf{cred}, t, [\mathsf{f}]_l)$:
1 : $(\mathsf{ct}', K') \leftarrow \Lambda.\mathtt{Encaps}(\mathsf{pk}^\star)$
2 : $\tau' \quad \leftarrow \mathcal{H}(K')$
3 : $mk' \leftarrow \mathsf{KDF}(K')$
4 : $\mathsf{pk}' \leftarrow \Delta.\mathtt{BlindPK}(\mathsf{pk}^\star, \tau')$
5 : $[\mathsf{s}]_l \leftarrow \mathtt{TSS.Share}(t, l, \tau')$
6 : $[\mathsf{c}]_l \leftarrow (\Pi.\mathtt{Enc}(\mathsf{ek}_{\Pi,i}, \mathsf{s}_i))_{i\in[l]}$
7 : $[\mathsf{m}]_l \leftarrow (\mathsf{cred}, \mathsf{ct}', \mathsf{f}_i, \mathsf{c}_i)$
8 : $[\mu]_l \leftarrow (\mathtt{MAC.Tag}(mk', \mathsf{m}_i))_{i\in[l]}$
9 : $[\mathsf{cred}']_l \leftarrow (\mathsf{m}_i, \mu_i)_{i\in[l]}$
10 : $\mathbf{return}\ (\mathsf{pk}', [\mathsf{cred}']_l)$
```

```
$\mathtt{DeriveSKSh}(\mathsf{dk}, \mathsf{sh}, \mathsf{cred}')$:
1 : $\mathbf{parse}\ \mathsf{cred}' =: (\mathsf{cred}, \mathsf{ct}', \mathsf{f}, \mathsf{c}, \mu')$
2 : $\mathbf{parse}\ \mathsf{cred} =: (\mathsf{ct}, \mathsf{aux}, \mu)$
3 : $K \leftarrow \Lambda.\mathtt{Decaps}(\mathsf{dk}_\Lambda, \mathsf{ct})$
4 : $K' \leftarrow \Lambda.\mathtt{Decaps}(\mathsf{dk}_\Lambda, \mathsf{ct}')$
5 : $mk, mk' \leftarrow \mathsf{KDF}(K), \mathsf{KDF}(K')$
6 : $\tau, \tau' \leftarrow \mathcal{H}(K), \mathcal{H}(K')$
7 : $\mathsf{m}, \mathsf{m}' \leftarrow (\mathsf{ct}, \mathsf{aux}), (\mathsf{cred}, \mathsf{ct}', \mathsf{f}, \mathsf{c})$
8 : $\mathbf{if} \quad \neg\mathtt{MAC.Verify}(mk, \mathsf{m}, \mu)$
9 :     $\vee \neg\mathtt{MAC.Verify}(mk', \mathsf{m}', \mu')$
10 :     $\mathbf{then\ return}\ \perp$
11 : $\mathsf{sk} \leftarrow \Delta.\mathtt{CombineSK}(\mathsf{dk}_\Gamma, \mathsf{sh}, \mathsf{f})$
12 : $\mathsf{sk}^\star \leftarrow \Delta.\mathtt{BlindSK}(\mathsf{sk}, \tau)$
13 : $s \leftarrow \Pi.\mathtt{Dec}(\mathsf{dk}_\Pi, \mathsf{c})$
14 : $\mathsf{sh}' \leftarrow (\mathsf{sk}^\star, s)$
15 : $\mathbf{return}\ \mathsf{sh}'$
```

Fig. 7: Main algorithms of our general dARKG construction. DerivePK is defined as the generalization of [FGK+] in [SW] for compatibility. It can be more efficiently combined with ShareDPK and τ' can be set to τ.

Game$_0(\lambda, m)$: this game is defined by the $\mathsf{Exp}^{\textsf{sk-secrecy}}_{\textsf{dARKG},\mathcal{A}}(\lambda, m)$ experiment for the honest-weak variant, thus

$$\mathsf{Adv}^{\textsf{sk-secrecy}}_{\textsf{dARKG},\mathcal{A}}(\lambda, m) = \Pr[\mathsf{Game}_0(\lambda, m) = 1].$$

Game$_1(\lambda, m)$: this game is defined as $\mathsf{Game}_0(\lambda, m)$ except that during oracle calls to $\mathsf{O}_{\mathsf{pk}'}$ inside $\mathsf{O}^b_{\mathsf{pk}'}$, in algorithm DerivePK on Line 4, we change $\mathsf{pk}^\star \leftarrow \mathtt{BlindPK}(\mathsf{pk}, \tau)$ for $\mathsf{pk}^\star \leftarrow \mathtt{BlindPK}(\mathsf{pk}_0, \tau)$ for some $(\mathsf{sk}_0, \mathsf{pk}_0) \leftarrow \Delta.\mathtt{KGen}(\mathsf{pp})$. For consistency, whenever $\mathcal{A}$ queries $\mathsf{O}_{\mathsf{sk}'}$ with the same session identifier and credentials $[\mathsf{cred}']_l$ such that $[\mathsf{cred}']_l \in \mathcal{L}_{\mathsf{sk}'}[\mathsf{sid}]$, Line 12 of DeriveSKSh in $\mathsf{O}_{\mathsf{Sh'Expose}}$ is replaced with

$$\mathsf{sk}^\star \leftarrow \Delta.\mathtt{BlindSK}(\mathsf{sk}_0, \tau).$$

Since $(\mathsf{sk}_0, \mathsf{pk}_0)$ are sampled from $\Delta.\mathtt{KGen}$, the blinding property makes output key pairs indistinguishable from freshly generated ones. The two games are thus indistinguishable and

$$\Pr[\mathsf{Game}_0(\lambda, m) = 1] = \Pr[\mathsf{Game}_1(\lambda, m) = 1].$$

