Compact and Divisible E-Cash with Threshold Issuance

Alfredo Rial
Nym Technologies
alfredo@nymtech.net

Ania M. Piotrowska
Nym Technologies
ania@nymtech.net

ABSTRACT

Decentralized, offline, and privacy-preserving e-cash could fulfil the need for both scalable and byzantine fault-resistant payment systems. Existing offline anonymous e-cash schemes are unsuitable for distributed environments due to a central bank. We construct a distributed offline anonymous e-cash scheme, in which the role of the bank is performed by a quorum of authorities, and present its two instantiations. Our first scheme is compact, i.e. the cost of the issuance protocol and the size of a wallet are independent of the number of coins issued, but the cost of payment grows linearly with the number of coins spent. Our second scheme is divisible and thus the cost of payments is also independent of the number of coins spent, but the verification of deposits is more costly. We provide formal security proof of both schemes and compare the efficiency of their implementations.

KEYWORDS

offline anonymous e-cash, threshold issuance, bilinear maps

1 INTRODUCTION

At the present moment, there is a pressing need for private electronic cash (e-cash), as shown by growing interest in blockchain-based systems and Centrally-Banked Digital Currencies (CBDCs), but no privacy-enhanced and decentralized system exists that scales to the requirements of this application scenario. To address this problem, we introduce the first decentralized offline e-cash scheme with provable security and full implementation that can efficiently support electronic payments. In contrast to the current privacy-enhanced blockchain, systems do not require a constant online presence of the payees. The issuance of coins is non-interactive, i.e., the authorities do not need to synchronise, as we do not rely on MPC protocols. Moreover, our scheme maps to distributed payment systems such as CBDCs, a technology that urgently requires further attention in terms of privacy. The idea of distributing issuance among a quorum of authorities has been so far explored in the context of attribute-based credentials [50, 54] and online anonymous e-cash schemes [5].

Contributions: Our paper makes the following contributions:

- We introduce a construction Π_{EC} for threshold issuance offline e-cash (EC). To the best of our knowledge, this is the first offline e-cash scheme with threshold issuance. To this end, we define the system model and the security properties in the ideal-world/real-world paradigm [21] by proposing an ideal functionality $\mathcal{F}_{\text{EC}}$ for EC (Sections §3, §4).
- We propose two instantiations of Π_{EC} based on compact [14] and divisible [49] e-cash (Section §5). Our schemes are more efficient than [14, 49] thanks to the use of Pointcheval-Sanders signatures in the random oracle model and to decreasing the number of coin secrets. We formally prove that our construction Π_{EC} realizes $\mathcal{F}_{\text{EC}}$ when instantiated with the algorithms of our compact and divisible e-cash schemes (Sections §D, §E).
- We provide an open-source Rust implementation of both schemes and present an extensive evaluation of their performance and trade-offs (Section §6). To the best of our knowledge, this is the first such practical comparison.

2 BACKGROUND AND MOTIVATION

Anonymous e-cash was originally proposed by Chaum as a digital analog of regular cash, which also allows for private payments [22]. Unlike physical cash, e-coins are easy to duplicate, hence e-cash schemes must prevent double-spending and typically that is done by having a centralized bank [2, 6, 7, 10, 14, 18–20, 45, 46, 49]. Thus, there has been a revival of interest in adopting privacy to decentralized blockchain systems, in which coins are authenticated by proving in ZK that they belong to a public list of valid coins maintained on the blockchain, thus they do not require a central bank to prevent double-spending [43].

Although blockchain-based privacy-enhanced systems such as ZCash and Monero have users [44, 51, 55], such decentralized e-cash systems require being online to check the status of payments, which simply does not scale to the speed of transactions needed by real-world payment systems or support the reality of transactions that need to be made without internet access, so the usage of blockchains for payments remains small in proportion to traditional payments. Also, as could happen to any other low-transaction fee blockchain that advertises high throughput, the ZCash blockchain has suffered an attack of *'spam'* transactions that have increased the blockchain size to such an extent that ZCash has suffered from what is effectively a *denial-of-service* attack that has collapsed its throughput (i.e., simply downloading the blockchain becomes nearly impossible) [25]. Similarly, 'layer 2' solutions based on blockchain technologies such as zero-knowledge roll-ups, which increase transaction speed and (in some cases) privacy, are also vulnerable to these attacks [59]. By virtue of not requiring a merchant to be online all the time but only needing eventual settlement over regular epochs, our system avoids these issues while maintaining the advantages of decentralization. Hence, it is to be expected that current privacy-enhanced blockchain systems that require an online setting will evolve into decentralized offline e-cash systems similar to the one presented in this paper. While proposals exist to enable offline payments in blockchain-based cryptocurrencies such as payment channel networks [33], they typically do not offer strong privacy protection as only users who share a channel can transact with each other and so users who make a payment are not anonymous,

Proceedings on Privacy Enhancing Technologies 2023(4), 381–415

https://doi.org/10.56553/popets-2023-0116

similarly as in [27], and cashing out payments still requires online blockchain interactions. Even privacy-preserving offline payment channels effectively restrict payment transactions and the network topology of payment channels can lead participants to be effectively de-anoymized [37, 53].

On the other side of the spectrum, centralized CBDCs have gained increasing interest in over a hundred countries and will soon be deployed in Europe and China [4, 57]. CBDC systems typically sacrifice user transaction privacy from the settlement layer and can lead to the possibility of dystopian surveillance of user financial transactions [26]. Although financial transaction data could be considered a matter of national security, MIT and the Federal Reserve of Massachusetts in the United States have begun exploring blockchain systems for its CBDC efforts without transaction unlinkability [41]. In stark contrast, Switzerland's Central Bank has put forward a centralized online privacy-preserving scheme called 'eCash 2.0', but offline payments would require special hardware and are currently not supported. The European Central Bank (ECB) recently published a list of requirements for the Digital Euro, which include privacy of user transactions from the settlement layer and offline transactions [4]. Our scheme would fulfil those criteria, and we present how it could be integrated with a blockchain. Furthermore, distributing the issuing power would be practical for emerging multi-nation economic proposals for joint issuance of currencies such as the South American joint reserve currency SUR in the 'Banco del Sur' recently endorsed by Brazil and Argentina [42]. Even more importantly, practically preventing byzantine faults in centralized CBDCs requires it to be managed by a set of distributed parties, similar to Facebook's Diem project's consensus protocol based on HotStuff [58]. In this manner, our system unifies the objectives of blockchain research for decentralization while maintaining compatibility with the requirements for privacy-preserving CBDC by decentralizing e-cash.

Online vs Offline Ecash. To revisit the original e-cash proposal, the solution to double-spending is to associate each coin with a unique *serial number*, which is used to detect double-spending by dishonest users and prevent dishonest providers from depositing a payment twice [22]. In *online e-cash* schemes [22, 54], providers are constantly connected to the bank and can then check if a coin has been double-spent before accepting a payment. An alternative and more realistic solution space is given by *offline e-cash* schemes [7, 10, 14, 18–20, 49], which do not require a permanent online connection between a provider and the bank. The provider can accept payment and deposit it at a later settlement stage as there is a guarantee that users who double-spend will be identified by the bank.

An important issue in the design of anonymous e-cash schemes is paying the exact amount as users cannot receive change, since the providers are not anonymous towards the bank. If a user receives change, the user would in fact become a provider and lose anonymity. In online e-cash, the user can contact the bank in order to exchange a coin for lower denominations in order to make the payment. In offline e-cash, contacting the bank is not allowed during the spending phase. *Transferable e-cash* schemes [2, 24, 45] are one solution to this problem, allowing a user to further spent a previously received coin without interacting with the bank. Hence, providers can return the change to the users that paid. Although transferable e-cash is appealing, state-of-the-art schemes [6] are much less efficient than non-transferable ones.

An alternative solution to preserve user unlinkability is to use coins of the smallest denomination. However, the large number of coins that may need to be spent easily yields an inefficient scheme. To solve this problem, researchers have focused on designing offline e-cash protocols whose complexity does not depend on the number of coins withdrawn or spent. In *anonymous compact e-cash* schemes [7, 14], the cost of storing a wallet of N coins and the cost of withdrawing N coins is independent of N. However, the cost of spending $n \leq N$ coins grows linearly with n. *Anonymous divisible e-cash* schemes [10, 18–20, 46, 49] improve the efficiency of compact e-cash and allow the user to spend $n \leq N$ coins with cost independent of n. Therefore, in the last decade, research has focused solely on divisible e-cash. However, divisible schemes achieve constant spending cost at the expense of much more expensive deposit and identification phases, and to our knowledge, efficiency analysis of compact e-cash schemes with multiple denominations has not been conducted, as well as a fair comparison between practical implementations of divisible and compact e-cash has never been done. Therefore, it is unclear which scheme is better for the use-case of offline e-cash as required by privacy-enhanced CBDCs [4] and blockchain-based scalability. Our analysis shows that compact e-cash with multiple denominations is preferable for small payments, which would naturally compose the majority of offline e-cash transactions in application scenarios such as a user-facing blockchain or CBDC where practical deployment concerns would necessitate distributed authorities.

In order to address the urgent scalability issues of blockchain systems and possibly dangerous centralization of CBDCs without privacy, a formal treatment of offline e-cash is needed, including a fair comparison of compact and divisible e-cash.

3 THRESHOLD ISSUANCE OFFLINE E-CASH

In this section, we introduce an offline e-cash scheme with threshold issuance (EC). First, we outline its system model and informally define the security properties. To define formally the security properties of EC, we construct the ideal functionality $\mathcal{F}_{\text{EC}}$ and explain how it guarantees those properties.

3.1 System Model

An EC scheme involves n authorities $(\mathcal{V}_1, \ldots, \mathcal{V}_n)$, any number of users $\mathcal{U}_j$ and any number of providers $\mathcal{P}_k$. The interaction between those parties takes place through a *setup*, *withdrawal*, *spend* and *deposit* phase. Users withdraw wallets containing one or more electronic coins from the authorities and spend them with providers, who then deposit them back with the authorities.

In the **setup** phase, a trusted third party generates the public parameters *params*. Next, the public verification key *pk* is generated, alongside key pairs $(sk_{\mathcal{V}_i}, pk_{\mathcal{V}_i})_{i\in[1,n]}$ for each of the authorities $\mathcal{V}_i$. The keys are generated in such a way that a user needs to receive a withdrawal from at least t authorities in order to create a wallet. The key generation can be executed by a trusted third party or run in a distributed way [34, 38]. Finally, each user $\mathcal{U}_j$ generates a key pair $(sk_{\mathcal{U}_j}, pk_{\mathcal{U}_j})$.

To obtain a wallet with L coins, where L is a public parameter of the scheme, a user $\mathcal{U}_j$ engages in the **withdrawal** protocol. To this end, $\mathcal{U}_j$ sends a request *req* to a set of t different authorities. Each $\mathcal{V}_i$ verifies *req* and using its secret key $sk_{\mathcal{V}_i}$ issues back a response *res*. $\mathcal{U}_j$ verifies *res* and extracts from it a partial wallet W_i. Once $\mathcal{U}_j$ has completed the protocol with at least t authorities, and collected the threshold number of partial wallets, $\mathcal{U}_j$ aggregates them to form a single consolidated wallet W with L coins of the same monetary value. To **spend** V coins with a provider $\mathcal{P}_k$ the user generates a payment *pay* using her wallet W and payment information *payinfo*. *payinfo* contains the provider's identity and other information about the payment and must be unique for each payment. $\mathcal{U}_j$ sends *pay* to $\mathcal{P}_k$, who verifies it. To **deposit** the payment *pay*, provider $\mathcal{P}_k$ sends *pay* to a bulletin board BB. Each authority reads *pay* from BB, verifies it and checks it against all the payments previously written on BB, in order to rule out double spending and double depositing. The double spending detection mechanism reveals the public key of the user $\mathcal{U}_j$ if the user double-spent any coin, while double depositing detection reveals that the payment is deposited twice if two payments contain the same payment information *payinfo*. Otherwise, the payment is deposited successfully.

3.2 Security Properties

As defined in [10], secure anonymous offline e-cash schemes should satisfy four properties. We describe them informally, taking into account that, in our schemes, the bank is replaced by a number of authorities.

Traceability: guarantees that no more coins can be deposited than those that have been withdrawn. In particular, adversarial parties are not able to forge wallets. It also guarantees that an honest authority is able to identify a user who double-spends a coin. Double-depositing is also detected by the authority.

Unlinkability: ensures that no coalition of dishonest authorities, users and providers is able to link the withdrawal of the wallet with the spending of its coins. It also guarantees that multiple spendings performed by the same user cannot be linked with each other.

Exculpability: requires that an honest user cannot be found guilty of double-spending.

Clearance: ensures that only the provider that receives a payment is able to deposit it.

3.3 Ideal Functionality

We define the security properties of offline e-cash with threshold issuance in the ideal-world/real-world paradigm. To this end, in Figure 1 we define the ideal functionality $\mathcal{F}_{\text{EC}}$. $\mathcal{F}_{\text{EC}}$ interacts with authorities $(\mathcal{V}_1, \ldots, \mathcal{V}_n)$, any number of users $\mathcal{U}_j$ and any number of providers $\mathcal{P}_k$. $\mathcal{F}_{\text{EC}}$ is parameterized by a threshold t, a universe of pseudonyms $\mathbb{U}_p$, a universe of wallet identifiers U_w, a universe of request identifiers U_{req}, a universe of payment information U_{info}, and a number L of coins in a wallet. In §A, we define the security properties of the cryptographic primitives used in our EC schemes.

Remarks about $\mathcal{F}_{\text{EC}}$. When describing ideal functionalities, we use the conventions introduced in [13] and summarised in §B.

Aborts. When invoked by any party, $\mathcal{F}_{\text{EC}}$ first checks the correctness of the input. $\mathcal{F}_{\text{EC}}$ aborts if any of the inputs does not belong to the correct domain. $\mathcal{F}_{\text{EC}}$ also aborts if an interface is invoked at an incorrect moment in the protocol. For example, an authority $\mathcal{V}_i$ cannot invoke the ec.issue interface on input a request identifier *reqid* if that authority did not receive a request associated with *reqid*. Similar abortion conditions are listed when $\mathcal{F}_{\text{EC}}$ receives a message from the simulator $\mathcal{S}$.

Session identifier. The session identifier *sid* has the structure $(\mathcal{V}_1, \ldots, \mathcal{V}_n, sid')$. This allows any authorities $(\mathcal{V}_1, \ldots, \mathcal{V}_n)$ to create an instance of $\mathcal{F}_{\text{EC}}$. After the first invocation of $\mathcal{F}_{\text{EC}}$, $\mathcal{F}_{\text{EC}}$ implicitly checks that the session identifier in a message is equal to the one received in the first invocation.

Query identifiers. Before $\mathcal{F}_{\text{EC}}$ queries the simulator $\mathcal{S}$, $\mathcal{F}_{\text{EC}}$ saves its state, which is recovered when receiving a response from $\mathcal{S}$. If an interface, e.g. ec.request, can be invoked by a party more than once, $\mathcal{F}_{\text{EC}}$ creates a query identifier *qid*, which allows $\mathcal{F}_{\text{EC}}$ to match a query to $\mathcal{S}$ to a response from $\mathcal{S}$. Creating *qid* is not necessary if an interface, such as ec.setup, can be invoked only once by each authority, and the authority identifier is revealed to $\mathcal{S}$.

Description of $\mathcal{F}_{\text{EC}}$. In the following, we explain how each of the interfaces of $\mathcal{F}_{\text{EC}}$ works:

(1) An authority $\mathcal{V}_i$ uses the ec.setup interface to set up $\mathcal{F}_{\text{EC}}$. $\mathcal{F}_{\text{EC}}$ stores the fact that $\mathcal{V}_i$ has run the setup interface and enforces that each authority runs the setup interface only once. The simulator $\mathcal{S}$ is allowed to learn that $\mathcal{V}_i$ has run the setup interface. $\mathcal{F}_{\text{EC}}$ allows each authority $\mathcal{V}_i$ to run the setup interface independently of other authorities, i.e. the execution of the setup interface for one authority can be finalized without the involvement of other authorities. Therefore, $\mathcal{F}_{\text{EC}}$ is realizable by protocols where authorities run the setup interface independently of each other. For example, protocols where each authority creates its own keys, or protocols where authorities obtain their keys from a trusted third party that generates them. $\mathcal{F}_{\text{EC}}$ can be modified so that it is realizable by protocols in which the setup interface requires interaction between authorities, e.g. protocols that use a distributed key generation protocol as a building block.

