Secure Multiparty SpendAuthSig for ZCash

Ariel Nof*

April 4, 2019

Abstract

SpendAuthSig is a Schnorr based signature used in Zcash to sign transactions of currency. In this paper, we describe an efficient protocol for threshold signing. Our protocol can be used to protect private signing keys by distributing them among several parties, who upon request can run the threshold protocol to generate a signature.

1 Introduction

In this paper, we describe an efficient protocol to produce a SpendAuthSig signature, which is used in Zcash to sign on transactions of money, in a distributed way. In particular, this means that the secret signing key will be shared among several parties, which will run a secure computation of the signature algorithm in order to sign a message. This can protect from signing keys theft, as breaching several devices is harder than hacking into one central device. Our protocol for performing this distributed computation is proven secure by showing that it securely computes a standard ideal functionality for signing, and relying on the hardness of the DDH problem. SpendAuthSig is a Schnor-based signature [15] with re-randomization of keys as defined in [2] and some ideas from EdDSA [6]. Thus, we will follow the guidelines of the existing solutions for threshold Schnor signatures with the appropriate adaptations.

The documnet is organized as follows. We begin by describing the signing and key generation algorithm. Then, we define the ideal functionality and give the security definition. This is followed by sub protocols that are used in the main protocol. Finally, we present our distributed protocol to sign messages by first describing and proving a protocol for 2 parties and then show how to extend it to any number of parties with any threshold.

Notation. Let $\mathbb{G}$ be an Elliptic curve group of order q with generator G where the DDH assumption is assumed to be hard. We will use addition as the group operation, upper-case characters for group elements, and lower-case characters for scalars in $\mathbb{Z}_q$. This is consistent with Elliptic curve notation, although all of our protocols work equivalently in finite field groups. Let $H : \{0,1\}^* \to \{0,1\}^\ell$, be a cryptographic hash function and let $F : \{0,1\}^{\ell+128} \times \{0,1\}^* \to \{0,1\}^\ell$ be a pseudo-random function[1], where ℓ is the security parameter.

*Dept. of Computer Science, Bar-Ilan University, Israel. ariel.nof@biu.ac.il. Some of this work was carried out for Kzen Ltd.

[1] In practice, this is also instantiated using a cryptographic hash function. Specifically, $F_k(m) = H(k||m)$.

The SpendAuthSig signature. We now describe the signing and the key generation algoorithms. Given a message m and the signing key ask the signing algorithm does the following:

1. Choose a random $\alpha \in \mathbb{G}$.
2. Let $rsk = ask + \alpha \bmod q$ and let $vk = rsk \cdot G \bmod q$.
3. Choose a random $T \in \{0,1\}^{\ell+128}$.
4. Compute: $r = F_T(vk||m) \bmod q$ [2].
5. Compute: $R = r \cdot G$.
6. Compute: $S = r + H(R||vk||m) \cdot rsk \bmod q$.
7. Output (R, S).

In the key generation step, which is executed once, a random $ask \in \{0,1\}^{\ell}$ is chosen to be the private signing key and $ak = sk \cdot G$ is the public verification key[3].

2 Security Definition

2.1 The SpendAuthSig Ideal Functionality

We show how to securely compute the functionality $\mathcal{F}_{\text{SIGN}}$. The functionality is defined with two functions: key generation and signing. The key generation is called once, and then any arbitrary number of signing operations can be carried out with the generated key. The functionality is defined in Figure 2.1.

[2] As mentioned above, in practice F is instantiated using a cryptographic hash function. However, the property required here is pseudo-randomness and so we feel that using a PRF for the description is more accurate. The use of hash function as a PRF is discussed and justified in [2].

[3] In Zcash, ask is computed by first choosing a random sk and then computing $ask = F_{sk}(0)$ where F is a pseudo random function. However, this is just a way to protect the randomness generation mechanism and it makes no difference for the distributed protocol.

FUNCTIONALITY 2.1 (The ECDSA Functionality $\mathcal{F}_{\text{SIGN}}$)

Functionality $\mathcal{F}_{\text{SIGN}}$ works with parties $P_1, \ldots, P_n$, as follows:

- Upon receiving $\mathsf{KeyGen}(\mathbb{G}, G, q)$ from all parties $P_1, \ldots, P_n$, where $\mathbb{G}$ is an Elliptic-curve group of order q with generator G:
  1. Generate a key pair (ak, ask) by choosing a random $ask \leftarrow \mathbb{Z}_q^*$ and computing $ak = ask \cdot G$. Then, store $(\mathbb{G}, G, ak, ask)$.
  2. Send ak to all $P_1, \ldots, P_n$.
  3. Ignore future calls to KeyGen.
- Upon receiving $\mathsf{Sign}(sid, m)$ from all $P_1, \ldots, P_n$, if KeyGen was already called and sid has not been previously used, compute an $\mathsf{SpendAuthSig}$ signature on m in the following way:
  1. Choose a random $\alpha \in \mathbb{G}$ and set $rsk = ask + \alpha$ and $vk = rsk \cdot G$.
  2. Choose a random $r \in \mathbb{Z}_q$ and set $R = r \cdot G$.
  3. Compute: $S = r + H(R||vk||m) \cdot rsk$.

  Send (R, S) and α to all $P_1, \ldots, P_n$.