$\underline{\mathsf{Game}_2(\lambda, m)\text{:}}$ this game is defined as $\mathsf{Game}_1(\lambda, m)$ except that during oracle calls to $\mathsf{O}_{\mathsf{pk}'}$ inside $\mathsf{O}^b_{\mathsf{pk}'}$, in algorithm `ShareDPK` we replace the ciphertexts in Line 6 with outputs from the HVZK simulator of the verifiable encryption scheme Π. The game is indistinguishable from $\mathsf{Game}_1(\lambda, m)$ under the HVZK assumption of Π and the assumption that `TSS` offers privacy. Indeed, since $t-1$ derived secret key shares have been exposed via oracle $\mathsf{O}_{\mathsf{Sh'Expose}}$, the information theoretic security of `TSS` implies that the adversary does not learn any information related to sk'. Since we simulate $tl \leq m^2$ ciphertexts in total, we therefore have

$$|\Pr[\mathsf{Game}_1(\lambda, m) = 1] - \Pr[\mathsf{Game}_2(\lambda, m) = 1]| \leq m^2 \mathsf{Adv}^{\mathsf{hvzk}}_{\mathsf{VE},\mathcal{B}_1}(\lambda) + \mathsf{Adv}^{\mathsf{privacy}}_{\mathsf{TSS},\mathcal{B}_2}(\lambda).$$

We now construct an adversary $\mathcal{B}$ against the $\mathsf{Exp}^{\mathsf{(s,k)\text{-}secrecy}}_{\mathsf{1PVAKA},\mathcal{A}}$ experiment from an adversary $\mathcal{A}$ against $\mathsf{Game}_2(\lambda, m)$. $\mathcal{B}$ guesses a session identifier sid corresponding to the static key generation session that $\mathcal{A}$ will attack. It also guesses an oracle call to $\mathsf{O}_{\mathsf{pk}'}$ that $\mathcal{A}$ will use to create $\mathsf{pk}^\star$. This is possible as the adversary cannot forge $\mathsf{pk}^\star$ or $[\mathsf{cred}^\star]_l$ for the honest variant: see Line 12 of the SK-security game in Figure 6).

According to the definition game $\mathsf{Game}_1(\lambda, m)$, adversary $\mathcal{B}$ can reply to this oracle call using the challenge query $(\mathsf{pk}, [\mathsf{f}]_l)$ from the challenger for the $\mathsf{Exp}^{\mathsf{(s,k)\text{-}secrecy}}_{\mathsf{1PVAKA},\mathcal{A}}$ game. Indeed, both values can be set arbitrarily since pk is indistinguishable from a fresh key and $[\mathsf{f}]_l$ contains simulated ciphertexts, which is the only information that was previously bound to pk.

Adversary $\mathcal{A}$ then returns a secret public key pair $(\mathsf{sk}^\star, \mathsf{pk}^\star)$ among its outputs. Since this is the honest variant of the experiment, $\mathsf{pk}^\star$ has been legitimately created during session sid meaning $\mathsf{sk}^\star$ is a valid challenge answer for the $\mathsf{Exp}^{\mathsf{(s,k)\text{-}secrecy}}_{\mathsf{1PVAKA},\mathcal{A}}$ experiment. By definition $\Delta.\mathtt{Check}(\mathsf{sk}^\star, \mathsf{pk}^\star) = \Delta.\mathtt{Check}(\mathsf{sk}^\star, \mathsf{pk})$ if $\mathcal{A}$ wins $\mathsf{Game}_2(\lambda, m)$. □

Lemma 4 (PK-unlinkability of dARKG). *Assume that Δ is blinding, Π is HVZK-secure and Λ is* ANO-CCA *secure. Then, the dARKG scheme in Figure 7 provides PK-unlinkability. More precisely, for every adversary against the* $\mathsf{Exp}^{\mathsf{pku}}_{\mathsf{dARKG},\mathcal{A}}$ *experiment, there exists an adversary $\mathcal{B} := (\mathcal{B}_1, \mathcal{B}_2)$ such that*

$$\mathsf{Adv}^{\mathsf{pku}}_{\mathsf{dARKG},\mathcal{A}}(\lambda, m) \leq \ q_1 q_2 \left(\mathsf{Adv}^{\mathsf{ANO\text{-}CPA}}_{\mathsf{KEM},\mathcal{B}_1}(\lambda) + m^2 \mathsf{Adv}^{\mathsf{hvzk}}_{\mathsf{VE},\mathcal{B}_2}(\lambda) \right)$$

where q_1 and q_2 are the number of sessions in $\mathsf{O}_{\mathsf{static}}$ and calls to $\mathsf{O}_{\mathsf{pk}'}$.

Proof. The proof follows the one of PK-unlinkability of the original scheme in [FGK+].