(2) A user $\mathcal{U}_j$ or a provider $\mathcal{P}_k$ use the ec.register interface to register. $\mathcal{F}_{\text{EC}}$ stores the fact that $\mathcal{U}_j$ or $\mathcal{P}_k$ has registered, and enforces that each user or provider runs the registration interface only once. The simulator $\mathcal{S}$ is allowed to learn that $\mathcal{U}_j$ or $\mathcal{P}_k$ has run the setup interface.

(3) A user $\mathcal{U}_j$ runs the ec.request interface given an authority identifier $\mathcal{V}_i$, a request identifier *reqid* and a wallet number *wn*. The request identifier *reqid* is used to bind a request to its subsequent issuance, while the wallet number *wn* is used to associate the request to a wallet. $\mathcal{F}_{\text{EC}}$ checks that the user has run the registration interface, and that there is not a request pending from $\mathcal{U}_j$ to $\mathcal{V}_i$ with the same identifier. Then, if a database for $\mathcal{U}_j$ is not stored, $\mathcal{F}_{\text{EC}}$ stores a tuple $(sid, \mathcal{U}_j, wct, \text{DB})$, where *wct* is a counter of the number of wallets of $\mathcal{U}_j$ initialized to 1, and DB is a (initially empty) database. DB has entries of the form $[wn, wid, l, \mathbb{V}]$, where *wn* is the wallet number, *wid* is a wallet identifier, l is the number of coins spent from the wallet *wn*, and $\mathbb{V}$ is the set of authorities that issued requests for the creation of this wallet. If *wn* received as input is such that $wn \notin [1, wct]$, then $\mathcal{F}_{\text{EC}}$ increments *wct*, sets $wn \leftarrow wct$, picks a random wallet identifier *wid* and stores a new entry $[wn, wid, 0, \emptyset]$ in the database for $\mathcal{U}_j$. The reason for creating *wid* is that,

(1) On input (ec.setup.ini, *sid*) from an authority $\mathcal{V}_i$:
- Abort if $sid \neq (\mathcal{V}_1, \ldots, \mathcal{V}_n, sid')$, or if $\mathcal{V}_i \notin sid$, or if $n < t$, or if $(sid, \mathcal{V}_i, 0)$ is already stored.
- Store $(sid, \mathcal{V}_i, 0)$ and send (ec.setup.sim, $sid, \mathcal{V}_i$) to $\mathcal{S}$.

S. On input (ec.setup.rep, $sid, \mathcal{V}_i$) from $\mathcal{S}$:
- Abort if $(sid, \mathcal{V}_i, 0)$ is not stored or if $(sid, \mathcal{V}_i, 1)$ is already stored.
- Store $(sid, \mathcal{V}_i, 1)$ and send (ec.setup.end, *sid*) to $\mathcal{V}_i$.

(2) On input (ec.register.ini, *sid*) from user $\mathcal{U}_j$ or provider $\mathcal{P}_k$:
- Abort if there is a tuple $(sid, \mathcal{U}_j, 0)$ stored.
- Store $(sid, \mathcal{U}_j, 0)$ and send (ec.register.sim, $sid, \mathcal{U}_j$) to $\mathcal{S}$.

S. On input (ec.register.rep, $sid, \mathcal{U}_j$) from the simulator $\mathcal{S}$:
- Abort if $(sid, \mathcal{U}_j, 0)$ is not stored or if $(sid, \mathcal{U}_j, 1)$ is stored.
- Store $(sid, \mathcal{U}_j, 1)$ and send (ec.register.end, *sid*) to $\mathcal{U}_j$.

(3) On input (ec.request.ini, $sid, \mathcal{V}_i, reqid, wn$) from user $\mathcal{U}_j$:
- Abort if $sid \neq (\mathcal{V}_1, \ldots, \mathcal{V}_n, sid')$, or if $\mathcal{V}_i \notin sid$, or if $n < t$, or if $reqid \notin U_{req}$.
- Abort if $(sid, \mathcal{U}_j, 1)$ is not stored, or if $(sid, \mathcal{U}'_j, \mathcal{V}'_i, reqid', wn,$ user) stored such that $\mathcal{U}'_j = \mathcal{U}_j$, $\mathcal{V}'_i = \mathcal{V}_i$ and $reqid' = reqid$.
- Store $(sid, \mathcal{U}_j, \mathcal{V}_i, reqid, wn,$ user).
- If $(sid, \mathcal{U}'_j, wct,$ DB) such that $\mathcal{U}'_j = \mathcal{U}_j$ is not stored, store $(sid, \mathcal{U}_j, wct,$ DB), where *wct* is a counter of the number of wallets of $\mathcal{U}_j$ initialized to 1, and DB is a (initially empty) database.
- If $wn \notin [1, wct]$, set $wct \leftarrow wct + 1$, set $wn \leftarrow wct$, pick random $wid \leftarrow U_w$, and store an entry $[wn, wid, 0, \emptyset]$ in DB. Update *wct* and DB in the tuple $(sid, \mathcal{U}_j, wct,$ DB).
- Create a fresh *qid* and store $(qid, \mathcal{U}_j, \mathcal{V}_i, reqid, wn)$.
- If $\mathcal{V}_i$ is honest, send (ec.request.sim, $sid, qid, \mathcal{U}_j, \mathcal{V}_i$) to the simulator $\mathcal{S}$, else pick *wid* from the entry $[wn', wid, 0, \emptyset] \in$ DB such that $wn' = wn$ and send (ec.request.sim, $sid, qid, \mathcal{U}_j, \mathcal{V}_i, wid$) to the simulator $\mathcal{S}$.

S. On input (ec.request.rep, sid, qid) from the simulator $\mathcal{S}$:
- Abort if $(qid', \mathcal{U}_j, \mathcal{V}_i, reqid, wn)$ such that $qid' = qid$ is not stored, or if $(sid, \mathcal{V}_i, 1)$ is not stored.
- Store the tuple $(sid, \mathcal{U}_j, \mathcal{V}_i, reqid, wn,$ authority), delete $(qid, \mathcal{U}_j, \mathcal{V}_i, reqid, wn)$, and send (ec.request.end, $sid, \mathcal{U}_j, reqid$) to $\mathcal{V}_i$.

(4) On input (ec.issue.ini, $sid, \mathcal{U}_j, reqid$) from an authority $\mathcal{V}_i$:
- Abort if a tuple $(sid, \mathcal{U}'_j, \mathcal{V}'_i, reqid', wn,$ authority) such that $\mathcal{U}'_j = \mathcal{U}_j$, $\mathcal{V}'_i = \mathcal{V}_i$ and $reqid' = reqid$ is not stored.
- Create a fresh *qid*, store $(qid, \mathcal{U}_j, \mathcal{V}_i, reqid, wn,$ issue) and delete $(sid, \mathcal{U}_j, \mathcal{V}_i, reqid, wn,$ authority).
- Send (ec.issue.sim, $sid, qid, \mathcal{V}_i, \mathcal{U}_j$) to the simulator $\mathcal{S}$.

S. On input (ec.issue.rep, sid, qid) from the simulator $\mathcal{S}$:
- Abort if $(qid', \mathcal{U}_j, \mathcal{V}_i, reqid, wn,$ issue) such that $qid' = qid$ is not stored.
- If $\mathcal{U}_j$ or $\mathcal{V}_i$ are honest, take the tuple $(sid, \mathcal{U}'_j, wct,$ DB) such that $\mathcal{U}'_j = \mathcal{U}_j$, and replace the entry $[wn', wid, l, \mathbb{V}]$ in DB such that $wn' = wn$ by $[wn', wid, l, \mathbb{V} \cup \{\mathcal{V}_i\}]$.
- Delete $(qid, \mathcal{U}_j, \mathcal{V}_i, reqid, wn,$ issue), and delete $(sid, \mathcal{U}'_j, \mathcal{V}'_i, reqid', wn,$ user) such that $\mathcal{U}'_j = \mathcal{U}_j$, $\mathcal{V}'_i = \mathcal{V}_i$ and $reqid' = reqid$.
- Send (ec.issue.end, $sid, reqid, \mathcal{V}_i$) to $\mathcal{U}_j$.

(5) On input (ec.spend.ini, $sid, wn, V, \mathbb{D}, payinfo, P, \mathcal{P}_k$) from $\mathcal{U}_j$:
- Abort if $P \notin \mathbb{U}_p$, or if $payinfo \notin U_{info}$, or if $payinfo$ does not contain $\mathcal{P}_k$, or if $(sid, \mathcal{U}'_j, wct,$ DB) such that $\mathcal{U}'_j = \mathcal{U}_j$ is not stored.
- Abort if there is not an entry $[wn', wid, l, \mathbb{V}] \in$ DB such that $wn' = wn$. Else, proceed as follows:
  - Abort if $\mathcal{U}_j$ is honest and either $V \notin [1, L]$ or $l + V > L$.
  - Abort if $\mathcal{U}_j$ is corrupt and $\mathbb{D} \notin [1, L]$. Else overwrite $V \leftarrow |\mathbb{D}|$.
  - Abort if $|\mathbb{V}| < t'$, where $t' = t$, if $\mathcal{U}_j$ is honest, or $t' = t - \tilde{t}$, if $\mathcal{U}_j$ is corrupt and there are $\tilde{t}$ corrupt authorities.
- If $\mathcal{U}_j$ is honest, update $[wn, wid, l, \mathbb{V}]$ to $[wn, wid, l + V, \mathbb{V}]$.
- Create a fresh *qid* and store $(qid, \mathcal{U}_j, wn, V, \mathbb{D}, payinfo, P, \mathcal{P}_k)$.
- Send (ec.spend.sim, sid, qid) to $\mathcal{S}$.

S. On input (ec.spend.rep, sid, qid) from $\mathcal{S}$:
- Abort if $(qid', \mathcal{U}_j, wn, V, \mathbb{D}, payinfo, P, \mathcal{P}_k)$ such that $qid' = qid$ is not stored and if $(sid, \mathcal{P}'_k, 1)$ such that $\mathcal{P}'_k = \mathcal{P}_k$ is not stored.
- Create a random unique payment identifier *payid* and store $(sid, payid, \mathcal{U}_j, wn, V, \mathbb{D}, payinfo, P, \mathcal{P}_k, 0)$.
- Delete $(qid, \mathcal{U}_j, wn, V, \mathbb{D}, payinfo, P, \mathcal{P}_k)$.
- Send (ec.spend.end, $payid, V, payinfo, P$) to $\mathcal{P}_k$.

(6) On input (ec.deposit.ini, $sid, payid$) from a provider $\mathcal{P}_k$:
- Abort if a tuple $(sid, payid', \mathcal{U}_j, wn, V, \mathbb{D}, payinfo, P, \mathcal{P}'_k, b)$ such that $payid' = payid$, $\mathcal{P}'_k = \mathcal{P}'_k$ and $b = 0$ is not stored.
- Create a fresh *qid* and store $(qid, payid, \mathcal{P}_k)$.
- If $\mathcal{U}_j$ is corrupt and there is at least one corrupt authority, send (ec.deposit.sim, $sid, qid, \mathcal{U}_j, wn, V, \mathbb{D}, payinfo, P, \mathcal{P}_k$) to $\mathcal{S}$. Else if $\mathcal{U}_j$ is honest and there is at least one corrupt authority, send (ec.deposit.sim, $sid, qid, V, payinfo$) to $\mathcal{S}$. Else, send (ec.deposit.sim, sid, qid) to $\mathcal{S}$.

S. On input (ec.deposit.rep, sid, qid) from $\mathcal{S}$:
- Abort if a tuple $(qid', payid, \mathcal{P}_k)$ such that $qid' = qid$ is not stored.
- Update the stored tuple $(sid, payid', \mathcal{U}_j, wn, V, \mathbb{D}, payinfo, P, \mathcal{P}'_k, b)$ such that $payid' = payid$, $\mathcal{P}'_k = \mathcal{P}'_k$ and $b = 0$ to contain $b = 1$.
- Delete $(qid, payid, \mathcal{P}_k)$.
- Send (ec.deposit.end, $sid, payid$) to $\mathcal{P}_k$.

(7) On input (ec.depvf.ini, $sid, \mathbb{U}, payinfo$) from an authority $\mathcal{V}_i$:
- Abort if $(sid, \mathcal{V}'_i, 1)$ such that $\mathcal{V}'_i = \mathcal{V}_i$ is not stored or if there exists $\mathcal{U}_j \in \mathbb{U}$ such that a tuple $(sid, \mathcal{U}_j, 1)$ is not stored.
- Create a fresh *qid* and store $(qid, \mathbb{U}, payinfo, \mathcal{V}_i)$.
- Send (ec.depvf.sim, $sid, qid, \mathbb{U}', d$) to $\mathcal{S}$. $\mathbb{U}'$ is the set of users $\mathcal{U}_j \in \mathbb{U}$ such that $\mathcal{U}_j$ did not send a request to $\mathcal{V}_i$ and $\mathcal{U}_j$ was not received previously as input by $\mathcal{V}_i$ in another set $\mathbb{U}$. *d* is the number of payments deposited since ec.depvf was last invoked by $\mathcal{V}_i$.

S. On input (ec.depvf.rep, sid, qid) from $\mathcal{S}$:
- Abort if $(qid', \mathbb{U}, payinfo, \mathcal{V}_i)$ such that $qid' = qid$ is not stored.
- If there is no tuple $(sid, payid, \mathcal{U}_j, wn, V, \mathbb{D}, payinfo', P, \mathcal{P}_k, b)$ such that $payinfo' = payinfo$ and $b = 1$, set $c \leftarrow 0$.
- If there are $K > 1$ tuples $(sid, payid, \mathcal{U}_j, wn, V, \mathbb{D}, payinfo', P, \mathcal{P}_k, b)$ such that $payinfo' = payinfo$ and $b = 1$, set $c \leftarrow (\mathcal{P}_1, \ldots, \mathcal{P}_K)$, where $(\mathcal{P}_1, \ldots, \mathcal{P}_K)$ are the provider identities such that either $\mathcal{P}_k$ is not included in *payinfo*, or there are two or more tuples deposited by $\mathcal{P}_k$.
- If there is 1 tuple $(sid, payid, \mathcal{U}_j, wn, V, \mathbb{D}, payinfo', P, \mathcal{P}_k, b)$ such that $payinfo' = payinfo$ and $b = 1$, proceed as follows:
  - If $\mathcal{P}_k$ is not in *payinfo*, output $c \leftarrow \mathcal{P}_k$.
  - Else, if $\mathcal{U}_j$ is honest, set $c \leftarrow 1$.
  - Else, if $\mathcal{U}_j$ is corrupt, check if there are other tuples $(sid, payid, \mathcal{U}'_j, wn', V, \mathbb{D}', payinfo, P, \mathcal{P}_k, b)$ such that $\mathcal{U}'_j = \mathcal{U}_j$, $wn' = wn'$ and $b = 1$. For all such tuples, check if $\mathbb{D}' \cap \mathbb{D} = \emptyset$. If that is the case for all tuples, set $c \leftarrow 1$. Else, if $\mathcal{U}_j \in \mathbb{U}$, set $c \leftarrow \mathcal{U}_j$, else set $c \leftarrow \perp$.
- Delete $(qid, \mathbb{U}, payinfo, \mathcal{V}_i)$.
- Send (ec.depvf.end, $sid, payinfo, c$) to $\mathcal{V}_i$.

Figure 1: Ideal Functionality $\mathcal{F}_{\text{EC}}$

in our e-cash schemes, requests for the same wallet can be linked to each other if authorities communicate with each other. Therefore, when $\mathcal{V}_i$ is dishonest, $\mathcal{F}_{\text{EC}}$ leaks *wid* to the simulator $\mathcal{S}$. $\mathcal{F}_{\text{EC}}$ does not leak *wn* because dishonest authorities do not necessarily learn how many wallets a user requests.

After being prompted by the simulator $\mathcal{S}$, $\mathcal{F}_{\text{EC}}$ checks that $\mathcal{V}_i$ has run the setup interface, records that $\mathcal{U}_j$ sends a request to $\mathcal{V}_i$ with identifier *reqid* for wallet *wn*, and sends $\mathcal{U}_j$ and *reqid* to $\mathcal{V}_i$.

(4) An authority $\mathcal{V}_i$ runs the ec.issue interface given a user identifier $\mathcal{U}_j$ and a request identifier *reqid*. $\mathcal{F}_{\text{EC}}$ checks whether there is a request identifier *reqid* pending for a request from $\mathcal{U}_j$ to $\mathcal{V}_i$. In that case, after being prompted by the simulator $\mathcal{S}$, $\mathcal{F}_{\text{EC}}$ records that $\mathcal{V}_i$ has run the issuance for the request *reqid*. Concretely, $\mathcal{F}_{\text{EC}}$ updates the entry $[wn, wid, l, \mathbb{V}]$ in DB such that *wn* is the wallet number associated with *reqid* to contain $\mathcal{V}_i$ in the set $\mathbb{V}$. Finally, $\mathcal{F}_{\text{EC}}$ informs $\mathcal{U}_j$ that $\mathcal{V}_i$ has run the issuance for the request *reqid*.