2.2 Security Model

Security in the presence of malicious adversaries. We prove security according to the standard simulation paradigm with the real/ideal model [3, 9], in the presence of *malicious adversaries* and *static corruptions*. As is standard for the case of no honest majority, we consider security with abort meaning that a corrupted party can learn output while the honest party does not. In our definition of functionalities, we describe the instructions of the trusted party. Since we consider security with abort, the ideal-model adversary receives output first and then sends either $(\mathsf{continue}, j)$ or (abort, j) to the trusted party, for every $j \in [n]$ to instruct the trusted party to either deliver the output to party P_j (in case of $\mathsf{continue}$) or to send abort to party P_j. This means that honest parties either receive the correct output or abort, but some honest parties may receive output while others abort. This was termed *non-unanimous abort* in [10]. As described at the end of Section 4, in this case of secure signing, it is easy to transform the protocol so that all parties receive output if any single honest party received output.

We remark that when all of the zero-knowledge proofs are UC secure [4], then our protocol can also be proven secure in this framework.

Security, the hybrid model and composition. We prove the security of our protocol in a hybrid model with ideal functionalities that securely compute $\mathcal{F}_{\mathsf{com}}, \mathcal{F}_{\mathsf{zk}}, \mathcal{F}_{\mathsf{com\text{-}zk}}, \mathcal{F}_{\text{coin}}$, defined next in Section 3. The soundness of working in this model is justified in [3] (for stand-alone security) and in [4] (for security under composition). Specifically, as long as subprotocols that securely compute the functionalities are used (under the definition of [3] or [4], respectively), it is guaranteed that the output of the honest and corrupted parties when using real subprotocols is indistinguishable to when calling a trusted party that computes the ideal functionalities.

3 Sub protocols and Building Blocks

In this section, we define the ideal functionalities we call in our main protocol and show how to securely realize them. As in [13], [5], we prove the security of our protocol using the ideal zero-knowledge functionality, denoted $\mathcal{F}_{\mathsf{zk}}$, and an ideal commit-and-prove functionality, denoted $\mathcal{F}_{\mathsf{com\text{-}zk}}$. In practice, these proofs are instantiated using Fiat-Shamir on highly efficient Sigma protocols. In addition, we define an ideal functionality $\mathcal{F}_{\text{coin}}$ which provide fresh public randomness upon request.

The ideal zero knowledge functionality $\mathcal{F}_{\mathsf{zk}}$. We use the standard ideal zero-knowledge functionality defined by $((x, w), \lambda) \to (\lambda, (x, R(x, w)))$, where λ denotes the empty string. For a relation R, the functionality is denoted by $\mathcal{F}_{\mathsf{zk}}^{R}$. Note that any zero-knowledge proof of knowledge fulfills the $\mathcal{F}_{\mathsf{zk}}$ functionality [11, Section 6.5.3]; non-interactive versions can be achieved in the random-oracle model via the Fiat-Shamir paradigm; see Functionality 3.1 for the formal definition.

FUNCTIONALITY 3.1 (The Zero-Knowledge Functionality $\mathcal{F}_{\mathsf{zk}}^{R}$ for Relation R)

Upon receiving $(\mathsf{prove}, sid, i, x, w)$ from a party P_i (for $i \in [n]$): if sid has been previously used then ignore the message. Otherwise, send $(\mathsf{proof}, sid, i, x, R(x, w))$ to all parties $P_1, \ldots, P_n$, where $R(x, w) = 1$ iff $(x, w) \in R$.

We define the following two standard relations for which knowledge is required to being proved in our protocols:

1. *Knowledge of the discrete log of an Elliptic-curve point:* Define the relation

$$R_{DL} = \{(\mathbb{G}, G, q, \mathcal{P}, w) \mid \mathcal{P} = w \cdot G\}$$

of discrete log values (relative to the given group). We use the standard Schnorr proof for this [15]. The cost of this proof is one exponentiation for the prover and two for the verifier, and communication cost of two elements of $\mathbb{Z}_q$.

2. *Diffie-Hellman tuple of Elliptic-curve points:* Define the relation

$$R_{DH} = \{(\mathbb{G}, G, q, (A, B, C), w) \mid B = w \cdot G \ \wedge \ C = w \cdot A\}$$

of Diffie-Hellman tuples (relative to the given group). We use the well-known proof that is an extension of Schnorr's proof for discrete log. The cost of this proof is two exponentiations for the prover and four for the verifier, and communication cost of two elements of $\mathbb{Z}_q$.

For clarity, we remove the $(\mathbb{G}, G, q)$ part from the input to the zero-knowledge proofs below, with the understanding that these parameters are fixed throughout.

For completeness, we provide a full description of non-interactive zero-knowledge for these relations in Appendix A.

The ideal commitment functionality $\mathcal{F}_{\mathsf{com}}$. In order to realize $\mathcal{F}_{\mathsf{com\text{-}zk}}$ and $\mathcal{F}_{\text{coin}}$ defined below, we use an ideal commitment functionality $\mathcal{F}_{\mathsf{com}}$, formally defined in Functionality 3.2.

FUNCTIONALITY 3.2 (The Commitment Functionality $\mathcal{F}_{\mathsf{com}}$)

Functionality $\mathcal{F}_{\mathsf{com}}$ works with parties $P_1, \ldots, P_n$, as follows:

- Upon receiving $(\mathsf{commit}, sid, i, x)$ from party P_i (for $i \in [n]$), record (sid, i, x) and send $(\mathsf{receipt}, sid, i)$ to all $P_1, \ldots, P_n$. If some $(\mathsf{commit}, sid, i, *)$ is already stored, then ignore the message.
- Upon receiving $(\mathsf{decommit}, sid, i)$ from party P_i, if (sid, i, x) is recorded then send $(\mathsf{decommit}, sid, i, x)$ to party $P_1, \ldots, P_n$.