$\underline{\mathsf{Game}_0(\lambda, m)\text{:}}$ this game is defined exactly as the $\mathsf{Exp}^{\mathsf{pku}}_{\mathsf{dARKG},\mathcal{A}}$ experiment and thus

$$\mathsf{Adv}^{\mathsf{pku}}_{\mathsf{dARKG},\mathcal{A}}(\lambda, m) = \Pr[\mathsf{Game}_0(\lambda, m) = 1].$$

$\underline{\mathcal{H}^1_i(\lambda, m)\text{:}}$ recursively define a series of hybrid games by $\mathcal{H}^1_0(\lambda, m) = \mathsf{Game}_0(\lambda)$ and $\mathcal{H}^1_i(\lambda, m) = \mathcal{H}^1_{i-1}(\lambda, m)$ with the exception that during the i-th call to $\mathsf{O}^b_{\mathsf{pk}'}$:

– computation of the public key $\mathsf{pk}^\star$ in $\mathtt{DerivePK}$ executed in the internal call to $\mathsf{O}_{\mathsf{pk}'}(\mathsf{sid}, \cdot)$ is replaced by $\Delta.\mathtt{BlindPK}(\mathsf{pk}_{\mathsf{rand}}, \tau)$,
– computation of the secret key $\mathsf{sk}^\star$ in $\mathtt{DerivePK}$ executed in the internal call to $\mathsf{O}_{\mathsf{Sh'Expose}}(\mathsf{sid}, \cdot, \cdot)$ is replaced by $\Delta.\mathtt{BlindSK}(\mathsf{sk}_{\mathsf{rand}}, \tau)$

where $(\mathsf{sk}_{\mathsf{rand}}, \mathsf{pk}_{\mathsf{rand}}) \leftarrow \Delta.\mathtt{KGen}(\mathsf{pp})$ is a fresh key pair. This simulation is perfect owing to the blinding property of Δ. Thus we have

$$\Pr[\mathsf{Game}_0(\lambda, m) = 1] = \Pr[\mathcal{H}^1_q(\lambda, m) = 1].$$

$\underline{\mathcal{H}^2_i(\lambda, m):}$ recursively define another series of hybrid games by $\mathcal{H}^2_0(\lambda, m) = \mathcal{H}^1_q(\lambda, m)$ and $\mathcal{H}^2_i(\lambda, m) = \mathcal{H}^2_{i-1}(\lambda, m)$ where instead of computing (ct, K) as an encapsulation of pk, the oracle uses a fresh key $(\mathsf{sk}_{\mathsf{rand}}, \mathsf{pk}_{\mathsf{rand}}) \leftarrow \Lambda.\mathtt{KGen}(\mathsf{pp})$. Since Λ is $\mathsf{ANO\text{-}CCA}$, the simulation is perfect under the $\mathsf{ANO\text{-}CCA}$ assumption: the credentials in $\mathcal{H}^2_i(\lambda, m)$ are indistinguishable from those in $\mathcal{H}^1_i(\lambda, m)$. Thus,

$$|\Pr[\mathcal{H}^1_{q_2}(\lambda, m) = 1] - \Pr[\mathcal{H}^2_{q_2}(\lambda, m) = 1]| \leq q_2 \mathsf{Adv}^{\mathsf{ANO\text{-}CCA}}_{\mathsf{KEM}, \mathcal{A}}.$$

2 $\underline{\mathcal{H}^3_i(\lambda, m):}$ these games are defined via $\mathcal{H}^3_0(\lambda, m) = \hat{\mathcal{H}}_q(\lambda, m)2$ and $\mathcal{H}^3_i(\lambda, m) = \mathcal{H}^3_{i-1}(\lambda, m)$ except in the i-th oracle call to $\mathsf{O}^b_{\mathsf{pk}'}$, the ciphertexts created using Π in $\mathtt{ShareDPK}$ are simulated using the HVZK property of Π. The capability of an adversary to distinguish the two games is bound by $\mathsf{Adv}^{\mathsf{hvzk}}_{\mathsf{VE}, \mathcal{B}_3}(\lambda)$, thus

$$|\Pr[\mathcal{H}^2_{q_2}(\lambda, m) = 1] - \Pr[\mathcal{H}^3_{q_2}(\lambda, m) = 1]| \leq mq_2 \mathsf{Adv}^{\mathsf{hvzk}}_{\mathsf{VE}, \mathcal{B}_3}(\lambda).$$

$\underline{\mathcal{H}^4_i(\lambda, m):}$ Start this game series as $\mathcal{H}^4_0(\lambda, m) = \mathcal{H}^3_{q_2}(\lambda, m)$. The output of oracle $\mathsf{O}^b_{\mathsf{pk}'}$ contains a public key pk' and credentials $[\mathsf{cred}']_l$. Those values are generated in $\mathtt{ShareDPK}$ using the outputs of $\mathtt{DerivePK}$: public key $\mathsf{pk}^\star$ is now indistinguishable from a freshly generated one by game $\mathcal{H}^1_q(\lambda, m)$ and cred was generated using a new encapsulation key $\mathsf{pk}_{\mathsf{rand}}$. The dealings for session sid are generated honestly which means the ciphertexts in $[\mathsf{m}]_l$ in $\mathtt{ShareDPK}$ can all be simulated. This leads to the definition of $\mathcal{H}^4_i(\lambda, m)$ similarly to $\mathcal{H}^3_i(\lambda, m)$ and since $tm \leq m^2$ ciphertexts have been simulated per call:

$$|\Pr[\mathcal{H}^3_{q_2}(\lambda, m) = 1] - \Pr[\mathcal{H}^4_{q_2}(\lambda, m) = 1]| \leq m^2 q_2 \mathsf{Adv}^{\mathsf{hvzk}}_{\mathsf{VE}, \mathcal{B}_3}(\lambda).$$

$\underline{\mathsf{Game}_1(\lambda, m):}$ Define this game as $\mathcal{H}^4_{q_2}(\lambda, m)$. $[\mathsf{cred}']_l$ is now indistinguishable from credentials generated via a fresh session. Moreover, the blinding factor τ' used in $\mathtt{ShareDPK}$ in $\mathsf{O}^b_{\mathsf{pk}'}$ was generated using public key $\mathsf{pk}^\star$ so pk' is indistinguishable from a freshly generated key. The outputs of $\mathsf{O}_{\mathsf{pk}'}$ in $\mathsf{Game}_1(\lambda, m)$ are thus indistinguishable from freshly generated values and we have

$$\Pr[\mathsf{Game}_1(\lambda, m) = 1] \leq 1/2$$

Finally, the reduction must guess the session identifier sid which is output by the adversary during the first phase. This adds a factor q_1 to the advantage and concludes the proof. □

5 Instantiation for Bilinear Groups and Performance

For a fixed security parameter we consider $\mathcal{G} = (\mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T, g_1, g_2, e, \gamma, q)$, a type-3 bilinear group such that

- $\mathbb{G}_1$, $\mathbb{G}_2$ and $\mathbb{G}_T$ are cyclic groups of order q,
- $\mathbb{G}_1$ (*resp.* $\mathbb{G}_2$) is generated by element g_1 (*resp.* g_2),
- $e : \mathbb{G}_1 \times \mathbb{G}_2 \to \mathbb{G}_T$ is a non-degenerate bilinear pairing,
- $\gamma : \mathbb{G}_2 \to \mathbb{G}_1$ is an isomorphism.