(5) A user $\mathcal{U}_j$ runs the ec.spend interface given a wallet number *wn*, a number V of coins to be spent, a set of coin indices $\mathbb{D}$, payment information *payinfo*, a pseudonym P and a provider identifier $\mathcal{P}_k$. The payment information *payinfo* should be unique for each spending, but $\mathcal{F}_{\text{EC}}$ checks that later when verifying a deposit. If $\mathcal{U}_j$ is honest, $\mathcal{F}_{\text{EC}}$ checks that there are enough non-spent coins related to *wn* ($l + V \leq L$, where l is in the entry $[wn, wid, l, \mathbb{V}] \in \text{DB}$) and in that case adds V to the number of spent coins. In contrast, if $\mathcal{U}_j$ is corrupt, $\mathcal{F}_{\text{EC}}$ records that the coins with indices $\mathbb{D}$ are spent, regardless of whether they were spent before or not. $\mathcal{F}_{\text{EC}}$ also checks that enough authorities have issued the wallet *wn* to $\mathcal{U}_j$. In that case, after being prompted by the simulator, $\mathcal{F}_{\text{EC}}$ checks that $\mathcal{P}_k$ has run the registration interface and creates a payment identifier *payid* to store the information related to this spending. Finally $\mathcal{F}_{\text{EC}}$ sends *payid*, V, *payinfo* and P to $\mathcal{P}_k$. We remark that $\mathcal{P}_k$ does not learn $\mathcal{U}_j$, and P can be different for each spending so that spendings by $\mathcal{U}_j$ are unlinkable to each other and to the request phase.

(6) A provider $\mathcal{P}_k$ runs the ec.deposit interface on input a payment identifier *payid*. If there is a payment with identifier *payid* related to $\mathcal{P}_k$ that is not deposited, $\mathcal{F}_{\text{EC}}$ proceeds to deposit it. If no authorities are corrupt, $\mathcal{F}_{\text{EC}}$ does not leak to $\mathcal{S}$ any information about the deposited payment. However, if at least one authority is corrupt and the user that computed the payment is corrupt, $\mathcal{F}_{\text{EC}}$ leaks the full information about the payment. If at least one authority is corrupt but the user that computed the payment is honest, the authority leaks the number V of coins spent and the payment information *payinfo*. After being prompted by the simulator, $\mathcal{F}_{\text{EC}}$ marks that payment as deposited and informs $\mathcal{P}_k$ that the payment has been deposited.

(7) An honest authority $\mathcal{V}_i$ runs the ec.depvf interface given a list of user identifiers $\mathbb{U}$ and payment information *payinfo*. $\mathcal{F}_{\text{EC}}$ checks that $\mathcal{V}_i$ has run the setup interface and that all the users in $\mathbb{U}$ have run the registration interface. $\mathcal{F}_{\text{EC}}$ leaks to the simulator the identities of those users in $\mathbb{U}$ that were unknown by $\mathcal{V}_i$. This is done because, in our protocol, $\mathcal{V}_i$ needs to retrieve the public key for that user, and the adversary learns that. $\mathcal{F}_{\text{EC}}$ also leaks the number of deposits that were made since the last time $\mathcal{V}_i$ run the deposit verification interface. This is done because, in our protocol, the authority needs to read the new deposits from the bulletin board, and the adversary learns that. We remark that those leakages can be avoided in our protocols by using a registration functionality that does not leak the retrieved public key to the adversary and a bulletin board functionality that does not leak the number of read operations. We allow this leakage in our functionality because we think that hiding such information is not crucial and because, thanks to that, more efficient constructions can realize our functionality.

After being prompted by the simulator, $\mathcal{F}_{\text{EC}}$ checks the deposited payment with payment information *payinfo*. If no such deposited payment exists, $\mathcal{F}_{\text{EC}}$ sets $c \leftarrow 0$. If there is more than one deposited payment with *payinfo*, $\mathcal{F}_{\text{EC}}$ sets c to contain the identifiers of the provider(s) that deposited those payments more than once, or just once if the identity of the provider is not in *payinfo*. If there is 1 such tuple, $\mathcal{F}_{\text{EC}}$ sets c to the identity of the provider that deposited the payment if that identity is not in *payinfo*. Else, $\mathcal{F}_{\text{EC}}$ sets $c \leftarrow 1$ if the user that made a payment with *payinfo* is honest, which means that there has not been a double spending. If the user is corrupt, $\mathcal{F}_{\text{EC}}$ checks whether there are deposited payments where the coins spent in the payment related with *payinfo* have also been spent. If that is not the case, $\mathcal{F}_{\text{EC}}$ sets $c \leftarrow 1$. Else, if the user identifier is in $\mathbb{U}$, $\mathcal{F}_{\text{EC}}$ sets c to the identifier of the user that double spent, and otherwise sets $c \leftarrow \perp$, which indicates that double spending has been detected but the user has not been identified. $\mathcal{F}_{\text{EC}}$ sends c along with *payinfo* to $\mathcal{V}_i$.

Security properties. We now argue that $\mathcal{F}_{\text{EC}}$ guarantees the security properties defined in §3.2.

Traceability. In the ec.spend interface, when a user wishes to spend coins in the wallet *wn*, $\mathcal{F}_{\text{EC}}$ checks that the user has been issued that wallet by at least $t - \tilde{t}$ authorities. This guarantees that users cannot forge wallets. Moreover, in the ec.depvf interface, $\mathcal{F}_{\text{EC}}$ guarantees that, if coins were double spent, the user is identified whenever the user identifier is included in the set $\mathbb{U}$. $\mathcal{F}_{\text{EC}}$ also finds providers guilty of wrongly depositing a payment, either when they deposit a payment with the same *payinfo* more than once, or when they deposit a payment with *payinfo* that does not contain the provider's identity.

Unlinkability. In the ec.spend interface, the user identity is not revealed to the provider. The provider only receives a pseudonym, which can be different at each spending. This guarantees that payments from the same user cannot be linked with each other or to withdrawals by that user.

Exculpability. $\mathcal{F}_{\text{EC}}$ never finds an honest user guilty of double spending. Any protocol that realizes $\mathcal{F}_{\text{EC}}$ must guarantee that.

Clearance. In the ec.depvf interface, $\mathcal{F}_{\text{EC}}$ never accepts a deposit as valid if the identity of the provider that made the deposit is not contained in *payinfo*.

We remark that $\mathcal{F}_{\text{EC}}$ does not take into account the case where t or more authorities are corrupt. We will analyze the security of construction Π_{EC} under the assumption that at most $t-1$ authorities are corrupt.

Discussion. $\mathcal{F}_{\text{EC}}$ defines an ideal protocol that consists of the usual phases of previous e-cash schemes and provides the security properties typically guaranteed by previous schemes. Therefore, $\mathcal{F}_{\text{EC}}$ can be used to analyze the security of other schemes in the ideal-world/real-world paradigm. In particular, it can be used for schemes without threshold issuance by setting $n = 1$ and $t = 1$.

4 CONSTRUCTION Π_{EC}

In Figure 2, we describe our construction Π_{EC} for $\mathcal{F}_{\text{EC}}$. Π_{EC} uses the ideal functionalities $\mathcal{F}_{\text{SMT}}$ for secure message transmission, $\mathcal{F}_{\text{NYM}}$ for a pseudonymous channel, $\mathcal{F}_{\text{KG}}$ for key generation, $\mathcal{F}_{\text{REG}}$ for registration and $\mathcal{F}_{\text{BB}}$ for an authenticated bulletin board, which are described in C. $\mathcal{F}_{\text{SMT}}$ is used for the communication channel between users and authorities in the ec.request and ec.issue interfaces, while $\mathcal{F}_{\text{NYM}}$ is used for the communication channel between users and providers in the ec.spend interface. $\mathcal{F}_{\text{KG}}$ runs algorithms Setup and KeyGenV, and is used in the ec.setup and ec.register interfaces to generate and distribute both the parameters of the scheme and the keys of the authorities. $\mathcal{F}_{\text{REG}}$ is used in the ec.register interface to register the user public keys, and in the ec.issue and ec.depvf interfaces to give those keys to authorities. $\mathcal{F}_{\text{BB}}$ is used in the ec.deposit interface to deposit payments, and in the ec.depvf interface to let authorities retrieve the deposited payments. We remark that Π_{EC} also uses a functionality for random oracle $\mathcal{F}_{\text{RO}}$ as in [50] to model the random oracle queries done in the algorithms used as a building block. However, this is omitted in the description of Π_{EC}.

In the ec.register interface, providers generate their own key pair, although it is not used later in the protocol. This is done to simplify the description of the protocol by making users and providers call the same ec.register interface. Nevertheless, when the protocol is instantiated by replacing the ideal functionalities used as building blocks with concrete protocols that realize them, providers will need to generate their own keys.

Π_{EC} uses the algorithms defined below. In §5, we instantiate them for both our compact and divisible e-cash schemes. We define these algorithms to provide a unique description of Π_{EC} that contains all the operations that are repeated in both our compact and divisible e-cash schemes.

Setup($1^k, L$). It computes the system parameters *params* on input the security parameter 1^k and the number of coins L in a full wallet. These parameters are publicly available.

KeyGenV(*params*, t, n). Given *params*, the threshold t, and the number of authorities n, output the public verification key pk and the key pairs $(sk_{\mathcal{V}_i}, pk_{\mathcal{V}_i})_{i\in[1,n]}$ for each of the authorities.

KeyGenU(*params*). It is run by each user $\mathcal{U}_j$ to generate a secret key and a public key $(sk_{\mathcal{U}_j}, pk_{\mathcal{U}_j})$.

Request(*params*, $sk_{\mathcal{U}_j}$). On input *params* and the secret key $sk_{\mathcal{U}_j}$ of the user $\mathcal{U}_j$, output a request *req* and request information *reqinfo*.

RequestVf(*params*, *req*, $pk_{\mathcal{U}_j}$). Given *params*, a request *req* and the public key $pk_{\mathcal{U}_j}$ of the user $\mathcal{U}_j$, output 1 if the request is valid and 0 otherwise.

Issue(*params*, $sk_{\mathcal{V}_i}$, *req*). On input *params*, the secret key of authority $\mathcal{V}_i$ and a request *req*, output a response *res*.

IssueVf(*params*, $pk_{\mathcal{V}_i}$, $sk_{\mathcal{U}_j}$, *res*, *reqinfo*). Given *params*, the public key $pk_{\mathcal{V}_i}$ of authority $\mathcal{V}_i$, the secret key $sk_{\mathcal{U}_j}$ of user $\mathcal{U}_j$, a response *res* and request information *reqinfo*, output a partial wallet W_i if the response *res* is correct and is associated to a request with request information *reqinfo*, else output 0.

AggrWallet(pk, $sk_{\mathcal{U}_j}$, $\mathbb{S}$, $\langle W_i\rangle_{i\in\mathbb{S}}$). Given the public key pk, the secret key $sk_{\mathcal{U}_j}$, a set of indices $\mathbb{S} \in [1, n]$ and partial wallets $\langle W_i\rangle_{i\in\mathbb{S}}$, output a wallet W if $|\mathbb{S}| \geq t$, else output 0.

Spend(pk, $sk_{\mathcal{U}_j}$, W, *payinfo*, V). Given the public key pk, the secret key $sk_{\mathcal{U}_j}$, a wallet W, payment information *payinfo*, and a number of coins V, outputs an updated wallet W' and a payment *pay* if there are V non-spent coins in W or 0 otherwise.

SpendVf(*pk*, *pay*, *payinfo*). Given the public key *pk*, a payment *pay*, and payment information *payinfo*, output the number V of coins received if the payment is correct, or 0 otherwise.

Identify(*params*, *PK*, pay_1, pay_2, $payinfo_1$, $payinfo_2$). Given the parameters *params*, a list of user public keys *PK*, and two payments pay_1 and pay_2 with respective payment information $payinfo_1$ and $payinfo_2$:

- Output 1 if pay_1 and pay_2 are payments where different coins were used.
- Else, output $payinfo_1$ if $payinfo_1 = payinfo_2$, which indicates that the payment has been double deposited.
- Else, output the public key $pk_{\mathcal{U}_j} \in PK$ of the user $\mathcal{U}_j$ that double spent a coin in payments pay_1 and pay_2.
- Else, output $\perp$.

5 INSTANTIATION OF Π_{EC}

5.1 Threshold Issuance Compact Ecash

Our compact EC scheme is based on the scheme proposed in [14]. In order to provide threshold issuance, we use the Coconut protocol [54] with the modifications in [50]. We also make some changes in the scheme in [14] to improve efficiency (see 5.1.2).

5.1.1 High-level Overview. In [14], a central bank plays the role of the authority. In the setup phase, the bank generates a key pair for a signature scheme, and each of the users generates a key pair.

Withdrawal Phase. A wallet of L coins is a signature under the bank's public key on a user secret key $sk_{\mathcal{U}_j}$ and two random values v and t. The user $\mathcal{U}_j$ obtains the signature from the bank on ($sk_{\mathcal{U}_j}$, v, t) through a blind signature protocol. The bank does not learn any of the signed values, but learns the user public key $pk_{\mathcal{U}_j}$ associated with $sk_{\mathcal{U}_j}$.

Spending Phase. In order to spend coin $l \in [0, L-1]$, $\mathcal{U}_j$ proves in zero-knowledge (ZK) possession of a signature on ($sk_{\mathcal{U}_j}$, v, t). Additionally, $\mathcal{U}_j$ generates a serial number S and a double-spending tag T, which are used to detect and identify double-spenders. S and T are computed by evaluating the pseudorandom function (PF) in §A.6 on input l. Concretely, S is the output of the PF $f_{g,v}$ on input l. T is computed on input $sk_{\mathcal{U}_j}$, the output of the PF $f_{g,t}$ on input l, and $R \leftarrow H(payinfo)$. *payinfo* is given by the provider and should be unique for each payment. $\mathcal{U}_j$ also proves in ZK that S and T are correctly computed.

Deposit phase. The provider sends to the bank the payment received from the user. To check whether the coin has been double-spent, the bank compares the serial number S with the serial numbers of previously received coins. If there is a match, but the payment information *payinfo* is the same in both coins, then the bank finds that the provider has deposited the coin twice. Else, the bank identifies the user that double-spent the coin by using the double-spending tags of both coins.

5.1.2 Our extensions. In our compact EC scheme, the signature scheme is instantiated with PS signatures, which are described in

(1) On input (ec.setup.ini, sid), $\mathcal{V}_i$ does the following:
- Abort if $sid \neq (\mathcal{V}_1, \ldots, \mathcal{V}_n, sid')$, or if $\mathcal{V}_i \notin sid$, or if $n < t$.
- Abort if $(sid, params, pk, sk_{\mathcal{V}_i}, pk_{\mathcal{V}_i})$ is already stored.
- Send (kg.getkey.ini, sid) to $\mathcal{F}_{\text{KG}}$. In its first invocation, $\mathcal{F}_{\text{KG}}$ runs $params \leftarrow \text{Setup}(1^k, L)$ and $(pk, \langle sk_{\mathcal{V}_i}, pk_{\mathcal{V}_i}\rangle_{i\in[1,n]}) \leftarrow \text{KeyGenV}(params, t, n)$. $\mathcal{F}_{\text{KG}}$ sends (kg.getkey.end, sid, $params$, pk, $sk_{\mathcal{V}_i}$, $pk_{\mathcal{V}_i}$) to $\mathcal{V}_i$.
- Store $(sid, params, pk, sk_{\mathcal{V}_i}, pk_{\mathcal{V}_i})$ and output (ec.setup.end, sid).