Since $\mathcal{F}_{\mathsf{com}}$ is defined so that all parties receive the same commitment, the value $\mathsf{Com}(x)$ needs to be broadcasted. However, as shown in [10], a simple echo-broadcast suffices here for the case of non-unanimous abort (this takes two rounds of communication). In order to ensure this, the parties send a hash of all the commitments that they received in the round after the commitments were sent. If any two parties received different commitments, then they notify all parties to abort and then halt. This adds very little complexity and ensures the same view for any committed values.

Any UC-secure commitment scheme fulfills $\mathcal{F}_{\mathsf{com}}$; e.g., [12, 1, 8]. We next describe two simple ways realize $\mathcal{F}_{\mathsf{com}}$.

COMMITMENT BASED ON DDH. Since we already works over groups where the DDH assumption is assumed to be hard, we can use this to construct a commitment. This is formalized in Protocol 3.3.

PROTOCOL 3.3 (Securely computing $\mathcal{F}_{\mathsf{com}}$ in the $(\mathcal{F}_{\mathsf{zk}}^{DL}, \mathcal{F}_{\mathsf{zk}}^{DH})$-Hybrid Model)

- **Input:** Party P_i holds x
- **Auxiliary input:** Each party holds the same El Gamal public key Q.
- **Commit:**
  1. Party P_i chooses random $\rho \in \mathbb{Z}_q$ and computes $U = \rho \cdot G$ and $V = \rho \cdot Q + x$. Then, it sends $(sid, i, [n], U, \rho)$ with $sid = com$ to $\mathcal{F}_{\mathsf{zk}}^{DL}$ and V to all P_j with $j \in [n]$.
  2. Upon receiving (sid, j, U, β) from $\mathcal{F}_{\mathsf{zk}}^{DL}$ and V from P_i, each party P_j $(j \neq i)$ aborts if any sid is incorrect or $\beta_j = 0$ for some j. Otherwise, it outputs (U, V).
- **De-Commit:**
  1. Each party P_i sends x to all P_j for $j \in [n]$ and $(sid, i, [n], (Q, U, V - x), \rho)$ with $sid = decom$ to $\mathcal{F}_{\mathsf{zk}}^{DH}$.
  2. Upon receiving x and $(sid, j, (Q, A, B), \beta)$, each party P_j $(j \neq i)$ verifies that $sid = decom$, $\beta = 1$, $A = U$ and $B + x = V$. If not, then it aborts. If yes, then it outputs x.

COMMITMENT IN THE RANDOM-ORACLE MODEL. In the random-oracle model, $\mathcal{F}_{\mathsf{com}}$ can be trivially realized with static security by simply defining $\mathsf{Com}(x) = H(x, r)$ where $r \leftarrow \{0,1\}^{\ell}$ is random, and sending $\mathsf{Com}(x)$ to all parties.

The committed non-interactive zero knowledge functionality $\mathcal{F}_{\mathsf{com\text{-}zk}}$. In our protocol, we will have parties send commitments to a statement together with a non-interactive zero-knowledge proof of the statement. As in [13], we model this formally via a commit-zk functionality, denoted $\mathcal{F}_{\mathsf{com\text{-}zk}}$, defined in Functionality 3.4. Given non-interactive zero-knowledge proofs of knowledge, this functionality is securely realized by just having the prover commit to the statement together

with its proof, using the ideal commitment functionality $\mathcal{F}_{\mathsf{com}}$. As in $\mathcal{F}_{\mathsf{com}}$, consistency of views is validated by all parties sending a hash of the commitments that they received.

FUNCTIONALITY 3.4 (The Committed NIZK Functionality $\mathcal{F}^R_{\mathsf{com\text{-}zk}}$ for Relation R)

Functionality $\mathcal{F}_{\mathsf{com\text{-}zk}}$ works with parties $P_1, \ldots, P_n$, as follows:

- Upon receiving $(\mathsf{ComProve}, sid, i, x, w)$ from a party P_i (for $i \in [n]$): if sid has been previously used then ignore the message. Otherwise, store (sid, i, x) and send $(\mathsf{ProofReceipt}, sid, i)$ to $P_1, \ldots, P_n$.
- Upon receiving $(\mathsf{DecomProof}, sid)$ from a party P_i (for $i \in [n]$): if (sid, i, x) has been stored then send $(\mathsf{DecomProof}, sid, i, x, R(x, w))$ to $P_1, \ldots, P_n$.

The secure coin-tossing functionality $\mathcal{F}_{\mathrm{coin}}$. We will need a secure coin-tossing protocol to generate public randomness (specifically, this will be used to generate the random $\alpha \in \mathbb{Z}_q$ in the signing protocol). Our secure coin-tossing protocol realizes the functionality $\mathcal{F}_{\mathrm{coin}}(sid, \ldots, sid) = (r, \ldots, r)$ where $r \in \mathbb{Z}_q$.

The protocol to securely realize this functionality works by having each party P_i choose independently its random coin $r_i \in \mathbb{Z}_q$ and sends a commitment to this value to the other parties (more formally, sends (sid, i, r_i) to $\mathcal{F}_{\mathsf{com}}$ with $sid = coin$). Upon receiving a commitment from all the parties (receiving $(\mathsf{receipt}, sid, j)$ with $sid = coin$ for all $j \in [n]$), each party P_i opens its commitment towards the other parties (sends $(\mathsf{decommit}, sid, i)$ to $\mathcal{F}_{\mathsf{com}}$). The output is set to be $r = \sum_{i=1}^{n} r_i$. One can realize the commitment functionality in each of the ways described above.