In the above definition, *non-degenerate bilinear pairing* means group homomorphism linear in both components and such that neither $e(g_1, \cdot)$ nor $e(\cdot, g_2)$ are trivial maps. We assume group operations as well as mapping e to be efficiently computable. Additionaly, neither γ nor γ^{-1} is efficiently computable. We also assume that the DDH assumption holds in both $\mathbb{G}_1$ and $\mathbb{G}_2$.

We instantiate 1PVAKA in Figure 8 using additive ElGamal encryption in a similar fashion as Groth's NI-DKG [Gro]. Importantly, when a proxy generates its contribution to the static key generation as a sample $(s, g_1^s) \leftarrow \Delta.\mathtt{KGen}(\mathsf{pp})$, we re-use randomness to encrypt s for the other proxies in the group efficiently. The public key is appended to the ciphertext along with additional verification information to form the dealing.

Encryption in the exponent is necessary for the public verifiability of our ciphertexts. This requires decomposing cleartexts s in a certain base B as $s =: \sum_{0 \leq k \leq M} s_k B^k$ and encrypt each digit s_k separately to ensure that a search of the cleartext space is possible via BSGS or Pollard's Rho algorithm. Assuming searching interval $[0, mB]$ for the discrete logarithm of a given element $g_1^{s_k}$ is efficient, we can actually perform aggregation of ciphertexts up to m times. More formally, the dealings of proxies in a group $[\mathsf{ek}]_n$ contain ciphertexts for a proxy with public key ek of the form $(g_1^{r_i}, \mathsf{ek}^{r_i} g_1^{s_{i,k}}, g_1^{s_i})$ for $k \in [0, M]$ and $i \in [n]$. After verifying those ciphertexts, the delegator aggregates them using the additive property of the scheme into $(g_1^{r_1+\cdots+r_n}, \mathsf{ek}^{r_1+\cdots+r_n} g_1^{(s_{1,k}+\cdots+s_{n,k})})$ which serve as new dealings. Since $n \leq m$, a search for each exponent in the ciphertext reveals the k-th digit of the combined secret key $\mathsf{sk} := \sum_k (s_{1,k} + \cdots + s_{n,k}) B^k$. The corresponding public key is combined by the delegator as $\mathsf{pk} := \prod_i g_1^{s_i}$. We stress that available VE schemes for discrete logarithms [CD,CS,GHM$^+$22] either use encryption in the exponent or more complicated MPC/NIZK techniques. In our case, we take advantage of the bilinear pairing to perform verifications.

For the sake of simplicity, we implicitly assume that during algorithm `Deal`, a user encrypts its secret share s under its own public key. This is obviously unnecessary and actually has an impact on the security of the scheme which goes from $(1, 1)$-sk secure to $(1, 0)$-sk secure. Indeed, in that case, a single long-term key exposure reveals the secret key for a session.