(2) On input (ec.register.ini, sid), $\mathcal{U}_j$ (or $\mathcal{P}_k$) does the following:
- Abort if $(sid, sk_{\mathcal{U}_j}, pk_{\mathcal{U}_j})$ is already stored.
- Send (kg.retrieve.ini, sid) to $\mathcal{F}_{\text{KG}}$. $\mathcal{F}_{\text{KG}}$ sends (kg.retrieve.end, sid, v). If $v = (params, pk, \langle pk_{\mathcal{V}_i}\rangle_{i\in[1,n]})$, store $(sid, params, pk, \langle pk_{\mathcal{V}_i}\rangle_{i\in[1,n]})$, else abort.
- Run $(sk_{\mathcal{U}_j}, pk_{\mathcal{U}_j}) \leftarrow \text{KeyGenU}(params)$.
- Set $sid_{\text{REG}} \leftarrow (\mathcal{U}_j, sid')$ and send (reg.register.ini, sid_{REG}, $pk_{\mathcal{U}_j}$) to $\mathcal{F}_{\text{REG}}$. $\mathcal{F}_{\text{REG}}$ sends (reg.register.end, sid_{REG}) to $\mathcal{U}_j$.
- Store $(sid, sk_{\mathcal{U}_j}, pk_{\mathcal{U}_j})$ and output (ec.register.end, sid).

(3) On input (ec.request.ini, sid, $\mathcal{V}_i$, $reqid$, wn), $\mathcal{U}_j$ and $\mathcal{V}_i$ do:
- $\mathcal{U}_j$ aborts if $sid \neq (\mathcal{V}_1, \ldots, \mathcal{V}_n, sid')$, or if $\mathcal{V}_i \notin sid$, or if $n < t$, or if $reqid \notin U_{req}$, or if $(sid, sk_{\mathcal{U}_j}, pk_{\mathcal{U}_j})$ is not stored, or if there is $(sid, \mathcal{V}_i', reqid', wn)$ such that $reqid' = reqid$ and $\mathcal{V}_i' = \mathcal{V}_i$.
- If there is not a tuple (sid, wct), $\mathcal{U}_j$ stores $(sid, 0)$.
- If $wn \notin [1, wct]$, $\mathcal{U}_j$ sets $wct \leftarrow wct+1$, sets $wn \leftarrow wct$, runs $(req, reqinfo) \leftarrow \text{Request}(params, sk_{\mathcal{U}_j})$, stores $(sid, wn, req, reqinfo)$ and updates (sid, wct). $\mathcal{U}_j$ stores $(sid, \mathcal{V}_i, reqid, wn)$.
- $\mathcal{U}_j$ sets $sid_{\text{SMT}} \leftarrow (\mathcal{U}_j, \mathcal{V}_i, sid')$ and sends (smt.send.ini, sid_{SMT}, $\langle reqid, req\rangle$) to $\mathcal{F}_{\text{SMT}}$.
- $\mathcal{V}_i$ receives (smt.send.end, sid_{SMT}, $\langle reqid, req\rangle$) from $\mathcal{F}_{\text{SMT}}$.
- $\mathcal{V}_i$ aborts if $(sid, params, pk, sk_{\mathcal{V}_i}, pk_{\mathcal{V}_i})$ is not stored.
- $\mathcal{V}_i$ aborts if there is a tuple $(sid, reqid', req, \mathcal{U}_j')$ stored such that $reqid' = reqid$ and $\mathcal{U}_j' = \mathcal{U}_j$.
- $\mathcal{V}_i$ parses sid_{SMT} as $(\mathcal{U}_j, \mathcal{V}_i, sid')$. If $(sid, \mathcal{U}_j', pk_{\mathcal{U}_j})$ such that $\mathcal{U}_j' = \mathcal{U}_j$ is not stored, $\mathcal{V}_i$ does the following:
  - $\mathcal{V}_i$ sets $sid_{\text{REG}} \leftarrow (\mathcal{U}_j, sid')$ and sends (reg.retrieve.ini, sid_{REG}) to $\mathcal{F}_{\text{REG}}$. $\mathcal{F}_{\text{REG}}$ sends (reg.retrieve.end, sid_{REG}, $pk_{\mathcal{U}_j}$) to $\mathcal{V}_i$.
  - If $pk_{\mathcal{U}_j} = \perp$, $\mathcal{V}_i$ aborts, else $\mathcal{V}_i$ stores $(sid, \mathcal{U}_j, pk_{\mathcal{U}_j})$.
- $\mathcal{V}_i$ runs $b \leftarrow \text{RequestVf}(params, req, pk_{\mathcal{U}_j})$. If $b = 0$, $\mathcal{V}_i$ aborts, else $\mathcal{V}_i$ stores $(sid, reqid, req, \mathcal{U}_j)$.
- $\mathcal{V}_i$ outputs (ec.request.end, sid, $\mathcal{U}_j$, $reqid$).

(4) On input (ec.issue.ini, sid, $\mathcal{U}_j$, $reqid$), $\mathcal{V}_i$ and $\mathcal{U}_j$ do the following:
- $\mathcal{V}_i$ aborts if $(sid, reqid', req, \mathcal{U}_j')$ such that $reqid' = reqid$ and $\mathcal{U}_j' = \mathcal{U}_j$ is not stored.
- $\mathcal{V}_i$ runs $res \leftarrow \text{Issue}(params, sk_{\mathcal{V}_i}, req)$.
- $\mathcal{V}_i$ deletes $(sid, reqid, req, \mathcal{U}_j)$, sets $sid_{\text{SMT}} \leftarrow (\mathcal{V}_i, \mathcal{U}_j, sid')$ and sends (smt.send.ini, sid_{SMT}, $\langle reqid, res\rangle$) to $\mathcal{F}_{\text{SMT}}$.
- $\mathcal{U}_j$ receives (smt.send.end, sid_{SMT}, $\langle reqid, res\rangle$) from $\mathcal{F}_{\text{SMT}}$.
- $\mathcal{U}_j$ parses sid_{SMT} as $(\mathcal{V}_i, \mathcal{U}_j, sid')$. $\mathcal{U}_j$ aborts if a tuple $(sid, \mathcal{V}_i', reqid', wn)$ such that $reqid' = reqid$ and $\mathcal{V}_i' = \mathcal{V}_i'$ is not stored. Else $\mathcal{U}_j$ takes the stored tuple $(sid, wn', req, reqinfo)$ such that $wn' = wn$ and runs $W_i \leftarrow \text{IssueVf}(params, pk_{\mathcal{V}_i}, sk_{\mathcal{U}_j}, res, reqinfo)$. If $W_i = 0$, $\mathcal{U}_j$ aborts.
- If there is not a tuple $(sid, wn', \mathbb{S}, \mathbb{W})$ such that $wn' = wn$, $\mathcal{U}_j$ stores $(sid, wn, \emptyset, \emptyset)$.
- $\mathcal{U}_j$ updates $(sid, wn, \mathbb{S}, \mathbb{W})$ to $(sid, wn, \mathbb{S} \cup \{i\}, \mathbb{W} \cup \{W_i\})$.
- If (sid, wn', W) such that $wn' = wn$ is not stored, and if $|\mathbb{S}| \geq t$ in the tuple $(sid, wn', \mathbb{S}, \mathbb{W})$ such that $wn' = wn$, then $\mathcal{U}_j$ runs $W \leftarrow \text{AggrWallet}(pk, sk_{\mathcal{U}_j}, \mathbb{S}, \mathbb{W})$ and stores (sid, wn, W).
- $\mathcal{U}_j$ deletes $(sid, \mathcal{V}_i, reqid, wn)$.
- Output (ec.issue.end, sid, $reqid$, $\mathcal{V}_i$).

(5) On input (ec.spend.ini, sid, wn, V, $\mathbb{D}$, $payinfo$, P, $\mathcal{P}_k$), $\mathcal{U}_j$ and $\mathcal{V}_i$ do:
- $\mathcal{U}_j$ aborts if $P \notin \mathbb{U}_p$, or if $payinfo \notin U_{info}$, or if $payinfo$ does not contain $\mathcal{P}_k$, or if (sid, wn', W) such that $wn' = wn$ is not stored, or if $V \notin [1, L]$.
- $\mathcal{U}_j$ runs $b \leftarrow \text{Spend}(pk, sk_{\mathcal{U}_j}, W, payinfo, V)$. If $b = 0$, $\mathcal{U}_j$ aborts. Else $\mathcal{U}_j$ parses b as (W', pay) and updates the stored tuple (sid, wn, W) to (sid, wn, W').
- $\mathcal{U}_j$ sends (nym.send.ini, sid, $\langle pay, payinfo\rangle$, P, $\mathcal{P}_k$) to $\mathcal{F}_{\text{NYM}}$.
- $\mathcal{P}_k$ receives (nym.send.end, sid, $\langle pay, payinfo\rangle$, P) from $\mathcal{F}_{\text{NYM}}$.
- $\mathcal{P}_k$ aborts if $(sid, params, pk, \langle pk_{\mathcal{V}_i}\rangle_{i\in[1,n]})$ is not stored, or if $payinfo$ does not contain $\mathcal{P}_k$.
- $\mathcal{P}_k$ runs $b \leftarrow \text{SpendVf}(pk, pay, payinfo)$. If $b = 0$, $\mathcal{P}_k$ aborts, else $\mathcal{P}_k$ sets $V \leftarrow b$, creates a random unique payment identifier $payid$ and stores $(sid, payid, pay, payinfo, V, 0)$.
- $\mathcal{P}_k$ outputs (ec.spend.end, $payid$, V, $payinfo$, P).

(6) On input (ec.deposit.ini, sid, $payid$), $\mathcal{P}_k$ does the following:
- Abort if a tuple $(sid, payid', pay, payinfo, V, b)$ such that $payid' = payid$ and $b = 0$ is not stored.
- Send (bb.write.ini, sid, $\langle pay, payinfo\rangle$) to the functionality $\mathcal{F}_{\text{BB}}$. $\mathcal{F}_{\text{BB}}$ sends (bb.write.end, sid) to $\mathcal{P}_k$.
- Update the tuple $(sid, payid, pay, payinfo, V, b)$ so that $b = 1$.
- Output (ec.deposit.end, sid, $payid$).

(7) On input (ec.depvf.ini, sid, $\mathbb{U}$, $payinfo$) from an authority $\mathcal{V}_i$:
- Abort if $(sid, params, pk, sk_{\mathcal{V}_i}, pk_{\mathcal{V}_i})$ is not stored.
- For all $\mathcal{U}_j \in \mathbb{U}$, if $(sid, \mathcal{U}_j', pk_{\mathcal{U}_j})$ such that $\mathcal{U}_j' = \mathcal{U}_j$ is not stored, do the following:
  - Set $sid_{\text{REG}} \leftarrow (\mathcal{U}_j, sid')$ and send (reg.retrieve.ini, sid_{REG}) to $\mathcal{F}_{\text{REG}}$. $\mathcal{F}_{\text{REG}}$ sends back (reg.retrieve.end, sid_{REG}, $pk_{\mathcal{U}_j}$).
  - If $pk_{\mathcal{U}_j} = \perp$, abort, else store $(sid, \mathcal{U}_j, pk_{\mathcal{U}_j})$.
- Include in PK the public keys $pk_{\mathcal{U}_j}$ in all the stored tuples $(sid, \mathcal{U}_j, pk_{\mathcal{U}_j})$ such that $\mathcal{U}_j \in \mathbb{U}$.
- If (sid, i) is not stored, set $i = 1$ and store (sid, i).
- While $m' \neq \perp$, do the following:
  - Send (bb.read.ini, sid, i) to $\mathcal{F}_{\text{BB}}$. $\mathcal{F}_{\text{BB}}$ sends back (bb.read.end, sid, m'). If $m' = (\mathcal{P}_k, pay, payinfo)$, store $(sid, \mathcal{P}_k, pay, payinfo)$.
  - Increment i.
- Update i in the tuple (sid, i).
- Find all the stored tuples $(sid, \mathcal{P}_k, pay, payinfo')$ such that $payinfo' = payinfo$ and do the following:
  - If there is no tuple such that $payinfo' = payinfo$, set $c \leftarrow 0$.
  - If there are $K > 1$ tuples such that $payinfo' = payinfo$, set $c \leftarrow (\mathcal{P}_1, \ldots, \mathcal{P}_K)$, where $(\mathcal{P}_1, \ldots, \mathcal{P}_K)$ are providers that deposited payments with $payinfo$ more than once, or that deposited a payment such that $\mathcal{P}_k$ is not included in $payinfo$.
  - If there is one tuple such that $payinfo' = payinfo$, output $c \leftarrow \mathcal{P}_k$ if the identity of the provider is not in $payinfo$. Else, for all the remaining tuples $(sid, \mathcal{P}_k', pay', payinfo')$, run the algorithm $c \leftarrow \text{Identify}(params, PK, pay, pay', payinfo, payinfo')$ until $c \neq 1$. If $c = 1$ for all tuples, set $c \leftarrow 1$.
- Output (ec.depvf.end, sid, $payinfo$, c).

Figure 2: Construction Π_{EC}

§A.5. In the setup phase, the secret keys $sk_{\mathcal{V}_i}$ for each of the authorities $(\mathcal{V}_1, \ldots, \mathcal{V}_n)$ are generated by evaluating random polynomials of degree $t-1$ on input $[1, n]$, while the verification key pk used to verify wallets corresponds to a secret key that would be given by the evaluation of those polynomials on input 0. In the withdrawal phase, the user runs a blind signature protocol with at least t authorities. After obtaining at least t valid signatures, the user uses Lagrange interpolation to obtain a signature verifiable with pk.

A wallet is a signature on $(sk_{\mathcal{U}_j}, v)$. In comparison to [14], we remove the secret t. Thanks to this change, the size of the wallet is smaller, and the ZK proofs used in both the withdrawal and spending phase are more efficient in comparison to [14]. To make this change possible, we modify the way the serial number S and double spending tag T are computed. Concretely, S is the output of the PF $f_{\delta,v}$ on input l, and the computation of T uses the evaluation of the PF $f_{g,v}$ on input l, i.e., we use a new generator δ for the computation of the serial numbers. This change allows us to use the same secret v as the index of both PFs without compromising the security of our scheme. In §F, we quantify the cost reduction attained by removing t. We further improve the efficiency of the withdrawal phase, by removing the need for the bank to contribute randomness to create v in the blind signature protocol in [14]. In our protocol, the user picks v and as discussed in §5.1.4, this change does not compromise the security of our scheme. We also improve efficiency by using one ZK proof π_v to spend V coins, instead of repeating V times the spending protocol for one coin.

5.1.3 Construction. The algorithms of our compact EC scheme are defined below. In §A, we describe the cryptographic primitives used by the algorithms.

Setup$(1^k, L)$**.** Execute the following steps:
- Run $(p, \mathbb{G}, \tilde{\mathbb{G}}, \mathbb{G}_t, e, g, \tilde{g}) \leftarrow \mathcal{G}(1^k)$.
- Pick 3 random generators $(\gamma_1, \gamma_2, \delta) \leftarrow \mathbb{G}$.
- Output $params \leftarrow (p, \mathbb{G}, \tilde{\mathbb{G}}, \mathbb{G}_t, e, g, \tilde{g}, \gamma_1, \gamma_2, \delta, L)$.

KeyGenV$(params, t, n)$**.** Execute the following steps:
- Choose $(1+2)$ polynomials (v, w_1, w_2) of degree $(t-1)$ with random coefficients in $\mathbb{Z}_p$.
- Set $(x, y_1, y_2) \leftarrow (v(0), w_1(0), w_2(0))$.
- For $i = 1$ to n, set the secret key $sk_{\mathcal{V}_i}$ of each authority $\mathcal{V}_i$ as $sk_{\mathcal{V}_i} = (x_i, y_{i,1}, y_{i,2}) \leftarrow (v(i), w_1(i), w_2(i))$.
- For $i = 1$ to n, set the verification key $pk_{\mathcal{V}_i}$ of each authority $\mathcal{V}_i$ as $pk_{\mathcal{V}_i} = (\tilde{\alpha}_i, \beta_{i,1}, \tilde{\beta}_{i,1}, \beta_{i,2}, \tilde{\beta}_{i,2}) \leftarrow (\tilde{g}^{x_i}, g^{y_{i,1}}, \tilde{g}^{y_{i,1}}, g^{y_{i,2}}, \tilde{g}^{y_{i,2}})$.
- Set the verification key $pk = (params, \tilde{\alpha}, \beta_1, \tilde{\beta}_1, \beta_2, \tilde{\beta}_2) \leftarrow (params, \tilde{g}^x, g^{y_1}, \tilde{g}^{y_1}, g^{y_2}, \tilde{g}^{y_2})$.
- Output $(pk, \langle pk_{\mathcal{V}_i}, sk_{\mathcal{V}_i}\rangle_{i=1}^n)$.

KeyGenU$(params)$**.** Execute the following steps:
- Pick random $sk_{\mathcal{U}_j} \leftarrow \mathbb{Z}_p$ and compute $pk_{\mathcal{U}_j} \leftarrow g^{sk_{\mathcal{U}_j}}$.
- Output $(sk_{\mathcal{U}_j}, pk_{\mathcal{U}_j})$.