4 2-out-of-2 Threshold Signing

In this section, we describe a distributed key generation and signing protocols for 2 parties.

PROTOCOL 4.1 (Securely Computing $\mathcal{F}_{\text{SIGN}}$)

Auxiliary input: Each party has the description $(\mathbb{G}, G, q)$ of a group.

Key generation: Upon input $\mathsf{KeyGen}(\mathbb{G}, G, q)$, each party P_i works as follows:

1. Each party P_i chooses a random $ask_i \in \mathbb{Z}_q$.
2. Each party P_i computes $ak_i = ask_i \cdot G$ and sends $(\mathsf{ComProve}, sid, 3-i, (G, ak_i), sk_i)$ to $\mathcal{F}_{\mathsf{com\text{-}zk}}^{DL}$ with $sid = keygen$.
3. Upon receiving $(\mathsf{ProofReceipt}, sid, 3-i)$ from $\mathcal{F}_{\mathsf{com\text{-}zk}}^{DL}$, party P_i sends $(\mathsf{DecomProof}, keygen)$ to $\mathcal{F}_{\mathsf{com\text{-}zk}}^{DL}$.
4. Upon receiving $(\mathsf{DecomProof}, sid, 3-i, ak_i, \beta)$ from $\mathcal{F}_{\mathsf{com\text{-}zk}}^{DL}$, party P_i sends aborts if sid is incorrect or $\beta = 0$. Otherwise, it proceeds to the next step.
5. The parties set: $ak = ak_1 + ak_2$.
6. *Output:* P_i locally stores ak as the $\mathsf{SpendAuthSig}$ public-key.

Signing: Upon input $\mathsf{Sign}(sid, m)$, where sid is a unique session id sid, each party P_i works as follows:

1. The parties call $\mathcal{F}_{\text{coin}}$ to receive a random $\alpha \in \mathbb{Z}_q$.
2. Each party locally computes $vk = ak + \alpha \cdot G$.
3. Each party P_i chooses a random $T_i \in \{0,1\}^{\ell+128}$ and computes $r_i = F_{T_i}(vk||m)$
4. Each party P_i computes $R_i = r_i \cdot G$ and sends $(\mathsf{ComProve}, sid, 3-i, (G, R_i), r_i)$ to $\mathcal{F}_{\mathsf{com\text{-}zk}}^{DL}$ functionality with $sid = sign$.
5. Upon receiving $(\mathsf{ProofReceipt}, sid, 3-i)$ from $\mathcal{F}_{\mathsf{com\text{-}zk}}^{DL}$ with $sid = sign$, party P_i sends $(\mathsf{DecomProof}, sid)$ to $\mathcal{F}_{\mathsf{com\text{-}zk}}^{DL}$.
6. Upon receiving $(\mathsf{DecomProof}, sid, 3-i, R_i, \beta)$ from $\mathcal{F}_{\mathsf{com\text{-}zk}}^{DL}$, each party P_i aborts if sid is incorrect or $\beta = 0$. Otherwise, it proceeds to the next step.
7. The parties define $R = R_1 + R_2$.
8. Party P_1 locally computes $S_1 = r_1 + H(R||vk||m) \cdot (ask_1 + \alpha)$ and sends it to P_2.
   Party P_2 locally computes $S_2 = r_2 + H(R||vk||m) \cdot ask_2$ and sends it to P_1.
9. Upon receiving S_{3-i} from P_{3-i}, each party P_i sets: $S = S_1 + S_2$ and checks that $\mathsf{vrfy}_{vk}(m, (R, S)) = 1$. if the equality holds, then it outputs (S, R). Otherwise, it aborts.

Theorem 4.2 *If F is a pseudo-random function, then Protocol 4.1 securely computes $\mathcal{F}_{\text{SIGN}}$ with abort in the $\mathcal{F}_{\mathsf{zk}}^{DL}, \mathcal{F}_{\text{coin}}$-hybrid model, in the presence of a malicious party.*

Proof: Let $\mathcal{A}$ be an adversary who controls one of the parties and assume without loss of generality that the P_1 is the corrupted party (since the protocol is symmetric the proof when P_2 is corrupted is identical). We construct a simulator $\mathcal{S}$ who invokes $\mathcal{A}$ internally and simulates an execution of the real protocol, while interacting with $\mathcal{F}_{\text{SIGN}}$ in the ideal model. We remark that in the simulation the adversary always receives the message from the honest party simulated by $\mathcal{S}$ at the beginning of each round before it sends the message from the corrupted party. This is due to the fact that according to the security definition the adversary is rushing.

Key generation: For the key generation, $\mathcal{S}$ sends $\mathsf{KeyGen}(\mathbb{G}, G, q)$ to $\mathcal{F}_{\text{SIGN}}$. Upon receiving back ak from $\mathcal{F}_{\text{SIGN}}$, simulator $\mathcal{S}$ works as follows:

1. $\mathcal{S}$ simulates $\mathcal{F}_{\mathsf{com\text{-}zk}}^{DL}$ handing $\mathcal{A}$ the message $(\mathsf{ProofReceipt}, keygen, 2)$.
2. Upon receiving the $(\mathsf{ComProve}, sid, 1, (G, ak_1), sk_1)$ from $\mathcal{A}$, the simulator $\mathcal{S}$ checks that $sid = keygen$ and that $ak_1 = sk_1 \cdot G$. If not, it sends abort to $\mathcal{F}_{\text{SIGN}}$, simulates the honest parties aborting in the real execution and outputs what ever $\mathcal{A}$ outputs.
3. $\mathcal{S}$ computes $ak_2 = ak - ak_1$ and simulates $\mathcal{F}_{\mathsf{com\text{-}zk}}^{DL}$ handing $\mathcal{A}$ the message $(\mathsf{DecomProof}, keygen, 2, ak_2, 1)$.
4. Upon receiving the message $(\mathsf{DecomProof}, keygen)$ from $\mathcal{A}$, the simulator $\mathcal{S}$ stores sk_1, as well as ak.