Regarding the choice of B and the resulting complexity of the scheme. Let γ be the largest integer for which computing the discrete logarithm of g^a for $a \in [0, \ldots, 2^\gamma]$ is efficient. For the scheme to remain n-times additively homomorphic for $n \leq m$, the maximum value for B is $2^\gamma / n$. Parameter q is of size roughly

```
$\mathtt{Setup}(1^\lambda, 1^m)$:
    $\mathsf{pp}_\Delta \leftarrow \Delta.\mathtt{Setup}(1^\lambda)$
    $\mathsf{pp} \leftarrow (\mathsf{pp}_\Delta, m, \mathcal{G})$
    return $\mathsf{pp}$
```

```
$\mathtt{KGen}(\mathsf{pp})$:
    $x \xleftarrow{\$} \mathbb{Z}_q$
    $(\mathsf{dk}, \mathsf{ek}) \leftarrow (x, g_1^x)$
    return $(\mathsf{dk}, \mathsf{ek})$
```

```
$\mathtt{Deal}([\mathsf{ek}]_n)$:
    $s \xleftarrow{\$} \mathbb{Z}_q$
    Write $s =: \sum_{0 \leq k \leq M} s_k B^k$
    $r, w \xleftarrow{\$} \mathbb{Z}_q$
    for $i \in [n]$ do
        $\mathsf{c}_i \leftarrow (\mathsf{ek}_i^r \cdot g_1^{s_k})_{k \in [0,M]}$
    $\mathsf{ct}_0 \leftarrow g_1^r$
    $\mathsf{ct}_1 \leftarrow [\mathsf{c}]_n$
    $\mathsf{ct}_2 \leftarrow (g_1^s, g_2^{rw}, g_2^w)$
    return $\mathsf{d} = (\mathsf{ct}_0, \mathsf{ct}_1, \mathsf{ct}_2)$
```

```
$\mathtt{Verify}([\mathsf{ek}]_n, \mathsf{d})$:
    $(R, (\mathsf{ct}_1, \ldots, \mathsf{ct}_n), (S, U, W)) := \mathsf{d}$
    if $\neg(e(R, W) \overset{?}{=} e(g_1, U))$
        then return $0$
    $\beta \leftarrow \sum_{0 \leq k \leq M} B^k$
    for $i \in [n]$ do
        $P_i \leftarrow \prod_{0 \leq k \leq M} e((\mathsf{ct}_i)_k, W)^{B^k}$
        if $\neg(P_i \cdot e(S, W)^{-1} \overset{?}{=} e(ek_i, U)^\beta)$
        then return $0$
    return $1$
```

```
$\mathtt{CombinePK}([\mathsf{d}]_n)$:
    parse $\mathsf{d}_i = (R_i, [\mathsf{c}_i]_n, (S_i, U_i, W_i))$
    parse $\mathsf{c}_{i,j} =: (\mathsf{e}_{i,j,k})_{k \in [0,M]}$
    $\mathsf{pk} \leftarrow \prod_{i \in [n]} S_i$
    for $i \in [n]$ do
        $R'_i \leftarrow \prod_{j \neq i} R_j$
        $(\mathsf{c}'_{i,k})_{k \in [0,M]} \leftarrow (\prod_{j \neq i} c_{j,i,k})_{k \in [0,M]}$
        $\mathsf{f}_i \leftarrow (R'_i, (\mathsf{c}'_{i,1}, \ldots, \mathsf{c}'_{i,k}))$
    return $[\mathsf{f}]_n$
```

```
$\mathtt{CombineSK}(\mathsf{dk}, \mathsf{sh}, \mathsf{f})$:
    parse $\mathsf{f} =: (R, (\mathsf{c}_0, \ldots, \mathsf{c}_k)$
    for $k \in [0, M]$ do :
        $S_k \leftarrow R^{-\mathsf{dk}} \mathsf{c}_k$
        $s_k \leftarrow \log_{g_1}(S_k)$
    $s \leftarrow \sum_{0 \leq k \leq M} s_k B^k$
    return $s$
```

Fig. 8: Instantiation of 1PVAKA in the bilinear pairing setting.

2λ, thus if M represents the number of digits required to represent a member of $\mathbb{Z}_q$, we need $2^{2\lambda} \leq (2^\gamma/n)^M$ and set $M(n) = \lceil 2\lambda/(\gamma \log(2) - \log(n)) \rceil$. Since M grows as $\mathcal{O}(\log(n)^{-1})$, n is essentially negligible next to γ. Encryption in the exponent thus allows for a compression factor of $\mathcal{O}(n)$ during the `DerivePK` algorithm for the cost of a few additional ciphertexts per dealing.

(n, γ)	$(4, 20)$	$(4, 25)$	$(4, 30)$	$(8, 20)$	$(8, 25)$	$(8, 30)$
M (# of group elements)	22	17	13	22	17	13
`KGenProxy` (in s)	1.09	1.23	1.4	2.78	2.81	2.69
`Verify` (in s)	0.8	0.79	0.82	0.98	0.91	0.9
`CombinePK` (in s)	0.0	0.01	0.01	0.03	0.03	0.03
`CombineSK` (in s)	2.93	8.88	46.48	3.22	10.43	43.83

Table 1: Runtime benchmarks for selected combinations of n and γ for 1PVAKA. The second row indicates the number of group elements (M) in a ciphertext. Rows three to six report the average runtime of the algorithms as a mean over ten executions. We observe the trade-off between the size $\mathcal{O}(M)$ of the dealings and the runtime of `CombineSK` which computes M discrete logarithms.

Table 1 summarizes our experimentally obtained benchmarks for 1PVAKA, which dominates the runtime of the dARKG scheme in Figure 7. The Jupyter notebook used for our experiments is publicly available[3]. Measurements were performed on a Dell Latitude 7430 laptop clocked at 4.8GHz with 16GB of RAM. Our implementation is not optimized and the discrete-logarithm bottleneck can be mitigated at the cost of heavier communications. The results are promising and decryption of dealings in `CombineSK` could even be parallelized, decreasing the execution time by a factor of 2 to 16. Our strategy of additively aggregating ciphertexts during `CombinePK` also pays off, saving n more discrete-logarithm searches per ciphertext. We are hopeful that the same technique can be applied to (verifiable) "naturally" homomorphic encryption to bring dealings size to $\mathcal{O}(n)$ and the run time of `CombineSK` to $\mathcal{O}(1)$.

6 Conclusion

In this work we introduced dARKG, a distributed version of ARKG [FGK+] in which originally one delegator could derive a public key pk' for a single proxy who could at a later time derive the corresponding private key sk'. Our dARKG functionality and security properties are designed to be as close to the original as possible but allow reconstruction of sk' by a group of receivers in a threshold manner. Our general dARKG scheme offers distributed security and fault-tolerance to the envisioned applications of ARKG in WebAuthn and blockchain transactions but also enables further applications when t-out-of-n is viewed as an access policy.

[3] https://gitlab.com/rv5MDg/jupyter-notebook-darkg

References

AMN$^+$21. Donald Q. Adams, Hemanta K. Maji, Hai H. Nguyen, Minh L. Nguyen, Anat Paskin-Cherniavsky, Tom Suad, and Mingyuan Wang. Lower bounds for leakage-resilient secret-sharing schemes against probing attacks. In *2021 IEEE ISIT*, 2021.