Request$(params, sk_{\mathcal{U}_j})$**.** Execute the following steps:
- Pick random $v \leftarrow \mathbb{Z}_p$ and set $(m_1, m_2) = (sk_{\mathcal{U}_j}, v)$.
- Pick random $o \leftarrow \mathbb{Z}_p$ and compute $com = g^o \prod_{j=1}^2 \gamma_j^{m_j}$.
- Compute $h \leftarrow H(com)$, where H is a hash function modeled as a random oracle.
- Compute commitments to each of the messages. For $j = 1$ to 2, pick random $o_j \leftarrow \mathbb{Z}_p$ and set $com_j = g^{o_j} h^{m_j}$.
- Compute a ZK argument of knowledge π_s via the Fiat-Shamir heuristic for the following relation: (In our schemes, we use the Fiat-Shamir heuristic for efficiency reasons. We discuss an alternative in C.6.)

$$\pi_s = \text{NIZK}\{(m_1, m_2, o, o_1, o_2) : com = g^o \prod_{j=1}^{2} \gamma_j^{m_j} \wedge pk_{\mathcal{U}_j} \leftarrow g^{m_1} \wedge \{com_j = g^{o_j} h^{m_j}\}_{\forall j \in [1,2]}\}$$

- Set $reqinfo \leftarrow (h, o_1, o_2, v)$ and $req \leftarrow (h, com, com_1, com_2, \pi_s)$. Output req and $reqinfo$.

RequestVf$(params, req, pk_{\mathcal{U}_j})$**.** Execute the following steps:
- Parse req as $(h, com, com_1, com_2, \pi_s)$.
- Compute $h' \leftarrow H(com)$ and output 0 if $h \neq h'$.
- Verify the ZK argument π_s by using the tuple $(params, h, com, com_1, com_2, pk_{\mathcal{U}_j})$. Output 0 if the proof π_s is not correct, else output 1.

Issue$(params, sk_{\mathcal{V}_i}, req)$**.** Execute the following steps:
- Parse req as $(h, com, com_1, com_2, \pi_s)$ and $sk_{\mathcal{V}_i}$ as $(x_i, y_{i,1}, y_{i,2})$.
- Compute $c = h^{x_i} \prod_{j=1}^2 com_j^{y_{i,j}}$.
- Set the blinded signature share $\hat{\sigma}_i \leftarrow (h, c)$.
- Output $res \leftarrow \hat{\sigma}_i$.

IssueVf$(params, pk_{\mathcal{V}_i}, sk_{\mathcal{U}_j}, res, reqinfo)$**.** Do the following:
- Parse $reqinfo$ as (h', o_1, o_2, v), res as $\hat{\sigma}_i = (h, c)$, and $pk_{\mathcal{V}_i}$ as $(\tilde{\alpha}_i, \beta_{i,1}, \tilde{\beta}_{i,1}, \beta_{i,2}, \tilde{\beta}_{i,2})$. Output 0 if $h \neq h'$.
- Compute $\sigma_i = (h, s) \leftarrow (h, c \prod_{j=1}^2 \beta_{i,j}^{-o_j})$.
- Set $(m_1, m_2) \leftarrow (sk_{\mathcal{U}_j}, v)$. Output 0 if $e(h, \tilde{\alpha}_i \prod_{j=1}^2 \tilde{\beta}_{i,j}^{m_j}) = e(s, \tilde{g})$ does not hold.
- Output $W_i \leftarrow (i, \sigma_i, v)$.

AggrWallet$(pk, sk_{\mathcal{U}_j}, \mathbb{S}, \langle W_i\rangle_{i\in\mathbb{S}})$**.** Execute the following steps:
- If $|\mathbb{S}| \neq t$, output 0.
- For all $i \in \mathbb{S}$, evaluate at 0 the Lagrange basis polynomials $l_i = [\prod_{j\in\mathbb{S}, j\neq i}(0-j)][\prod_{j\in\mathbb{S}, j\neq i}(i-j)]^{-1} \bmod p$
- For all $i \in \mathbb{S}$, parse W_i as (i, σ_i, v) and σ_i as (h, s_i).
- Compute the signature $\sigma = (h, s) \leftarrow (h, \prod_{i\in\mathbb{S}} s_i^{l_i})$.
- Parse pk as $(params, \tilde{\alpha}, \beta_1, \tilde{\beta}_1, \beta_2, \tilde{\beta}_2)$.
- Set $(m_1, m_2) = (sk_{\mathcal{U}_j}, v)$ and output 0 if $e(h, \tilde{\alpha} \prod_{j=1}^2 \tilde{\beta}_j^{m_j}) = e(s, \tilde{g})$ does not hold, else output $W \leftarrow (\sigma, v, l)$, where l is a counter from 0 to $L-1$ initialized to 0.

Spend$(pk, sk_{\mathcal{U}_j}, W, payinfo, V)$**.** Execute the following steps:
- Parse W as (σ, v, l). If $l + V - 1 \geq L$, output 0.
- Parse σ as (h, s) and pk as $(params, \tilde{\alpha}, \beta_1, \tilde{\beta}_1, \beta_2, \tilde{\beta}_2)$.
- Pick random $r \leftarrow \mathbb{Z}_p$ and $r' \leftarrow \mathbb{Z}_p$.
- Compute $\sigma' = (h', s') \leftarrow (h^{r'}, s^{r'}(h')^r)$ and $\kappa \leftarrow \tilde{\alpha} \tilde{\beta}_1^{sk_{\mathcal{U}_j}} \tilde{\beta}_2^v \tilde{g}^r$.
- Pick random $o_c \leftarrow \mathbb{Z}_p$ and compute the commitment $C \leftarrow g^{o_c} \gamma_1^v$.
- For $k \in [0, V-1]$, compute $R_k \leftarrow H'(payinfo, k)$, where $payinfo$ must contain the identifier of the merchant, and H' is a collision-resistant hash function.
- For $k \in [0, V-1]$, set $l_k \leftarrow l+k$, pick random o_{a_k} and compute $A_k = g^{o_{a_k}} \gamma_1^{l_k}$.
- For $k \in [0, V-1]$, compute the serial numbers $S_k \leftarrow f_{\delta,v}(l_k) = \delta^{1/(v+l_k+1)}$ and also compute the double spending tags $T_k \leftarrow g^{sk_{\mathcal{U}_j}} (f_{g,v}(l_k))^{R_k} = g^{sk_{\mathcal{U}_j} + R_k/(v+l_k+1)}$.

- For $k \in [0, V-1]$, compute the values $\mu_k \leftarrow 1/(v + l_k + 1)$ and $o_{\mu_k} \leftarrow -(o_{a_k} + o_c)\mu_k$.
- Compute a ZK argument of knowledge π_v via the Fiat-Shamir heuristic for the following relation:

$$\begin{aligned}\pi_v = \text{NIZK}\{&(sk_{\mathcal{U}_j}, v, r, o_c, \langle l_k, o_{a_k}, \mu_k, o_{\mu_k}\rangle_{k=0}^{V-1}) : \\ &\kappa = \tilde{\alpha}\tilde{\beta}_1^{sk_{\mathcal{U}_j}}\tilde{\beta}_2^{v}\tilde{g}^{r} \ \wedge \ C = g^{o_c}\gamma_1^{v} \ \wedge \\ &\langle A_k = g^{o_{a_k}}\gamma_1^{l_k} \ \wedge \ l_k \in [0, L-1] \ \wedge \\ &S_k = \delta^{\mu_k} \ \wedge \ \gamma_1 = (A_k C\gamma_1)^{\mu_k} g^{o_{\mu_k}} \ \wedge \\ &T_k = g^{sk_{\mathcal{U}_j}} (g^{R_k})^{\mu_k} \ \rangle_{k\in[0,V-1]}\}\end{aligned}$$

  In §F, we explain this ZK proof and show how to prove the statement $l_k \in [0, L-1]$.
- Output a payment $pay \leftarrow (\kappa, \sigma', \langle S_k, T_k, A_k\rangle_{k\in[0,V-1]}, V, C, \pi_v)$ and an updated wallet $W' \leftarrow (\sigma, v, l+V)$.

SpendVf(pk, pay, $payinfo$). Execute the following steps:

- Parse pk as $(params, \tilde{\alpha}, \beta_1, \tilde{\beta}_1, \beta_2, \tilde{\beta}_2)$.
- Parse pay as $(\kappa, \sigma', \langle S_k, T_k, A_k\rangle_{k\in[0,V-1]}, V, C, \pi_v)$.
- Parse σ' as (h', s') and output 0 if $h' = 1$ or if $e(h', \kappa) = e(s', \tilde{g})$ does not hold.
- Output 0 if not all the serial numbers $\langle S_k\rangle_{k\in[0,V-1]}$ are different from each other.
- For $k \in [0, V-1]$, compute $R_k \leftarrow H'(payinfo, k)$.
- Verify π_v by using $payinfo$, pk, $\langle S_k, T_k, A_k, R_k\rangle_{k\in[0,V-1]}$, C and κ. Output 0 if the proof is not correct, else output V.

Identify($params$, PK, pay_1, pay_2, $payinfo_1$, $payinfo_2$). Do:

- Parse pay_1 as $(\kappa_1, \sigma_1', \langle S_{k,1}, T_{k,1}, A_{k,1}\rangle_{k\in[0,V_1-1]}, V_1, C_1, \pi_{v,1})$.
- Parse pay_2 as $(\kappa_2, \sigma_2', \langle S_{k,2}, T_{k,2}, A_{k,2}\rangle_{k\in[0,V_2-1]}, V_2, C_2, \pi_{v,2})$.
- For $k \in [0, V_1-1]$, for $j \in [0, V_2-1]$, check whether $S_{k,1} = S_{j,2}$. If the equality never holds, output 1.
- Else, output $payinfo_1$ if $payinfo_1 = payinfo_2$.
- Else, for $k \in [0, V_1-1]$ and $j \in [0, V_2-1]$ such that $S_{k,1} = S_{j,2}$, compute $pk_{\mathcal{U}_j} \leftarrow (T_{j,2}^{R_{k,1}} / T_{k,1}^{R_{j,2}})^{(R_{k,1}-R_{j,2})^{-1}}$ If $pk_{\mathcal{U}_j} \in PK$ output $pk_{\mathcal{U}_j}$, else output $\perp$.

5.1.4 Security Analysis of Compact E-Cash. In §D, we prove formally that Π_{EC}, when instantiated with the algorithms of our compact EC scheme, realizes $\mathcal{F}_{\text{EC}}$. In this section, we give intuition on why our scheme is secure.

Unlinkability. In the withdrawal phase, a corrupt authority does not learn the user secrets $(sk_{\mathcal{U}_j}, v)$ thanks to the hiding property of the Pedersen commitment scheme and to the ZK property of the argument π_s. In the spend phase, a corrupt provider does not learn anything from a payment beyond the number of coins spent. To prove that, several properties are used. First, we use the ZK property of the argument π_v. Second, to prove that C and A_k do not reveal any information about v or l_k, we use the hiding property of the Pedersen commitment scheme. Third, to prove that S_k and T_k do not reveal any information about $pk_{\mathcal{U}_j}$, v or l_k, we use the pseudorandomness property of the PF, along with the XDH assumption.[1] Finally, as in the modified version of Coconut in [50], the PS signature is "randomized" in a way that enables us to prove signature possession without revealing any information about the original signature or the signed messages.

Traceability, Exculpability and Clearance. In order to prove that a user cannot spend coins that she has not withdrawn before, we use several properties of our building blocks. The weak simulation extractability property allows us to extract the witnesses from the ZK arguments π_s. Thanks to that extraction, we can use the binding property of the commitment scheme com included in a request message to ensure that different commitments com and com' commit to different tuples $(sk_{\mathcal{U}_j}, v)$. This is required for the unforgeability of PS signatures in the RO model. We remark that com is the input to the random oracle and that it is necessary to ensure that a different generator h is created to sign different message tuples. The binding property of com guarantees that. Second, we use the weak simulation extractability property of arguments π_v to extract the witnesses. Thanks to that, in the spending phase, we can extract a signature on $(sk_{\mathcal{U}_j}, v)$ from a payment message and show that, if a user did not withdraw at least t signatures from t different authorities, then the user can be used to break the existential unforgeability property of PS signatures in the RO model.

Therefore, we know that, if more coins are deposited than those being withdrawn, it is the case that a user has double-spent coins or that a provider has double-deposited coins. In the deposit phase, an authority checks that payments that are deposited have different serial numbers. We show that the extractability of π_v, along with the discrete logarithm assumption, guarantees that the serial numbers and double spending tags are correctly computed. Therefore, if two payments have at least one common serial number, there are three possibilities: (1) Double depositing, (2) double spending, (3) none of the former. Double depositing can be punished by checking whether two payments are associated with the same payment information $payinfo$, which contains the identifier of the provider. We recall that $payinfo$ is signed in π_v. Moreover, our construction Π_{EC} uses an authenticated bulletin board. Thus, an authority can check that the provider that deposits a payment is the same whose identity is in $payinfo$. The latter guarantees the clearance property. If double depositing did not happen because the payment information $payinfo$ and $payinfo'$ is different in those payments, the authority can retrieve the public key of the user who double spent a coin through the computation described in the algorithm Identify. This computation requires that $R_k \leftarrow H'(payinfo, k)$ is different from $R'_{k'} \leftarrow H'(payinfo', k')$. We show that, if the hash function H' is collision-resistant, the double spender can always be identified. In our scheme, a corrupt user is able to compute two payments where there is no double spending, yet two serial numbers are equal. The reason is that, unlike in the scheme in [14], the user picks the coin secret v on its own. When a corrupt user does that, our security analysis guarantees that, under the hardness of the discrete logarithm assumption, algorithm Identify will never identify an honest user as the double spender. This guarantees the exculpability property. Moreover, we also show that, under the discrete logarithm assumption, a corrupt user cannot compute a payment with a serial number that is equal to a serial number in a payment computed by an honest user. Consequently, when Identify detects that two serial numbers are equal, but is unable to find the public key of the

[1] In contrast to [14], the XDH assumption is needed in our scheme because we use the same coin secret v to compute S_k and T_k.

user who double spent (i.e. Identify outputs $\perp$), we are in a case in which in fact there is no double-spending.

5.2 Threshold Issuance Divisible E-Cash

In our compact EC scheme in §5.1, the cost of the spending phase grows linearly with the number of coins spent. In divisible e-cash, the cost of the spending phase is independent of the number of coins spent. To make that possible, the main change in comparison to compact e-cash is that the serial numbers of coins are generated during the deposit phase, rather than the spending phase.

Our divisible EC scheme is based on the work by Pointcheval et al. [49], which proposes a scheme in the standard model with Groth-Sahai proofs [36]. We modify that scheme as follows. In [49], a wallet is a signature on two group elements $(U_1, U_2) = (u_1^{sk_{\mathcal{U}_j}}, u_2^v)$ We replace the signature scheme used in [49] by the PS signature scheme, and we sign $(sk_{\mathcal{U}_j}, v)$. Thus, the wallet in our divisible EC scheme has the same form as in our compact EC scheme. Thanks to that, in the withdrawal phase we use the same algorithms Request, RequestVf, Issue, IssueVf and AggrWallet (see §5.1.3) to provide a threshold issuance protocol. At setup, algorithms KeyGenV and KeyGenU also work as in §5.1.3.

In the spending phase in [49], a Groth-Sahai non-interactive ZK proof and a Groth-Sahai non-interactive witness-indistinguishable proof are computed. The latter involves proof of possession of the signature on (U_1, U_2). In our scheme, we use a NIZK argument computed via the Fiat-Shamir heuristic, which involves all the statements proven in both the Groth-Sahai ZK proof and witness-indistinguishable proof. To prove possession of a PS signature on $(sk_{\mathcal{U}_j}, v)$, we use the method depicted in algorithm Spend in §5.1.3.

Because Groth-Sahai proofs are randomizable, in [49], the user needs to compute a one-time signature on the payment. A statement is added to the Groth-Sahai proof to certify the public key used for the one-time signature. In our scheme, this is not needed, because non-interactive ZK arguments computed via the Fiat-Shamir heuristic are signatures of knowledge.

The remaining values computed in the spending phase, and the statements proven about them, are the same in [49] and in our scheme. We note that our non-interactive ZK argument involves proving knowledge of group elements in addition to discrete logarithm representations. We show how this is done in §A.3.

5.2.1 High-level Overview. A wallet of L coins is a signature on $(sk_{\mathcal{U}_j}, v)$, where $sk_{\mathcal{U}_j}$ is the secret key of user $\mathcal{U}_j$ and v is a coin secret. The L serial numbers of the coins in a wallet are given by $\mathrm{SN}_l = e(\varsigma, \tilde{g})^{v y^l}$, $l \in [1, L]$ where values y and ς are part of the parameters of the scheme.