This completes the simulation of the key generation phase.

Signing: For the signing phase, upon input $\mathsf{Sign}(sid, m)$, if sid has not been used previously, then $\mathcal{S}$ sends $\mathsf{Sign}(sid, m)$ to $\mathcal{F}_{\text{SIGN}}$. Upon receiving back a signature (S, R) and a random α, simulator $\mathcal{S}$ works as follows:

1. $\mathcal{S}$ simulates $\mathcal{F}_{\text{coin}}$ handing α to $\mathcal{A}$.
2. $\mathcal{S}$ defines $vk = ak + \alpha \cdot G$.
3. $\mathcal{S}$ simulates $\mathcal{F}_{\mathsf{com\text{-}zk}}^{DL}$ handing $\mathcal{A}$ the message $(\mathsf{ProofReceipt}, sid, 2)$.
4. Upon receiving the message $(\mathsf{ComProve}, sid, 1, (G, R_1), r_1)$ sent by $\mathcal{A}$, the simulator checks that sid is correct and $R_1 = r_1 \cdot G$. If not, it sends abort to $\mathcal{F}_{\text{SIGN}}$, simulates the honest parties aborting in the real execution and outputs whatever $\mathcal{A}$ outputs.
5. $\mathcal{S}$ defines $R_2 = R - R_1$ and simulates $\mathcal{F}_{\mathsf{com\text{-}zk}}^{DL}$ handing $\mathcal{A}$ the message $(\mathsf{DecomProof}, sid, 2, R_2, 1)$.
6. Upon receiving the message $(\mathsf{DecomProof}, sid)$ from $\mathcal{A}$, the simulator $\mathcal{S}$ computes $S_1 = r_1 + H(R||vk||m) \cdot (ask_1 + \alpha)$[4] and defines $S_2 = S - S_1$. Then, it sends S_2 to $\mathcal{A}$.
7. Upon receiving S_1' from $\mathcal{A}$, the simulator $\mathcal{S}$ checks that $S_1' = S_1$. If not, it sends abort to $\mathcal{F}_{\text{SIGN}}$, simulates the honest parties aborting in the real world execution. Otherwise, it sends $\mathsf{continue}$ to $\mathcal{F}_{\text{SIGN}}$. Finally, $\mathcal{S}$ outputs whatever $\mathcal{A}$ outputs.

The only difference between the simulated and real execution is the way R is computed. Specifically, in the real execution $R = (F_{T_1}(vk||m) + F_{T_2}(vk||m)) \cdot G$ whereas in the ideal execution $R = r \cdot G$ where r is distributed uniformly over $\mathbb{Z}_q$. However, since F is a pseudo-random function, this makes no difference. We prove this by showing that if the view of the adversary in the real execution can be distinguished from its view in the ideal execution with more than a negligible probability, than we can construct a distinguisher D that can distinguish between a random and a pseudo-random function with the same probability.

In particular, consider a distinguisher D who receives an oracle $\mathcal{O}$ which is either a random or a pseudo-random function. $\mathcal{D}$ invokes $\mathcal{A}$ to run an execution where $\mathcal{D}$ plays the role of the honest

[4]Note that here if P_2 is corrupted $\mathcal{S}$ does not add α. This is the only difference in the simulation between the two cases.

party and follows the instructions of the protocol with one exception: instead of choosing T_2 and computes $F_{T_2}(vk||m)$, it sends $vk||m$ to his oracle and use the answer as r_2.

It is immediate that when $\mathcal{O}$ is a pseudo-random function, the execution is identical to a real execution. We claim also that when $\mathcal{O}$ is a random function the obtained transcript is identical to an ideal execution with the simulator $\mathcal{S}$. To see this, observe that in the execution with $\mathcal{D}$, r_2 is a random string and r_1 is pseudo-random, and so $R = (r_1 + r_2) \cdot G$ is a uniformly distributed over $\mathbb{G}$, exactly as in the ideal world execution. This completes the proof. ■

Output to all parties. As we have described above, our protocol as described is not secure with unanimous abort, since some honest parties may abort while others receive output. However, in this case of ECDSA signing, it is easy to transform the protocol so that if one honest party receives output then so do all. This is achieved by having any party who receives (r, s) as output send it to all other parties. Then, if a party who otherwise aborted receives (r, s), it can check that (r, s) is a valid signature on m and output it if yes.

5 Extending the Protocol to Any Number of Parties and Any Threshold

In the previous section, we described a protocol for 2 parties. In this section, we show how the protocol can be extended to any number of parties with different number of corrupted parties. We begin with the setting where there are n parties and $n - 1$ are assumed to be corrupted and then move to the general setting where any $t < n$ parties are corrupted. This setting in particular enables us to obtain a signing scheme where any subset of $t+1$ parties can sign a message. Having this property is important since one cannot expect all the parties to be available each time currency need to be transfered.

n-out-of-n multi-party protocol. Extending our protocol to this setting is straightforward. In the key generation, each party chooses its share ask_i randomly and broadcasts $ak_i = ask_i \cdot G$ to all the other parties using the $\mathcal{F}_{\mathsf{com\text{-}zk}}^{DL}$ ideal functionality. The public-key is defined to be $ak = \sum_{i=1}^{n} ask_i$.