BCF23. Jacqueline Brendel, Sebastian Clermont, and Marc Fischlin. Post-quantum asynchronous remote key generation for fido2 account recovery. Cryptology ePrint Archive, 2023.

BCH$^+$19. Dirk Balfanz, Alexei Czeskis, Jeff Hodges, J.C. Jones, Michael B. Jones, Akshay Kumar, Angelo Liao, Rolf Lindemann, and Emil Lundberg. Web authentication: An API for accessing public key credentials level 3. Technical report, 2019.

BDK$^+$18. Joppe Bos, Leo Ducas, Eike Kiltz, T Lepoint, Vadim Lyubashevsky, John M. Schanck, Peter Schwabe, Gregor Seiler, and Damien Stehle. CRYSTALS - Kyber: A CCA-Secure Module-Lattice-Based KEM. In *EuroS&P*, 2018.

Bei11. Amos Beimel. Secret-sharing schemes: A survey. In *IWCC*, Lecture Notes in Computer Science. Springer, 2011.

BFG$^+$20. Jacqueline Brendel, Marc Fischlin, Felix Günther, Christian Janson, and Douglas Stebila. Towards post-quantum security for signal's x3dh handshake. In *Selected Areas in Cryptography: 27th International Conference*, 2020.

BZ. Dan Boneh and Mark Zhandry. Multiparty key exchange, efficient traitor tracing, and more from indistinguishability obfuscation. In *CRYPTO 2014*. Springer Berlin Heidelberg.

CD. Jan Camenisch and Ivan Damgård. Verifiable encryption, group encryption, and their applications to separable group signatures and signature sharing schemes. In *ASIACRYPT 2000*.

CL19. Elizabeth C. Crites and Anna Lysyanskaya. Delegatable anonymous credentials from mercurial signatures. In Mitsuru Matsui, editor, *Topics in Cryptology – CT-RSA 2019*, pages 535–555, Cham, 2019. Springer International Publishing.

CS. Jan Camenisch and Victor Shoup. Practical verifiable encryption and decryption of discrete logarithms. In *CRYPTO 2003*.

DLL$^+$18. Léo Ducas, Tancrede Lepoint, Vadim Lyubashevsky, Peter Schwabe, Gregor Seiler, and Damien Stehlé. Crystals–dilithium: Digital signatures from module lattices. 2018.

FGK$^+$. Nick Frymann, Daniel Gardham, Franziskus Kiefer, Emil Lundberg, Mark Manulis, and Dain Nilsson. Asynchronous remote key generation: An analysis of yubico's proposal for w3c webauthn. In *Proceedings of the 2020 ACM SIGSAC*, page 939–954. Association for Computing Machinery.

FGM. N. Frymann, D. Gardham, and M. Manulis. Asynchronous remote key generation for post-quantum cryptosystems from lattices. In *2023 IEEE 8th European Symposium on Security and Privacy (EuroS&P)*, pages 928–941. IEEE Computer Society.

FGM22. Nick Frymann, Daniel Gardham, and Mark Manulis. Unlinkable delegation of webauthn credentials. In Vijayalakshmi Atluri, Roberto Di Pietro, Christian D. Jensen, and Weizhi Meng, editors, *Computer Security – ESORICS 2022*, pages 125–144, Cham, 2022. Springer Nature Switzerland.

FGMN23. Nick Frymann, Daniel Gardham, Mark Manulis, and Hugo Nartz. Generalised asynchronous remote key generation for pairing-based cryptosystems. In *Applied Cryptography and Network Security: 21st International Conference*, page 394–421. Springer-Verlag, 2023.

GHM$^+$22. Kristian Gjøsteen, Thomas Haines, Johannes Müller, Peter Rønne, and Tjerand Silde. Verifiable Decryption in the Head. In *Australasian Conference on Information Security and Privacy*, volume 13494 of *Lecture Notes in Computer Science*, pages 355–374. Springer International Publishing, 2022.

Gro. Jens Groth. Non-interactive distributed key generation and key resharing. Cryptology ePrint Archive, Paper 2021/339.

KGS. Chelsea Komlo, Ian Goldberg, and Douglas Stebila. A formal treatment of distributed key generation, and new constructions. Cryptology ePrint Archive, Paper 2023/292.

KWZ22. Venkata Koppula, Brent Waters, and Mark Zhandry. Adaptive multiparty NIKE. In *TCC 2022*, Berlin, Heidelberg, 2022. Springer-Verlag.

MBG$^+$23. Omid Mir, Balthazar Bauer, Scott Griffy, Anna Lysyanskaya, and Daniel Slamanig. 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*, page 30–44. Association for Computing Machinery, 2023.

PTDH. Sihang Pu, Sri AravindaKrishnan Thyagarajan, Nico Döttling, and Lucjan Hanzlik. Post quantum fuzzy stealth signatures and applications. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*, page 371–385.

Rao. Vanishree Rao. Adaptive multiparty non-interactive key exchange without setup in the standard model. Cryptology ePrint Archive, Paper 2014/910.

Sha79. Adi Shamir. How to share a secret. 22(11):612–613, 1979.

SW. Douglas Stebila and Spencer Wilson. Quantum-safe account recovery for webauthn. Cryptology ePrint Archive, Paper 2024/678.

WMS$^+$. Qianhong Wu, Yi Mu, Willy Susilo, Bo Qin, and Josep Domingo-Ferrer. Asymmetric group key agreement. In Antoine Joux, editor, *EUROCRYPT 2009*, pages 153–170. Springer Berlin Heidelberg.