Withdrawal Phase. The withdrawal phase is the same as in our compact EC scheme in §5.1.3.

Spending Phase. In the spending phase, to spend V coins, the user needs to give information that allows the authorities to compute the V serial numbers of the spent coins, but no more than that. To this end, to spend V coins with indices $[l, l+V-1]$, the user computes an ElGamal encryption $\phi_{V,l}$ of ς_l^v under the public key η_V. The values ς_l (for $l \in [1, L]$) and η_V (for $V \in [1, L]$) are part of the public parameters of the scheme. $\phi_{V,l}$ is used in the deposit phase by the authorities to generate the serial numbers $(\mathrm{SN}_l, \ldots, \mathrm{SN}_{l+V-1})$. This ELGamal encryption with public key η_V restricts the authorities to generate only those V serial numbers.

To enable identification of double spenders, the user computes the double spending tag $\varphi_{V,l}$ as an ElGamal encryption of $(g^R)^{sk_{\mathcal{U}_j}} \theta_l^v$ under public key η_V, where R is a hash of the payment information *payinfo* given by the provider. The values θ_l, for $l \in [1, L]$, are part of the parameters of the scheme.

The user also needs to prove in zero-knowledge that $\phi_{V,l}$ and $\varphi_{V,l}$ are correctly computed. To this end, the user proves possession of a signature on $(sk_{\mathcal{U}_j}, v)$ and proves that those values were used to compute $\phi_{V,l}$ and $\varphi_{V,l}$. Additionally, the user needs to prove that the correct values ς_l and θ_l in the public parameters have been used, and that $l \leq L - V + 1$. To allow the user to prove those statements, the public parameters of the scheme contain signatures on the pairs (ς_l, θ_l) (for $l \in [1, L]$). The user proves possession of the signature on the pair $(\varsigma_{l+V-1}, \theta_{l+V-1})$ and proves that (ς_l, θ_l) are correctly chosen through the equations $e(\varsigma_l, \tilde{\delta}_{V-1}) = e(\varsigma_{l+V-1}, \tilde{g})$ and $e(\theta_l, \tilde{\delta}_{V-1}) = e(\theta_{l+V-1}, \tilde{g})$. The values $\tilde{\delta}_k \leftarrow \tilde{g}^{y^k}$ (for $k \in [0, L-1]$) are part of the parameters of the scheme. Although this proof does not prove that $l \geq 1$, in the security analysis it is shown that the user is unable to generate (ς_l, θ_l) such that $l \notin [1, L]$.

Deposit Phase. In the deposit phase, an authority checks whether a coin has been double spent. For this purpose, the authority computes the serial numbers of the spent coins by doing $\mathrm{SN}_k \leftarrow e(\phi_{V,l}[2], \tilde{\delta}_k) e(\phi_{V,l}[1], \tilde{\eta}_{V,k})$ for $k \in [0, V-1]$. Here, the values $\tilde{\eta}_{l,k}$ for $l \in [1, L]$ and $k \in [0, l-1]$ are part of the parameters of the scheme. Because of those values, the size of the parameters is quadratic in the number L of coins in a wallet, but this is only the case for the parameters that the authorities need, i.e., the size of parameters for users is linear in L.

When a collision between serial numbers of two payments is detected, the authority uses the security tags of both payments for identification of the user who double spent. The mechanism used is similar to the one of our compact EC scheme in §5.1. However, in the divisible EC scheme, the authority, rather than computing the user key, checks whether an equality holds for each of the public keys of the users one by one, which is a disadvantage.

5.2.2 Construction. Our divisible e-cash scheme works as follows:

Setup$(1^k, L)$**.** Execute the following steps:

- Run $grp = (p, \mathbb{G}, \tilde{\mathbb{G}}, \mathbb{G}_t, e, g, \tilde{g}) \leftarrow \mathcal{G}(1^k)$.
- Pick random generators $\eta, \gamma_1, \gamma_2 \leftarrow \mathbb{G}$.
- Generate random scalars $(z, y) \leftarrow \mathbb{Z}_p$ and compute $(\varsigma, \theta) \leftarrow (g^z, \eta^z)$. Generate for $l \in [1, L]$, $a_l \leftarrow \mathbb{Z}_p$.
- For $l \in [1, L]$, compute $(\varsigma_l, \theta_l) \leftarrow (\varsigma^{y^l}, \theta^{y^l})$.
- For $k \in [0, L-1]$, compute $\tilde{\delta}_k \leftarrow \tilde{g}^{y^k}$.
- For $l \in [1, L]$, compute $\eta_l \leftarrow g^{a_l}$.
- For $l \in [1, L]$, for $k \in [0, l-1]$, compute $\tilde{\eta}_{l,k} \leftarrow \tilde{g}^{-a_l \cdot y^k}$.
- Run algorithm $(pk_{sps}, sk_{sps}) \leftarrow \mathrm{KeyGen}(grp, 2, 0)$ of the SPS scheme in §A.5.
- For $l \in [1, L]$, compute $\tau_l \leftarrow \mathrm{Sign}(sk_{sps}, \langle \varsigma_l, \theta_l \rangle)$.
- Set the parameters for users $params_u \leftarrow (p, \mathbb{G}, \tilde{\mathbb{G}}, \mathbb{G}_t, e, g, \tilde{g}, \eta, \gamma_1, \gamma_2, \{\eta_l, \varsigma_l, \theta_l, \tau_l\}_{l=1}^{L}, \{\tilde{\delta}_k\}_{k=0}^{L-1}, pk_{sps})$. Set the additional parameters for authorities $params_a \leftarrow (\{\langle \tilde{\eta}_{l,k} \rangle_{k=0}^{l-1}\}_{l=1}^{L-1})$.
- Output $params \leftarrow (params_u, params_a)$.

Spend($pk, sk_{\mathcal{U}_j}, W, payinfo, V$). Execute the following steps:

- Parse W as (σ, v, l) and σ as (h, s). If $l + V - 1 > L$, output 0.
- Parse pk as $(params_u, \tilde{\alpha}, \beta_1, \tilde{\beta}_1, \beta_2, \tilde{\beta}_2)$.
- Pick random scalars $r \leftarrow \mathbb{Z}_p$ and $r' \leftarrow \mathbb{Z}_p$ and compute $\sigma' = (h', s') \leftarrow (h^{r'}, s^{r'}(h')^r)$.
- Compute $\kappa \leftarrow \tilde{\alpha}\tilde{\beta}_1^{sk_{\mathcal{U}_j}}\tilde{\beta}_2^{v}\tilde{g}^r$.
- Pick random $r_1, r_2 \leftarrow \mathbb{Z}_p$ and set $\phi_{V,l} = (\phi_{V,l}[1], \phi_{V,l}[2]) \leftarrow (g^{r_1}, \varsigma_l^v \eta_V^{r_1})$.
- Set $R \leftarrow H'(payinfo)$, where H' is a collision-resistant hash function, and compute

$$\varphi_{V,l} = (\varphi_{V,l}[1], \varphi_{V,l}[2]) \leftarrow (g^{r_2}, (g^R)^{sk_{\mathcal{U}_j}}\theta_l^v \eta_V^{r_2})$$

- Take $params_u$ from pk. Take the public key $pk_{sps} = (Y, W_1, W_2, Z)$ and the signature $\tau_{l+V-1} = (R_{l+V-1}, S_{l+V-1}, T_{l+V-1})$.
- Compute a ZK argument of knowledge π_v via the Fiat-Shamir heuristic for the following relation:

$$\begin{aligned}
\pi_v = \text{NIZK}\{&(sk_{\mathcal{U}_j}, v, r, r_1, r_2, \varsigma_l, \theta_l, \varsigma_{l+V-1}, \\
&\theta_{l+V-1}, R_{l+V-1}, S_{l+V-1}, T_{l+V-1}) : \\
&\kappa = \tilde{\alpha}\tilde{\beta}_1^{sk_{\mathcal{U}_j}}\tilde{\beta}_2^{v}\tilde{g}^r \wedge \\
&\phi_{V,l}[1] = g^{r_1} \wedge \phi_{V,l}[2] = \varsigma_l^v \eta_V^{r_1} \wedge \\
&\varphi_{V,l}[1] = g^{r_2} \wedge \varphi_{V,l}[2] = (g^R)^{sk_{\mathcal{U}_j}}\theta_l^v \eta_V^{r_2} \wedge \\
&e(\varsigma_l, \tilde{\delta}_{V-1}) = e(\varsigma_{l+V-1}, \tilde{g}) \wedge \\
&e(\theta_l, \tilde{\delta}_{V-1}) = e(\theta_{l+V-1}, \tilde{g}) \wedge \\
&e(R_{l+V-1}, T_{l+V-1})e(g, \tilde{g})^{-1} = 1 \wedge \\
&e(R_{l+V-1}, Y)e(S_{l+V-1}, \tilde{g})e(\varsigma_{l+V-1}, W_1) \cdot \\
&e(\theta_{l+V-1}, W_2)e(g, Z)^{-1} = 1
\end{aligned}$$

  This signature of knowledge signs the payment information $payinfo$. Since there are group elements in the witness, the transformation described in §A.3 is needed. We depict the argument after applying the transformation in Appendix G.
- Output a payment $pay \leftarrow (\kappa, \sigma', \phi_{V,l}, \varphi_{V,l}, R, \pi_v, V)$ and an updated wallet $W' \leftarrow (\sigma, v, l + V)$.

SpendVf($pk, pay, payinfo$). Execute the following steps:

- Parse pay as $(\kappa, \sigma', \phi_{V,l}, \varphi_{V,l}, R, \pi_v, V)$ and σ' as (h', s'). Output 0 if $h' = 1$ or if $e(h', \kappa) = e(s', \tilde{g})$ does not hold.
- Output 0 if $R \neq H'(payinfo)$.
- Verify π_v by using $payinfo$, pk, $\phi_{V,l}$, $\varphi_{V,l}$, V, R and κ. Output 0 if the proof is not correct, else output V.

Identify($params, PK, pay_1, pay_2, payinfo_1, payinfo_2$). Compute:

- Parse pay_1 as $(\kappa_1, \sigma_1', \phi_{V_1,l_1,1}, \varphi_{V_1,l_1,1}, R_1, \pi_{v,1}, V_1)$.
- Parse pay_2 as $(\kappa_2, \sigma_2', \phi_{V_2,l_2,2}, \varphi_{V_2,l_2,2}, R_2, \pi_{v,2}, V_2)$.
- For $k \in [0, V_1 - 1]$, compute the serial numbers

$$\text{SN}_{k,1} \leftarrow e(\phi_{V_1,l_1,1}[2], \tilde{\delta}_k)e(\phi_{V_1,l_1,1}[1], \tilde{\eta}_{V_1,k}).$$

  For $k \in [0, V_2 - 1]$, compute the serial numbers

$$\text{SN}_{k,2} \leftarrow e(\phi_{V_2,l_2,2}[2], \tilde{\delta}_k)e(\phi_{V_2,l_2,2}[1], \tilde{\eta}_{V_2,k}).$$

- Output 1 if none of the serial numbers $\text{SN}_{k_1,1}$, for $k_1 \in [0, V_1 - 1]$, is equal to $\text{SN}_{k_2,2}$, for $k_2 \in [0, V_2 - 1]$.
- Else, output $payinfo_1$ if $payinfo_1 = payinfo_2$.
- Else, let $k_1 \in [0, V_1 - 1]$ and $k_2 \in [0, V_2 - 1]$ be two indices such that $\text{SN}_{k_1,1} = \text{SN}_{k_2,2}$. Compute

$$T_1 \leftarrow e(\varphi_{V_1,l_1,1}[2], \tilde{\delta}_{k_1})e(\varphi_{V_1,l_1,1}[1], \tilde{\eta}_{V_1,k_1})$$

  and

$$T_2 \leftarrow e(\varphi_{V_2,l_2,2}[2], \tilde{\delta}_{k_2})e(\varphi_{V_2,l_2,2}[1], \tilde{\eta}_{V_2,k_2}).$$

  For each $pk_{\mathcal{U}_j} \in$ PK, check whether $T_1 T_2^{-1} = e(pk_{\mathcal{U}_j}, \tilde{\delta}_{k_1}^{R_1}\tilde{\delta}_{k_2}^{-R_2})$ and output $pk_{\mathcal{U}_j}$ if the equality holds. Output $\perp$ if the equality does not hold for any $pk_{\mathcal{U}_j} \in$ PK.

5.2.3 Security Analysis of Divisible E-Cash. In §E, we prove formally that Π_{EC}, when instantiated with the algorithms of our divisible EC scheme, realizes $\mathcal{F}_{\text{EC}}$. In this section, we give intuition on why our scheme is secure. The security analysis of our divisible EC scheme is based on the security analysis given for our compact EC scheme regarding the withdrawal phase and the non-interactive ZK argument of possession of PS signatures used in the spending phase. As in our compact EC scheme, the anonymity property also relies on the hiding property of Pedersen commitments and the ZK property of the proof system, as well as on the method to "randomize" signatures in the spend phase. The traceability property relies on the weak simulation extractability property of the non-interactive ZK arguments of knowledge, the binding property of the commitment scheme and the existential unforgeability property of PS signatures in the RO model.

The remaining part of our analysis follows the security proof given in [49] for the divisible e-cash scheme. The anonymity property holds under the N-MXDH' assumption (see §A.2). In [49], it is shown that this assumption holds in the generic bilinear group model.

The traceability property relies on the existential unforgeability of the SPS scheme in [1], which guarantees that the values $(\varsigma_{l+V-1}, \theta_{l+V-1})$ used as a witness in the ZK argument π_v are correct. It also relies on the *BDHI* assumption, which guarantees that the adversary cannot generate values (ς_l, θ_l) such that $l < 1$. Those two properties, along with the above-mentioned binding property of the commitment scheme and the existential unforgeability property of PS signatures in the RO model, guarantee that the ElGamal encryptions $\phi_{V,l}$ and $\varphi_{V,l}$ in payment are computed correctly. This ensures that, if there is double-spending, an authority can identify the user that double-spent a coin.

As noted in [10], in an RO model version of the scheme in [49], like our scheme, we can extract the user's secret key from the ZK argument π_v. Thus, we can show that an honest user cannot be found guilty of double spending under the discrete logarithm assumption. In our scheme, unlike in [49], the authority does not contribute randomness to the generation of the coin secret v. This means that, as in our compact EC scheme, the adversary is able to generate two payments where there is no double spending, and yet there is a match between serial numbers. In that case, it is guaranteed that an honest user will not be found guilty.

As for clearance, like in our compact EC scheme, our construction in §4 guarantees that only the provider that receives a payment can deposit it. This is done by using an authenticated bulletin board and checking that the provider's identity is contained in the payment information $payinfo$.

Table 1: Average number of spent coins given D and P_{max}

P_{max}	D	Avg. Num. Coins
10	$[1, 2, 5]$	1.9
100	$[1, 2, 5, 10, 20, 50]$	3.4
1000	$[1, 2, 5, \ldots, 100, 200, 500]$	5.1
10000	$[1, 2, 5, \ldots, 1000, 2000, 5000]$	6.8
100000	$[1, 2, 5, \ldots, 10000, 20000, 50000]$	8.5
1000000	$[1, 2, 5, \ldots, 10000, 20000, 50000]$	17.5

6 EFFICIENCY ANALYSIS AND COMPARISON

For years research has focused almost exclusively on divisible e-cash because of the constant cost of the spending phase. However, this is achieved at the expense of much more expensive deposit and identification phases. In this section, we compare the efficiency of our compact and divisible EC schemes. To this end, in §F, we describe an instantiation of our compact EC scheme with a concrete set membership proof, and in §G, we describe how the NIZK arguments of our divisible EC scheme are instantiated. Our comparison shows that our compact EC with multiple denominations, an idea mentioned but never explored, keeps efficient deposit and identification phases, while dramatically reducing the spending phase cost as opposed to using one denomination. In fact, when the price range is not large, the concrete (as opposed to asymptotic) cost of compact EC with multiple denominations is smaller than that of divisible EC also in the spending phase. Such a comparison was not done before and can guide the choice of an EC scheme for a practical payment system.

In §6.1, we analyze the average number of coins that need to be spent depending on the choice of denominations. In §6.2, we describe the implementation of our schemes and compare their performance.