In the signing protocol, the parties call $\mathcal{F}_{\text{coin}}$ to generate α. Then, each party P_i produces r_i as described in Protocol 4.1 and broadcasts $R_i = r_i \cdot G$ using $\mathcal{F}_{\mathsf{com\text{-}zk}}^{DL}$. Holding $R = \sum_{i=1}^{n} R_i$, each party P_i can compute S_i as described in Protocol 4.1, where P_1 adds α to its share ask_1, and send it to all the other parties. Finally, the parties define $S = \sum_{i=1}^{n} S_i$ and run the verification procedure to verify the correctness of the signature.

t-out-of-n multi-party protocol. For this setting, we can use the Shamir's secret sharing scheme. Recall that in this scheme, the dealer shares a secret s by choosing a random t-degree polynomial $p()$ over Z_q such that $p(x) = s + a_1 \cdot x + a_2 \cdot x^2 + \cdots + a_t x^t$ and handing party P_i the share $p(i)$. A problem that may arise is that a malicious dealer may not use a polynomial with the correct degree. To overcome this, we can use the Feldman's VSS (verifiable secret sharing) scheme where the dealer sends also the vector $(s \cdot G, a_1 \cdot G, \ldots, a_t \cdot G)$ to all the parties. Upon receiving its share α_i and the vector $(A_0, A_1, \ldots, A_t)$, each party P_i checks that $\alpha_i \cdot G = \prod_{j=0}^{t} A_j^{i^j}$. If the equality does not hold, it aborts.

Incorporating this into our protocol, the key generation protocol is changed such that each party also shares its private key using the VSS scheme. Once each party P_i holds the public key $ak_j = ask_j \cdot G$, a share α_i^j and a vector $(A_1^j, \ldots, A_t^j)$ for each $j \in [n]$, it checks that the shares were dealt correctly using the above check. Then, it defines $ak = \sum_{j=1}^{n} ak_j$ to be the public key, $\alpha_i = \sum_{j=1}^{n} \alpha_i^j$ to be its share of the public key and $(A_1, \ldots, A_t) = (\sum_{j=1}^{n} A_1^j, \ldots, \sum_{j=1}^{n} A_t^j)$ to be the commitment to the coefficient of the t-degree polynomial $p()$ such that $p(0) = ask$.

To sign a message m, a subset of $t+1$ parties can use our signing protocol by translating their sharing into an additive sharing. This can be easily done by having each party multiplying its share with the appropriate Lagrange coefficient.

Refreshing the keys. To further strengthen the security it is possible to provide partial proactive security by having the parties refreshing their shares periodically. This is done by generating a random sharing of 0 and adding it to the current sharing. After the update, the parties should erase the previous shares of the private key and the shares of 0 they held. The security property guaranteed by this process is that if an adversary corrupted t parties *before* the update and t parties *after* the update, nothing is learned even if these are not the same subset of parties.

6 Verified Cold Back-up

In order to provide a strong custody solution that will be used to store very large amounts of currency, it is absolutely essential that a cold backup key-pair be generated to ensure that money is not irretrievable lost. The public key can be used to encrypt the SpendAuthSig private key, and the private key can be stored in disconnected HSMs stored in bunkers at different locations around the world. This is not trivial since just asking each participant to separately encrypt their share x_i of the private key under the cold-backup public key is not sufficient. This is because a malicious party can encrypt an incorrect value, rendering the backup solution useless. We therefore require the parties to prove that they indeed encrypted the correct value. If the cold backup key can be of any type required, then we can use Paillier and invoke the zero-knowledge proof of PDL constructed in [13]. Given a Paillier ciphertext c and a group element ak_i, this proof verifies that c encrypts some value $ask_i \in \mathbb{Z}_q$ such that $ask_i \cdot G = ak_i$. This is a good solution, except for the fact that in practice, the cold backup key would likely be stored in an HSM that does not support Paillier. Thus, it is desirable to provide a solution that works with RSA or ElGamal. Since neither of these schemes are additively homomorphic, this is very challenging.

We present a solution from [14] that requires the cold backup to store many ciphertexts for each share. However, when considering a high-end custody solution (e.g., for protecting cryptocurrency held by investment funds), this is a reasonable cost.

Let $P_1, \ldots, P_n$ be the set of parties who participate in cold backup key generation. Each party works in the same way, and so we describe the instructions for party P_i. All parties work non-interactively and send their result to a server S_0. All of the values and proofs are stored, and since the proofs are non-interactive, they are publicly verifiable. Let G denote the generator point of the Elliptic curve group, and let q denote its order.

Point and proof generation: Party P_i receives the RSA public key pk and works as follows:

1. P_i chooses a random $ask_i \leftarrow \mathbb{Z}_q$, and computes $ak_i = ask_i \cdot G$.

2. For $j = 1, \ldots, 128$, P_i works as follows:

(a) P_i chooses random $r_j \leftarrow \mathbb{Z}_q$ and $s_{j,0}, s_{j,1} \leftarrow \{0,1\}^\kappa$ (where κ is the length of the random string in RSA-OAEP).

(b) P_i computes $y_j = ask_i + r_j \bmod q$.

(c) P_i computes $c_{j,0} = \mathsf{Enc}_{pk}(r_j, s_{j,0})$ and $c_{j,1} = \mathsf{Enc}_{pk}(y_j, s_{j,1})$.