Yu. Gary Yu. Blockchain stealth address schemes. Cryptology ePrint Archive, Paper 2020/548.

A Asynchronous Remote Key Generation

An ARKG scheme is composed of five algorithms ($\texttt{Setup}$, $\texttt{KGen}$, $\texttt{DerivePK}$, $\texttt{DeriveSK}$, $\texttt{Check}$) defined as follows:

$\texttt{Setup}(1^\lambda)$: The randomized setup algorithm takes as input a security parameter 1^λ and outputs a description pp of the public parameters.

$\texttt{KGen}(\mathsf{pp})$: The randomized key generation algorithm takes as input public parameters pp and outputs a private-public keypair $(\mathsf{sk}, \mathsf{pk})$.

$\texttt{DerivePK}(\mathsf{pp}, \mathsf{pk}, \mathsf{aux})$: The randomized public key derivation algorithm takes as input public parameters pp, a public key pk and auxiliary information aux. It probabilistically returns a public key pk' together with a link cred between pk and pk'.

$\texttt{DeriveSK}(\mathsf{pp}, \mathsf{sk}, \mathsf{cred})$: The secret-key derivation algorithm takes as input public parameters pp, a secret key sk and credential cred. It either outputs the secret key sk' corresponding to pk' or $\perp$ on error.

$\texttt{Check}(\mathsf{pp}, \mathsf{sk}', \mathsf{pk}')$: The check algorithm takes as input public parameters pp, a secret key sk' and a public key pk'. It returns 1 if the keypair $(\mathsf{sk}', \mathsf{pk}')$ is valid, otherwise 0.

An ARKG scheme is said to be correct if and only if for all $\lambda \in \mathbb{N}$, it holds that

$$\Pr\left[\begin{array}{c} \mathsf{pp} \leftarrow \texttt{Setup}(1^\lambda),\ (\mathsf{sk}, \mathsf{pk}) \leftarrow \texttt{KGen}(\mathsf{pp}), \\ (\mathsf{pk}', \mathsf{cred}) \leftarrow \texttt{DerivePK}(\mathsf{pp}, \mathsf{pk}, \cdot), \\ \mathsf{sk}' \leftarrow \texttt{DeriveSK}(\mathsf{pp}, \mathsf{sk}, \mathsf{cred}) : \texttt{Check}(\mathsf{pp}, \mathsf{sk}', \mathsf{pk}') \neq 1 \end{array}\right] = \mathsf{negl}(\lambda).$$

Security properties of ARKG. An adversary $\mathcal{A}$ is modelled as a probabilistic polynomial time (PPT) algorithm. The oracles to which the adversary has access in the security experiments are introduced below.

$\mathsf{O}_{\mathsf{pk}'}(\mathsf{pk}, \mathsf{aux})$: this oracle is parameterized with a public key pk and takes as optional input aux. It outputs the result of $\texttt{DerivePK}(\mathsf{pp}, \mathsf{pk}, \mathsf{aux})$. The couple $(\mathsf{pk}', \mathsf{cred})$ is stored a list $\mathcal{L}_{\mathsf{pk}'}$.

$\mathsf{O}^b_{\mathsf{pk}'}(b, \mathsf{sk}_0, \mathsf{pk}_0)$: this oracle is parameterized by a keypair $(\mathsf{sk}_0, \mathsf{pk}_0)$ and bit b. It outputs $(\mathsf{sk}', \mathsf{pk}')$ derived using key pair $(\mathsf{sk}_0, \mathsf{pk}_0)$ when $b = 0$ or a freshly generated key pair if $b = 1$.

$\mathsf{O}_{\mathsf{sk}'}(\mathsf{sk}, \mathsf{cred})$: this oracle parameterized by a secret key sk takes a credential cred as input. It outputs the results of $\texttt{DeriveSK}(\mathsf{pp}, \mathsf{sk}, \mathsf{cred})$ if $(\cdot, \mathsf{cred}) \in \mathcal{L}_{\mathsf{pk}'}$, otherwise $\perp$. The results are stored as cred in a list $\mathcal{L}_{\mathsf{sk}'}$ initially set.

PK-unlinkability: An ARKG scheme provides PK-unlinkability if the following advantage is negligible in λ:

$$\mathsf{Adv}^{\mathsf{pku}}_{\mathsf{ARKG},\mathcal{A}}(\lambda) = \left|\Pr\left[\mathsf{Exp}^{\mathsf{ks}}_{\mathsf{ARKG},\mathcal{A}}(\lambda) = 1\right] - \frac{1}{2}\right|$$

where the pku experiment is defined in Figure 9.

SK-secrecy: An ARKG scheme provides SK-secrecy if the following advantage is negligible in λ:

$$\mathsf{Adv}^{\mathsf{ks}}_{\mathsf{ARKG},\mathcal{A}}(\lambda) = \Pr\left[\mathsf{Exp}^{\mathsf{ks}}_{\mathsf{ARKG},\mathcal{A}}(\lambda) = 1\right]$$

where the ks experiment is defined in Figure 9.