6.1 Choice of Multiple Denominations

The spending phase is arguably the phase in which time constraints are more demanding. Spending one coin is more efficient in our compact e-cash scheme than in our divisible e-cash scheme. However, for any real-world payment, multiple coins need to be spent, and consequently, the compact e-cash scheme is not practical. To counter this problem, a solution that has often been suggested in the e-cash literature, but not studied in depth, is to use multiple denominations. A question that arises when using multiple denominations is how to choose those denominations optimally, i.e., to minimize the average number of coins that need to be spent. This *optimal denomination problem* has been studied in the context of Fiat currencies [52]. Assuming that all the prices within a range $[1, P_{max}]$ are equally likely, and fixing the number D of denominations, the problem is to find the set of denominations that minimizes the average number of spent coins. As calculated in [52], for a price range of $[1, 100]$ cents, and the denominations $[1, 5, 10, 25]$, the average number of coins spent is 4.7. However, using the optimal sets, which are $[1, 5, 18, 25]$ and $[1, 5, 18, 29]$, the average is 3.89.

In general, the optimal denomination problem is NP-hard. Given a naive approach of choosing 1 as the smallest denomination (to be able to pay for the lowest price), we must calculate the number of all possible sets of $D - 1$ denominations in the range $[1, P_{max}]$, which is given by the number of combinations without repetition $C_{D-1}(P_{max})$, and the average number of coins needed to pay for prices in $[1, P_{max}]$ for each set of those D denominations. This computation is too expensive for practical values of D and P_{max} (e.g. $D = 15$ and $P_{max} = 1000000$ cents), although it can be optimized by restricting the set of denominations to those that fulfil certain properties.

A problem with using multiple denominations in e-cash is that a user may not be able to pay a price even given enough funds. For example, a user left with two coins of 10 cents is not able to pay for a price of 11 cents. The obvious solution to this problem would be to allow the user to exchange one coin of 10 cents for 10 coins of 1 cent, but this would require interaction with authorities, turning the scheme into an online one. For the scheme to remain offline, the only solution would be for the user to withdraw coins of 1 cent denomination. Therefore, for practical use of the compact e-cash scheme, we need an easy-to-use set of denominations. Taking the denominations of the euro (i.e. $[1, 2, 5, 10, 20, 50, 100, 200, 500, 1000, 2000, 5000, 10000, 20000, 50000]$ cents) as an example, in Table 1 we calculate the average number of coins that need to be spent for different values of P_{max} and D. Although these denominations are not optimal, the results in [52] suggest that the improvement derived from using an optimal denomination set for the same values of D is not dramatic. Furthermore, these denominations allow us to compute the optimal representation for a given price (i.e. the optimal number of coins of each denomination that are needed to pay), by running the simple *greedy algorithm* [52], whereas the optimal denomination set may not allow that. Therefore, we use those denominations to compare the efficiency of our compact e-cash with that of the divisible e-cash scheme.

6.2 Implementation and Comparison

Implementation. We implement the protocols presented in §5.1 and §5.2 in Rust1.56.0. Our implementation is open source.[2]For the implementation of the elliptic curve we use a fork of `bls12_381` library and further extend it to facilitate operations in $\mathbb{G}_t$.[3] All benchmarks were run on a dedicated 16 GB Linode machine, with a 2.2 GHz AMD CPU. To minimize accuracy errors, we execute each measurement 300 times and compute its average. With this specification, a single exponentiation in $\mathbb{G}$ takes approximately 531.24 us, in $\tilde{\mathbb{G}}$ approximately 2.13 ms, while in $\mathbb{G}_t$ approximately 3.85 ms. A single pairing operation takes 2.59 ms.

Evaluation. For our benchmarks, we set the number of authorities to $n = 100$ with a threshold of $t = 70$. The maximum number of coins in a wallet is set to $L = 100$. We also set the number of users to 100, which determines the size of the set of public keys used as input to algorithm Identify. We always place the target public key as the last one PK list.

The withdrawal protocol is the same in both our compact and divisible EC schemes. Algorithm Request takes on average 9.16 ms. The algorithms run by the authorities, i.e. RequestVf and Issue, take in total 8.64 ms on average. Algorithm IssueVf takes 8.47 ms. We remark that to withdraw a wallet, a user needs to run Request only once because the same request is sent to at least t authorities.

[2]https://github.com/nymtech/nym/tree/research/ecash
[3]https://github.com/jstuczyn/bls12_381

Table 2: Performance benchmarks of Compact and Divisible EC. For Compact EC we measure for $V = 1$.

	Compact EC	Divisible EC
Spend	34.75 ms	124.49 ms
SpendVf	34.15 ms	129.81 ms
Identify	1.61 ms	133.81 ms

However, algorithm IssueVf needs to be run at least t times to verify each of the responses. Finally, algorithm AggrWallet, which is run just once after t valid responses have been received from different authorities, takes 65.28 ms. We consider that those timings are practical, and we stress that they do not depend on the number of coins in a full wallet.

Table 2 presents a comparison between spending a single coin in our compact EC scheme vs spending V coins in our divisible EC scheme. Algorithms Spend and SpendVf are almost four times faster in the compact than in the divisible EC scheme. However, as we noted earlier, the cost of the spending phase in the divisible EC scheme is independent of the number of coins spent, while in the compact EC scheme, it grows linearly. As an example, let 1267 be the price to be paid. The total execution time to settle a payment using the compact EC scheme would take 44.03 seconds, thus significantly more than in the divisible EC scheme.

However, the spending phase of our compact e-cash scheme can be optimized as follows. First, algorithm Spend allows us to spend V coins with cost smaller than the cost of running V times Spend to spend one coin. Our benchmarks show that spending $V = 2$ takes on average 53.43 ms, which is almost 25% faster than executing the spend protocol twice sequentially to spend one coin. Second, as discussed in §6.1, we can run several instances of the scheme in parallel and assign to each of them a different denomination. For example, let's consider a set of denominations $[1000, 500, 100, 50, 20, 10, 5, 2, 1]$. Given our price of 1267, the spender now executes the pay function five times with value $V = 1$ for coins $[1000, 50, 10, 5, 2]$ and once with value $V = 2$ for coins with denomination 100. Thus, the total execution time is 261.93 ms, which is significantly faster. However, in case of large payments and a limited number of denominations, the compact e-cash is still significantly inefficient. For example, given denominations $[100, 50, 20, 10, 5, 2, 1]$ we need 425.34 ms to complete a payment of 1267. A similar dependency can be observed in the case of payment verification.

We use our analysis from §6.1 to estimate the average time required to complete a payment using our compact EC scheme given different price ranges $[1, P_{max}]$ and sets of denominations. The results are summarised in Table 3. We observe that our compact EC scheme is more efficient than our divisible EC scheme for small price ranges and small sets of denominations.

The major advantage of the compact EC scheme over the divisible EC scheme is a fast *identification* phase. In the compact EC scheme, to detect double-spending, an authority simply needs to compare serial numbers. In a practical setting, it is likely that double spending happens infrequently, and so the computation cost for the authority is negligible. However, in the divisible EC scheme, the authority needs to compute the serial numbers of a payment by running Identify before the authority can compare them with the serial numbers of other payments. The computation of a serial number involves two pairings.

Table 3: Average time required to complete a payment in Compact EC for different values of D and P_{max}

P_{max}	D	Spend [ms]
10	$[1, 2, 5]$	50.84
100	$[1, 2, 5, 10, 20, 50]$	90.98
1000	$[1, 2, 5, \ldots, 100, 200, 500]$	136.46
10000	$[1, 2, 5, \ldots, 1000, 2000, 5000]$	181.95
100000	$[1, 2, 5, \ldots, 10000, 20000, 50000]$	227.43
1000000	$[1, 2, 5, \ldots, 10000, 20000, 50000]$	468.26

Once double spending is detected, the compact EC scheme can identify the user guilty of double spending in 1.61 ms, independently of the number of users in the system. However, in the divisible EC scheme, the computation is more expensive and it grows linearly with the number of users in the system. We remark that the timings given in Table 2 are calculated for 100 users, but in practice this number could be orders of magnitude bigger, which makes identification in compact e-cash much more efficient than in divisible e-cash.

7 CONCLUSION AND FUTURE WORK

In this work, we proposed the first offline anonymous e-cash scheme with threshold issuance, motivated by the concrete scalability concerns of blockchains and concerns with centralization in CBDCs [4]. We define the ideal functionality and propose two instantiations based on an improved compact and a divisible e-cash. We have shown that our schemes realize the ideal functionality and formally prove their security. We have also implemented both schemes and compared their efficiency, showing that compact e-cash is more efficient and feasible for smaller transactions, which would naturally compose the majority of offline e-cash transactions in application scenarios such as a user-facing blockchain or CBDC where practical deployment concerns would necessitate distributed authorities.

As future work, we will describe how our schemes can be integrated with a blockchain-based bulletin board. We outline how this integration could be done in §H. This would allow our scheme to fulfil the requirements for a distributed privacy-enhanced CBDC [4] and even provide scalability via offline transactions for existing blockchain systems like ZCash [51].

ACKNOWLEDGMENTS

This research received no specific grant from any funding agency in the public, commercial, or not-for-profit sectors.

REFERENCES

[1] Masayuki Abe, Jens Groth, Kristiyan Haralambiev, and Miyako Ohkubo. 2011. Optimal Structure-Preserving Signatures in Asymmetric Bilinear Groups. In *Advances in Cryptology - CRYPTO 2011 - 31st Annual Cryptology Conference, Santa Barbara, CA, USA, August 14-18, 2011. Proceedings*. 649–666. https://doi.org/10.1007/978-3-642-22792-9_37

[2] Foteini Baldimtsi, Melissa Chase, Georg Fuchsbauer, and Markulf Kohlweiss. 2015. Anonymous Transferable E-Cash. In *Public-Key Cryptography - PKC 2015 - 18th IACR International Conference on Practice and Theory in Public-Key Cryptography, Gaithersburg, MD, USA, March 30 - April 1, 2015, Proceedings (Lecture Notes in*

Computer Science, Vol. 9020), Jonathan Katz (Ed.). Springer, 101–124. https://doi.org/10.1007/978-3-662-46447-2_5

[3] Lucas Ballard, Matthew Green, Breno de Medeiros, and Fabian Monrose. 2005. Correlation-Resistant Storage via Keyword-Searchable Encryption. *IACR Cryptol. ePrint Arch.* (2005), 417. http://eprint.iacr.org/2005/417

[4] European Central Bank. 2023. "Market research on possible technical solutions for a digital euro". https://www.ecb.europa.eu/paym/intro/news/html/ecb.mipnews230113.en.html.

[5] Mathieu Baudet, Alberto Sonnino, Mahimna Kelkar, and George Danezis. 2022. Zef: Low-latency, Scalable, Private Payments. *IACR Cryptol. ePrint Arch.* (2022), 83. https://eprint.iacr.org/2022/083

[6] Balthazar Bauer, Georg Fuchsbauer, and Chen Qian. 2021. Transferable E-Cash: A Cleaner Model and the First Practical Instantiation. In *Public-Key Cryptography - PKC 2021 - 24th IACR International Conference on Practice and Theory of Public Key Cryptography, Virtual Event, May 10-13, 2021, Proceedings, Part II (Lecture Notes in Computer Science, Vol. 12711)*, Juan A. Garay (Ed.). Springer, 559–590. https://doi.org/10.1007/978-3-030-75248-4_20

[7] Mira Belenkiy, Melissa Chase, Markulf Kohlweiss, and Anna Lysyanskaya. 2009. Compact E-Cash and Simulatable VRFs Revisited. In *Pairing-Based Cryptography - Pairing 2009, Third International Conference, Palo Alto, CA, USA, August 12-14, 2009, Proceedings (Lecture Notes in Computer Science, Vol. 5671)*, Hovav Shacham and Brent Waters (Eds.). Springer, 114–131. https://doi.org/10.1007/978-3-642-03298-1_9

[8] Dan Boneh and Xavier Boyen. 2004. Efficient Selective-ID Secure Identity Based Encryption Without Random Oracles. *IACR Cryptol. ePrint Arch.* (2004), 172. http://eprint.iacr.org/2004/172

[9] Dan Boneh and Xavier Boyen. 2008. Short Signatures Without Random Oracles and the SDH Assumption in Bilinear Groups. *J. Cryptol.* 21, 2 (2008), 149–177. https://doi.org/10.1007/s00145-007-9005-7

[10] Florian Bourse, David Pointcheval, and Olivier Sanders. 2019. Divisible E-Cash from Constrained Pseudo-Random Functions. In *Advances in Cryptology - ASIACRYPT 2019 - 25th International Conference on the Theory and Application of Cryptology and Information Security, Kobe, Japan, December 8-12, 2019, Proceedings, Part I (Lecture Notes in Computer Science, Vol. 11921)*, Steven D. Galbraith and Shiho Moriai (Eds.). Springer, 679–708. https://doi.org/10.1007/978-3-030-34578-5_24

[11] Elette Boyle, Shafi Goldwasser, and Ioana Ivan. 2014. Functional Signatures and Pseudorandom Functions. In *Public-Key Cryptography - PKC 2014 - 17th International Conference on Practice and Theory in Public-Key Cryptography, Buenos Aires, Argentina, March 26-28, 2014. Proceedings (Lecture Notes in Computer Science, Vol. 8383)*, Hugo Krawczyk (Ed.). Springer, 501–519. https://doi.org/10.1007/978-3-642-54631-0_29

[12] Jan Camenisch, Rafik Chaabouni, and Abhi Shelat. 2008. Efficient Protocols for Set Membership and Range Proofs. In *Advances in Cryptology - ASIACRYPT 2008, 14th International Conference on the Theory and Application of Cryptology and Information Security, Melbourne, Australia, December 7-11, 2008. Proceedings (Lecture Notes in Computer Science, Vol. 5350)*, Josef Pieprzyk (Ed.). Springer, 234–252. https://doi.org/10.1007/978-3-540-89255-7_15

[13] Jan Camenisch, Maria Dubovitskaya, and Alfredo Rial. 2016. UC Commitments for Modular Protocol Design and Applications to Revocation and Attribute Tokens. In *Advances in Cryptology - CRYPTO 2016 - 36th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 14-18, 2016, Proceedings, Part III*. 208–239. https://doi.org/10.1007/978-3-662-53015-3_8

[14] Jan Camenisch, Susan Hohenberger, and Anna Lysyanskaya. 2005. Compact E-Cash. In *Advances in Cryptology - EUROCRYPT 2005, 24th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Aarhus, Denmark, May 22-26, 2005, Proceedings (Lecture Notes in Computer Science, Vol. 3494)*, Ronald Cramer (Ed.). Springer, 302–321. https://doi.org/10.1007/11426639_18

[15] Jan Camenisch, Markulf Kohlweiss, and Claudio Soriente. 2009. An Accumulator Based on Bilinear Maps and Efficient Revocation for Anonymous Credentials. In *Public Key Cryptography - PKC 2009, 12th International Conference on Practice and Theory in Public Key Cryptography, Irvine, CA, USA, March 18-20, 2009. Proceedings (Lecture Notes in Computer Science, Vol. 5443)*, Stanislaw Jarecki and Gene Tsudik (Eds.). Springer, 481–500. https://doi.org/10.1007/978-3-642-00468-1_27

[16] Jan Camenisch, Anja Lehmann, Gregory Neven, and Alfredo Rial. 2014. Privacy-Preserving Auditing for Attribute-Based Credentials. In *Computer Security - ESORICS 2014 - 19th European Symposium on Research in Computer Security, Wroclaw, Poland, September 7-11, 2014. Proceedings, Part II (Lecture Notes in Computer Science, Vol. 8713)*, Miroslaw Kutylowski and Jaideep Vaidya (Eds.). Springer, 109–127. https://doi.org/10.1007/978-3-319-11212-1_7

[17] Jan Camenisch and Markus Stadler. 1997. *Proof Systems for General Statements about Discrete Logarithms.* Technical Report TR 260. Institute for Theoretical Computer Science, ETH Zürich.