(d) P_i computes $\hat{Q}_j = r_j \cdot G$.

3. P_i computes $e \leftarrow SHA256(ak_i, c_{1,0}, c_{1,1}, \hat{Q}_1, \ldots, c_{128,0}, c_{128,1}, \hat{Q}_{128})$, truncated to 128 bits. Denote $e_i = e_i^1, \ldots, e_i^{128}$.

4. For $j = 1, \ldots, 128$, P_i works as follows:

(a) If $e_i^j = 0$ then define $z_i^j = (r_j, s_{j,0}, c_{j,1})$.

(b) If $e_i^j = 1$ then define $z_i^j = (y_j, s_{j,1}, c_{j,0})$.

5. Output Q_i and $\pi_i \leftarrow (e, z_i^1, \ldots, z_i^{128})$.

Proof verification: Server S_0 receives Q_i and $\pi_i \leftarrow (e_i, z_i^1, \ldots, z_i^{128})$ and works as follows:

1. For $j = 1, \ldots, 128$, S_0 works as follows:

(a) If $e_i^j = 0$ then define $c_{j,0} = \mathsf{Enc}_{pk}(r_j, s_{j,0})$ and $\hat{Q}_j = r_j \cdot G$.

(b) If $e_i^j = 1$ then define $c_{j,1} = \mathsf{Enc}_{pk}(y_j, s_{j,1})$ and $\hat{Q}_j = y_j \cdot G - ak_i$.

2. Compute $e_i' \leftarrow SHA256(Q_i, c_{1,0}, c_{1,1}, \hat{Q}_1, \ldots, c_{128,0}, c_{128,1}, \hat{Q}_{128})$, truncated to 128 bits, and accept if and only if $e_i' = e_i$.

Public-key generation: If the verification of the proofs from all P_i is successful, then compute $ak = \sum_{i=1}^n ak_i$ and verify that it equals the public-key ak (if not, then the key generation has failed and currency should not be transferred to this key). S_0 stores $(Q_1, \pi_1, \ldots, Q_n, \pi_n)$.

Private-key reconstruction: Given the RSA private key sk, and the stored values $(Q_1, \pi_1, \ldots, Q_n, \pi_n)$, the server S_0 works as follows, for every $i = 1, \ldots, n$:

1. For $j = 1, \ldots, 128$:

(a) If $e_i^j = 0$ then compute $y_j \leftarrow \mathsf{Dec}_{sk}(c_{j,1})$ and $x_i^j \leftarrow y_j - r_j \bmod q$. If $ak_i = x_i^j \cdot G$ then set $x_i \leftarrow x_i^j$ and break from the loop (for j).

(b) If $e_i^j = 1$ then compute $r_j \leftarrow \mathsf{Dec}_{sk}(c_{j,0})$ and $x_i^j \leftarrow y_j - r_j \bmod q$. If $ak_i = x_i^j \cdot G$ then set $ask_i \leftarrow x_i^j$ and break from the loop (for j).

If $ak_i \neq x_i^j \cdot G$ for all $j = 1, \ldots, 128$, then output fail and halt.

2. Let $ask_1, \ldots, ask_n$ be the values obtained. Then, compute $ask = \sum_{i=1}^n ask_i \bmod q$ and verify that $ak = ask \cdot G$. If yes,output the private $\mathsf{SpendAuthSig}$ key ask and halt.

We remark that in practice, all parties would send their values and proofs to multiple servers, to ensure that no server S_0 can tamper with the values (or just erase them).

Security. We first prove that for every i, the probability that the proof verification succeeds and the private-key reconstruction fails is negligible. Specifically, fix z_i^j and assume that $y_j - r_j \neq ask_i$, where $ak_i = ask_i \cdot G$, $c_{j,0} = \mathsf{Enc}_{pk}(r_j; s_{j,0})$ and $c_{j,1} = \mathsf{Enc}_{pk}(y_j; s_{j,1})$. If verification succeeds for both $e_i^j = 0$ and $e_i^j = 1$, then this implies that $\hat{Q}_j = r_j \cdot G$ and $\hat{Q}_j = y_j \cdot G - ak_i$. (Note that equality is not checked explicitly but is checked via verifying that $e_i = e_i'$.) This implies that $r_j \cdot G = y_j \cdot G - ak_i$ and so $ak_i = (y_j - r_j) \cdot G$, implying that $ask_i = y_j - r_j \bmod q$. The above implies that for every j, the probability that the proof passes if $y_i - r_j \neq ask_i \bmod q$ is $1/2$. Applying the Fiat-Shamir paradigm, we have the result.

For zero-knowledge the protocol can be simulated, implying that the $\mathsf{SpendAuthSig}$ private key remains private even given the proof. In order to simulate, we assume the random oracle model, and so the simulator knows e ahead of time. In this case, it can simply choose r_j or y_j at random (and encrypt 0 for the other ciphertext). Since $y_j = ask_i + r_j \bmod q$ where r_j is random, both r_j and y_j in isolation are distributed uniformly. Thus, the simulation is indistinguishable (with the only difference being the encryptions to zeroes).

7 Incorporating the Protocol into Sapling

In this section, we discuss the integration of our threshold signing protocol inside the sapling protocol.

In Sapling, there is a master key sk from which all the keys are derived. First, three private keys are generated: ask, nsk and ovk in the following way:

$$ask = \mathsf{PRF}_{sk}(0), \qquad nsk = \mathsf{PRF}_{sk}(1), \qquad ovk = \mathsf{PRF}_{sk}(2)$$

These three keys together are the "expanded spending key".