```
$\mathsf{Exp}^{\mathsf{ks}}_{\mathsf{ARKG},\mathcal{A}}(\lambda)$
$\mathsf{pp} \leftarrow \mathtt{Setup}$
$(\mathsf{pk}_0, \mathsf{sk}_0) \leftarrow \mathtt{KGen}(\mathsf{pp})$
$b \leftarrow\$ \{0,1\}$
$b' \leftarrow \mathcal{A}^{\mathsf{O}^b_{\mathsf{pk}'}}(\mathsf{pp}, \mathsf{pk}_0)$
$\mathbf{return}\ b \overset{?}{=} b'$
```

```
$\mathsf{Exp}^{\mathsf{ks}}_{\mathsf{ARKG},\mathcal{A}}(\lambda)$
$\mathsf{pp} \leftarrow \mathtt{Setup}$
$(\mathsf{sk}, \mathsf{pk}) \leftarrow \mathtt{KGen}(\mathsf{pp})$
$(\mathsf{sk}^\star, \mathsf{pk}^\star, \mathsf{cred}^\star) \leftarrow \mathcal{A}^{\mathsf{O}_{\mathsf{pk}'}, \boxed{\mathsf{O}_{\mathsf{sk}'}(\mathsf{sid}, \cdot)}}(\mathsf{pp}, \mathsf{pk})$
$\mathsf{sk}' \leftarrow \mathtt{DeriveSK}(\mathsf{pp}, \mathsf{sk}, \mathsf{cred}^\star)$
$\mathbf{return}\ \mathtt{Check}(\mathsf{pp}, \mathsf{sk}^\star, \mathsf{pk}^\star) \overset{?}{=} 1$
    $\wedge\, \mathtt{Check}(\mathsf{pp}, \mathsf{sk}', \mathsf{pk}^\star) \overset{?}{=} 1$
    $\boxed{\wedge \mathsf{cred}^\star \notin \mathcal{L}_{\mathsf{sk}'}}$
    $\boxed{\wedge (\mathsf{pk}^\star, \mathsf{cred}^\star) \in \mathcal{L}_{\mathsf{pk}'}}$
```

Fig. 9: Security experiments for PK-unlinkability on the left and SK-secrecy on the right. dashed boxes give strong variants (msKS, hsKS) of the ks security experiment while dotted boxes give honest variants (hwKS, hsKS).

B Additional Building Blocks

B.1 Key Derivation Functions

Definition 15 (Key Derivation Function). *A key derivation function* $\mathsf{KDF} : K \times L \to K'$ *takes as input a key* k *and a label* l*. It outputs a new key* k' *not necessarily from the same key space as* k*.*

noindent A KDF is secure if the following advantage is negligible in λ for all PPT adversary $\mathcal{A}$

$$\mathsf{Adv}^{\mathsf{KDF}}_{\mathcal{A}}(\lambda) = \left|\Pr\left[\mathsf{Exp}^{\mathsf{IND}}_{\mathsf{KDF},\mathcal{A}}(\lambda) = 1\right] - \frac{1}{2}\right|,$$

where the indistinguishability IND experiment is defined in figure 10. In the following we will fix once and for all a KDF function $\mathsf{KDF} : \mathbb{P} \times L \to \mathbb{S}$, two labels l_1, l_2 and consider $\mathsf{KDF}_1 = \mathsf{KDF}(\cdot, l_1)$ and $\mathsf{KDF}_2 = \mathsf{KDF}(\cdot, l_2)$

B.2 Message Authentication Code

A Message Authentication Code (MAC) is a triple $\mathsf{MAC} = (\mathtt{KGen}, \mathtt{Tag}, \mathtt{Verify})$. Algorithm $\mathtt{KGen}(1^\lambda)$ takes as input a security parameter 1^λ and outputs a secret key $mk \leftarrow\$ \{0,1\}^\lambda$, $\mathtt{Tag}(mk, m)$ outputs a tag μ for a secret key mk and a message m and $\mathtt{Verify}(mk, m, \mu)$ outputs 1 if the tag μ is valid for mk and m, otherwise 0. A MAC scheme is said to be correct if and only if for every $\lambda \in \mathbb{N}, m \in \{0,1\}^*$,

$$(mk \leftarrow \mathtt{KGen}(1^\lambda);\ \mu \leftarrow \mathtt{Tag}(mk, m)) \Rightarrow \mathtt{Verify}(mk, m, \mu),$$

Define oracle $\mathsf{O}_{\mathsf{Tag}}(mk, \cdot)$ parameterized by a key mk and taking a message m as input. It returns the result of $\mathtt{Tag}(mk, m)$. A MAC is unforgeable if the following advantage is negligible in λ for all PPT adversary $\mathcal{A}$

$$\mathsf{Adv}^{\mathsf{UNF}}_{\mathsf{MAC},\mathcal{A}}(\lambda) = \Pr\left[\mathsf{Exp}^{\mathsf{UNF}}_{\mathsf{MAC},\mathcal{A}}(\lambda) = 1\right],$$

where the unforgeability UNF experiment is defined in Figure 10.

```
$\mathsf{Exp}^{\mathsf{IND}}_{\mathsf{KDF},\mathcal{A}}(\lambda)$
    $k \leftarrow\$ K$
    $l \leftarrow\$ L$
    $y_0 \leftarrow \mathsf{KDF}(k, l)$
    $y_1 \leftarrow\$ \{0,1\}^\lambda$
    $b \leftarrow\$ \{0,1\}$
    $b' \rightarrow \mathcal{A}(y_b)$
    return $b' \overset{?}{=} b$
```

```
$\mathsf{Exp}^{\mathsf{UNF}}_{\mathsf{MAC},\mathcal{A}}(\lambda)$
    $mk \leftarrow \mathtt{KGen}(1^\lambda)$
    $(m^\star, \mu^\star) \leftarrow \mathcal{A}^{\mathsf{O}_{\mathsf{Tag}}(mk,\cdot)}$
    return $m \neq m^\star$
      $\wedge\ \mathtt{Verify}(mk, m^\star, \mu^\star)$
```

Fig. 10: Security experiments for KDF and MAC.