As we have seen, the spending authorization key is $ak = ask \cdot G$. The full viewing key is the triple ak, nk and ovk where nk is derived from nsk by setting $nk = nsk \cdot \mathcal{H}$ where $\mathcal{H}$ is generated deterministically by a taking a GroupHash over a constant string. Then, the incoming view key is $ivk = \mathsf{CRHF}(ak, nk)$. Finally, the transmission key pk_d is computed by choosing a random diversifier d, hashing it to obtain g_d and setting $pk_d = ivk \cdot g_d$.

TO BE CONTINUED...

References

[1] O. Blazy, C. Chevalier, D. Pointcheval and D. Vergnaud. Analysis and Improvement of Lindell's UC-Secure Commitment Schemes. In *ACNS 2013*, Springer (LNCS 7954), pages 534–551, 2013.

[2] D.J. Bernstein, N. Duif, T. Lange, P. Schwabe, B.Y. Yang: High-speed high-security signatures. J. Cryptographic Engineering 2(2): 77-89 (2012)

[3] R. Canetti. Security and Composition of Multiparty Cryptographic Protocols. *Journal of Cryptology*, 13(1):143–202, 2000.

[4] R. Canetti. Universally Composable Security: A New Paradigm for Cryptographic Protocols. In 42*nd FOCS*, pages 136–145, 2001. Full version available at `http://eprint.iacr.org/2000/067`.

[5] J. Doerner, Y. Kondi, E. Lee and a. shelat. Secure Two-party Threshold ECDSA from ECDSA Assumptions In the 39*th IEEE Symposium on Security and Privacy*, 2018.

[6] N. Fleischhacker, J. Krupp, G. Malavolta, J. Schneider, D. Schröder, M. Simkin: Efficient Unlinkable Sanitizable Signatures from Signatures with Re-randomizable Keys. Public Key Cryptography (1) 2016: 301-330

[7] A. Fiat and A. Shamir: How to Prove Yourself: Practical Solutions to Identification and Signature Problems. In *CRYPTO 1986*, Springer (LNCS 263), pages 186–194, 1986.

[8] E. Fujisaki. Improving Practical UC-Secure Commitments Based on the DDH Assumption. In *SCN 2016*, Springer (LNCS 9841), pages 257–272, 2016.

[9] O. Goldreich. *Foundations of Cryptography: Volume 2 – Basic Applications*. Cambridge University Press, 2004.

[10] S. Goldwasser and Y. Lindell. Secure Computation Without Agreement. *Journal of Cryptology,* 18(3):247–287, 2005.

[11] C. Hazay and Y. Lindell. *Efficient Secure Two-Party Protocols: Techniques and Constructions*. Springer, November 2010.

[12] Y. Lindell. Highly-Efficient Universally-Composable Commitments Based on the DDH Assumption. In *EUROCRYPT 2011*, Springer (LNCS 6632), pages 446–466, 2011.

[13] Y. Lindell. Fast Secure Two-Party ECDSA Signing. In *CRYPTO 2017*, Springer (LNCS 10402), pages 613–644, 2017.

[14] Y. Lindell and A. Nof. Fast Secure Multiparty ECDSA with Practical Distributed Key Generation and Applications to Cryptocurrency Custody. In the 25th ACM CCS 2018, pages 1837-1854, 2018.

[15] C.P. Schnorr. Efficient Identification and Signatures for Smart Cards. In *CRYPTO 1989*, Springer (LNCS 435), pages 239–252, 1990.

A Zero-Knowledge Proofs

In this section, We give a full specification of the two zero-knowledge proofs used in our protocol. In the description, we make use of the Fiat-Shamir technique to obtain an non-interactive proof.

A.1 $\mathcal{F}_{\mathsf{zk}}^{DL}$ - Protocol for proving knowledge of an EC discrete logarithm

In this protocol, the prover P and the verifier V have a common pair (G, Q) and the prover has a witness w such that $Q = w \cdot G$.

1. **Round 1:**

   (a) P chooses a random $\sigma \in \mathbb{Z}_q$ and computes $X = \sigma \cdot G$.

   (b) P computes: $e = H(G||Q||X||sid)$ (where sid is the session identifier).

(c) P computes $u = \sigma + e \cdot w \bmod q$.

(d) P sends the proof $\pi = (e, u)$ to V.

2. **Round 2:**

(a) V parses the proof π as (e, u).

(b) V computes $X = u \cdot G - e \cdot Q$.

(c) V outputs 1 if $e = H(G||Q||X||sid)$. Otherwise, it outputs $\perp$ and aborts.

A.2 $\mathcal{F}_{\mathsf{zk}}^{DH}$ - Protocol for proving that an EC tuple is a DH tuple

In this protocol, the prover P and the verifier V have a common tuple (G, A, B, C) and the prover has a witness w such that $B = w \cdot G$ and $C = w \cdot A$.

1. **Round 1:**

(a) P chooses a random $\sigma \in \mathbb{Z}_q$ and computes $X = \sigma \cdot G$ and $Y = \sigma \cdot A$.

(b) P computes: $e = H(G||A||B||C||X||Y||sid)$ (where sid is the session identifier).

(c) P computes $u = \sigma + e \cdot w \bmod q$.

(d) P sends the proof $\pi = (e, u)$ to V.

2. **Round 2:**

(a) V parses the proof π as (e, u).

(b) V computes $X = u \cdot G - e \cdot B$ and $Y = u \cdot A - e \cdot C$.

(c) V outputs 1 if $e = H(G||A||B||C||X||Y||sid)$. Otherwise, it outputs $\perp$ and aborts.