[18] Sébastien Canard and Aline Gouget. 2007. Divisible E-Cash Systems Can Be Truly Anonymous. In *Advances in Cryptology - EUROCRYPT 2007, 26th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Barcelona, Spain, May 20-24, 2007, Proceedings (Lecture Notes in Computer Science, Vol. 4515)*, Moni Naor (Ed.). Springer, 482–497. https://doi.org/10.1007/978-3-540-72540-4_28

[19] Sébastien Canard, David Pointcheval, Olivier Sanders, and Jacques Traoré. 2015. Divisible E-Cash Made Practical. In *Public-Key Cryptography - PKC 2015 - 18th IACR International Conference on Practice and Theory in Public-Key Cryptography, Gaithersburg, MD, USA, March 30 - April 1, 2015, Proceedings (Lecture Notes in Computer Science, Vol. 9020)*, Jonathan Katz (Ed.). Springer, 77–100. https://doi.org/10.1007/978-3-662-46447-2_4

[20] Sébastien Canard, David Pointcheval, Olivier Sanders, and Jacques Traoré. 2015. Scalable Divisible E-cash. In *Applied Cryptography and Network Security - 13th International Conference, ACNS 2015, New York, NY, USA, June 2-5, 2015, Revised Selected Papers (Lecture Notes in Computer Science, Vol. 9092)*, Tal Malkin, Vladimir Kolesnikov, Allison Bishop Lewko, and Michalis Polychronakis (Eds.). Springer, 287–306. https://doi.org/10.1007/978-3-319-28166-7_14

[21] Ran Canetti. 2001. Universally Composable Security: A New Paradigm for Cryptographic Protocols. In *42nd Annual Symposium on Foundations of Computer Science, FOCS 2001, 14-17 October 2001, Las Vegas, Nevada, USA*. 136–145. https://doi.org/10.1109/SFCS.2001.959888

[22] David Chaum. 1982. Blind Signatures for Untraceable Payments. In *Advances in Cryptology: Proceedings of CRYPTO '82, Santa Barbara, California, USA, August 23-25, 1982*, David Chaum, Ronald L. Rivest, and Alan T. Sherman (Eds.). Plenum Press, New York, 199–203. https://doi.org/10.1007/978-1-4757-0602-4_18

[23] David Chaum, Christian Grothoff, and Thomas Moser. 2021. How to issue a central bank digital currency. *arXiv preprint arXiv:2103.00254* (2021).

[24] David Chaum and Torben P. Pedersen. 1992. Transferred Cash Grows in Size. In *Advances in Cryptology - EUROCRYPT '92, Workshop on the Theory and Application of of Cryptographic Techniques, Balatonfüred, Hungary, May 24-28, 1992, Proceedings (Lecture Notes in Computer Science, Vol. 658)*, Rainer A. Rueppel (Ed.). Springer, 390–407. https://doi.org/10.1007/3-540-47555-9_32

[25] Vishal Chawla. 2022. "Someone is clogging up the Zcash blockchain with a spam attack". https://www.theblock.co/post/175259/someone-is-clogging-up-the-zcash-blockchain-with-a-spam-attack.

[26] George Danezis and Sarah Meiklejohn. 2016. Centrally banked cryptocurrencies. *NDSS Symposium* (2016).

[27] Alexandra Dmitrienko, David Noack, and Moti Yung. 2017. Secure Wallet-Assisted Offline Bitcoin Payments with Double-Spender Revocation *(ASIA CCS '17)*. Association for Computing Machinery, New York, NY, USA, 520–531. https://doi.org/10.1145/3052973.3052980

[28] Yevgeniy Dodis and Aleksandr Yampolskiy. 2005. A Verifiable Random Function with Short Proofs and Keys. In *Public Key Cryptography - PKC 2005, 8th International Workshop on Theory and Practice in Public Key Cryptography, Les Diablerets, Switzerland, January 23-26, 2005, Proceedings (Lecture Notes in Computer Science, Vol. 3386)*, Serge Vaudenay (Ed.). Springer, 416–431. https://doi.org/10.1007/978-3-540-30580-4_28

[29] Sebastian Faust, Markulf Kohlweiss, Giorgia Azzurra Marson, and Daniele Venturi. 2012. On the non-malleability of the Fiat-Shamir transform. In *Progress in Cryptology-INDOCRYPT 2012*. Springer, 60–79.

[30] Amos Fiat and Adi Shamir. 1987. How to Prove Yourself: Practical Solutions to Identification and Signature Problems. In *CRYPTO '86*, Andrew M. Odlyzko (Ed.), Vol. 263. Springer Verlag, 186–194.

[31] Oded Goldreich, Shafi Goldwasser, and Silvio Micali. 1986. How to construct random functions. *J. ACM* 33, 4 (1986), 792–807. https://doi.org/10.1145/6490.6503

[32] Shafi Goldwasser, Silvio Micali, and Ronald L. Rivest. 1988. A Digital Signature Scheme Secure Against Adaptive Chosen-Message Attacks. *SIAM J. Comput.* 17, 2 (1988), 281–308.

[33] Matthew Green and Ian Miers. 2017. Bolt: Anonymous Payment Channels for Decentralized Currencies. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017*, Bhavani Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu (Eds.). ACM, 473–489. https://doi.org/10.1145/3133956.3134093

[34] Jens Groth. 2021. Non-interactive distributed key generation and key resharing. Cryptology ePrint Archive, Report 2021/339. https://eprint.iacr.org/2021/339.

[35] Jens Groth, Rafail Ostrovsky, and Amit Sahai. 2012. New Techniques for Noninteractive Zero-Knowledge. *J. ACM* 59, 3 (2012), 11:1–11:35. https://doi.org/10.1145/2220357.2220358

[36] Jens Groth and Amit Sahai. 2008. Efficient Non-interactive Proof Systems for Bilinear Groups. In *Advances in Cryptology - EUROCRYPT 2008, 27th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Istanbul, Turkey, April 13-17, 2008. Proceedings (Lecture Notes in Computer Science, Vol. 4965)*, Nigel P. Smart (Ed.). Springer, 415–432. https://doi.org/10.1007/978-3-540-78967-3_24

[37] George Kappos, Haaroon Yousaf, Ania M. Piotrowska, Sanket Kanjalkar, Sergi Delgado-Segura, Andrew Miller, and Sarah Meiklejohn. 2021. An Empirical Analysis of Privacy in the Lightning Network. In *Financial Cryptography and Data Security - 25th International Conference, FC 2021, Virtual Event, March 1-5, 2021, Revised Selected Papers, Part I (Lecture Notes in Computer Science, Vol. 12674)*, Nikita Borisov and Claudia Díaz (Eds.). Springer, 167–186. https://doi.org/10.1007/978-3-662-64322-8_8

[38] Aniket Kate, Yizhou Huang, and Ian Goldberg. 2012. Distributed Key Generation in the Wild. *IACR Cryptol. ePrint Arch.* 2012 (2012), 377. http://eprint.iacr.org/2012/377

[39] Hyoseung Kim, Youngkyung Lee, Michel Abdalla, and Jong Hwan Park. 2021. Practical dynamic group signature with efficient concurrent joins and batch verifications. *J. Inf. Secur. Appl.* 63 (2021), 103003. https://doi.org/10.1016/j.jisa.2021.103003

[40] Hyoseung Kim, Olivier Sanders, Michel Abdalla, and Jong Hwan Park. 2021. Practical Dynamic Group Signatures Without Knowledge Extractors. *IACR Cryptol. ePrint Arch.* (2021), 351. https://eprint.iacr.org/2021/351

[41] James Lovejoy, Cory Fields, Madars Virza, Tyler Frederick, David Urness, Kevin Karwaski, Anders Brownworth, and Neha Narula. 2022. A high performance payment processing system designed for central bank digital currencies. *Cryptology ePrint Archive* (2022).

[42] Wesley C Marshall and Louis-Philippe Rochon. 2009. Financing economic development in Latin America: the Banco del Sur. *Journal of Post Keynesian Economics* 32, 2 (2009), 185–198.

[43] Ian Miers, Christina Garman, Matthew Green, and Aviel D. Rubin. 2013. Zerocoin: Anonymous Distributed E-Cash from Bitcoin. In *2013 IEEE Symposium on Security and Privacy, SP 2013, Berkeley, CA, USA, May 19-22, 2013*. IEEE Computer Society, 397–411. https://doi.org/10.1109/SP.2013.34

[44] Shen Noether and Adam Mackenzie. 2016. Ring Confidential Transactions. *Ledger* 1 (2016), 1–18. https://ledgerjournal.org/ojs/index.php/ledger/article/view/34

[45] Tatsuaki Okamoto and Kazuo Ohta. 1989. Disposable Zero-Knowledge Authentications and Their Applications to Untraceable Electronic Cash. In *Advances in Cryptology - CRYPTO '89, 9th Annual International Cryptology Conference, Santa Barbara, California, USA, August 20-24, 1989, Proceedings (Lecture Notes in Computer Science, Vol. 435)*, Gilles Brassard (Ed.). Springer, 481–496. https://doi.org/10.1007/0-387-34805-0_43

[46] Tatsuaki Okamoto and Kazuo Ohta. 1991. Universal Electronic Cash. In *Advances in Cryptology - CRYPTO '91, 11th Annual International Cryptology Conference, Santa Barbara, California, USA, August 11-15, 1991, Proceedings (Lecture Notes in Computer Science, Vol. 576)*, Joan Feigenbaum (Ed.). Springer, 324–337. https://doi.org/10.1007/3-540-46766-1_27

[47] Torben P. Pedersen. 1991. Non-Interactive and Information-Theoretic Secure Verifiable Secret Sharing. In *CRYPTO (Lecture Notes in Computer Science, Vol. 576)*, Joan Feigenbaum (Ed.). Springer, 129–140.

[48] David Pointcheval and Olivier Sanders. 2016. Short Randomizable Signatures. In *Topics in Cryptology - CT-RSA 2016 - The Cryptographers' Track at the RSA Conference 2016, San Francisco, CA, USA, February 29 - March 4, 2016, Proceedings (Lecture Notes in Computer Science, Vol. 9610)*, Kazue Sako (Ed.). Springer, 111–126. https://doi.org/10.1007/978-3-319-29485-8_7

[49] David Pointcheval, Olivier Sanders, and Jacques Traoré. 2017. Cut Down the Tree to Achieve Constant Complexity in Divisible E-cash. In *Public-Key Cryptography - PKC 2017 - 20th IACR International Conference on Practice and Theory in Public-Key Cryptography, Amsterdam, The Netherlands, March 28-31, 2017, Proceedings, Part I (Lecture Notes in Computer Science, Vol. 10174)*, Serge Fehr (Ed.). Springer, 61–90. https://doi.org/10.1007/978-3-662-54365-8_4

[50] Alfredo Rial and Ania M. Piotrowska. 2022. Security Analysis of Coconut, an Attribute-Based Credential Scheme with Threshold Issuance. Cryptology ePrint Archive, Report 2022/011. https://ia.cr/2022/011.

[51] Eli Ben Sasson, Alessandro Chiesa, Christina Garman, Matthew Green, Ian Miers, Eran Tromer, and Madars Virza. 2014. Zerocash: Decentralized anonymous payments from bitcoin. In *2014 IEEE symposium on security and privacy*. IEEE, 459–474.

[52] Jeffrey Shallit. 2003. What this country needs is an 18c piece. *Mathematical Intelligencer* 25, 2 (2003), 20–23.

[53] Piyush Kumar Sharma, Devashish Gosain, and Claudia Diaz. 2022. On the anonymity of peer-to-peer network anonymity schemes used by cryptocurrencies. *arXiv preprint arXiv:2201.11860* (2022).

[54] Alberto Sonnino, Mustafa Al-Bassam, Shehar Bano, Sarah Meiklejohn, and George Danezis. 2019. Coconut: Threshold Issuance Selective Disclosure Credentials with Applications to Distributed Ledgers. In *26th Annual Network and Distributed System Security Symposium, NDSS 2019, San Diego, California, USA, February 24-27, 2019*. The Internet Society. https://www.ndss-symposium.org/ndss-paper/coconut-threshold-issuance-selective-disclosure-credentials-with-applications-to-distributed-ledgers/

[55] Nicolas van Saberhagen. 2013. Cryptonote v2.0. https://cryptonote.org/whitepaper.pdf.

[56] Douglas Wikström. 2004. A Universally Composable Mix-Net. In *Theory of Cryptography, First Theory of Cryptography Conference, TCC 2004, Cambridge, MA, USA, February 19-21, 2004, Proceedings (Lecture Notes in Computer Science, Vol. 2951)*, Moni Naor (Ed.). Springer, 317–335. https://doi.org/10.1007/978-3-540-24638-1_18

[57] Jianguo Xu. 2022. Developments and implications of central bank digital currency: The case of China e-CNY. *Asian Economic Policy Review* 17, 2 (2022), 235–250.

[58] Maofan Yin, Dahlia Malkhi, Michael K Reiter, Guy Golan Gueta, and Ittai Abraham. 2019. HotStuff: BFT consensus with linearity and responsiveness. In *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing*. 347–356.

[59] Alexei Zamyatin, Mustafa Al-Bassam, Dionysis Zindros, Eleftherios Kokoris-Kogias, Pedro Moreno-Sanchez, Aggelos Kiayias, and William J Knottenbelt. 2021. Sok: Communication across distributed ledgers. In *Financial Cryptography and Data Security: 25th International Conference, FC 2021, Virtual Event, March 1–5, 2021, Revised Selected Papers, Part II 25*. Springer, 3–36.

A BUILDING BLOCKS

A.1 Bilinear Maps

Let $\mathbb{G}, \tilde{\mathbb{G}}$ and $\mathbb{G}_t$ be groups of prime order p. A map $e : \mathbb{G} \times \tilde{\mathbb{G}} \to \mathbb{G}_t$ must satisfy bilinearity, i.e., $e(g^x, \tilde{g}^y) = e(g, \tilde{g})^{xy}$; non-degeneracy, i.e., for all generators $g \in \mathbb{G}$ and $\tilde{g} \in \tilde{\mathbb{G}}$, $e(g, \tilde{g})$ generates $\mathbb{G}_t$; and efficiency, i.e., there exists an efficient algorithm $\mathcal{G}(1^k)$ that outputs the pairing group setup $(p, \mathbb{G}, \tilde{\mathbb{G}}, \mathbb{G}_t, e, g, \tilde{g})$ and an efficient algorithm to compute $e(a, b)$ for any $a \in \mathbb{G}, b \in \tilde{\mathbb{G}}$. In type 3 pairings, $\mathbb{G} \neq \tilde{\mathbb{G}}$ and there exists no efficiently computable homomorphism $f : \tilde{\mathbb{G}} \to \mathbb{G}$.

A.2 Assumptions

We recall the assumptions that are needed to prove the security of our schemes.

Definition A.1. [XDH and SXDH Assumptions [3]] Given $(p, \mathbb{G}, \tilde{\mathbb{G}}, \mathbb{G}_t, e, g, \tilde{g})$, the external Diffie-Hellman assumption states that the decisional Diffie-Hellman problem is intractable in $\mathbb{G}$ or in $\tilde{\mathbb{G}}$. The symmetric external Diffie-Hellman assumption states that it is intractable in both $\mathbb{G}$ and $\tilde{\mathbb{G}}$.

Definition A.2. [q-SDH Assumption [9]] Given $(g, g^x, g^{x^2}, \ldots, g^{x^q}) \in \mathbb{G}^{q+1}$, the strong Diffie-Hellman assumption states that it is hard to output a pair $(m, g^{1/(x+m)}) \in \mathbb{Z}_p \times \mathbb{G}$.

Definition A.3. [y-DDHI Assumption [14]] Let $\mathbb{G}$ be a group of prime order q and let g be a generator of $\mathbb{G}$. Given $(g, g^x, \ldots, g^{(x^y)}, R)$ for a random $x \leftarrow \mathbb{Z}_p$, the decisional Diffie-Hellman inversion assumption states that it is hard to decide if $R = g^{1/x}$ or not.

Definition A.4. [N-BDHI Assumption [8]] Given $(p, \mathbb{G}, \tilde{\mathbb{G}}, \mathbb{G}_t, e, g, \tilde{g})$ and the tuple $(\{g^{y^i}\}_{i=0}^N, \{\tilde{g}^{y^i}\}_{i=0}^N) \in \mathbb{G}^{N+1} \times \tilde{\mathbb{G}}^{N+1}$, the bilinear Diffie-Hellman inversion assumption states that it is hard to compute $e(g, \tilde{g})^{1/y} \in \mathbb{G}_t$.

Definition A.5. [N-MXDH' Assumption [49]] $\forall N \in \mathbb{N}^*$, we define $C = N^3 - N^2$, $S = C + 1$, $E = N^2 - N$, $D = S + E$, and $P = D + C$. Given $(p, \mathbb{G}, \tilde{\mathbb{G}}, \mathbb{G}_t, e, g, \tilde{g})$ and $\{(g^{\gamma^k}, h^{\gamma^k})_{k=0}^P, (g^{\alpha\delta\gamma^{-k}}, h^{\alpha\delta\gamma^{-k}})_{k=0}^E, (g^{\chi\gamma^k}, h^{\chi\gamma^k})_{k=D+1}^P, (g^{\alpha\gamma^{-k}}, g^{\chi\gamma^k/\alpha}, h^{\chi\gamma^k/\alpha})_{k=0}^C\} \in \mathbb{G}^{2P+5S+2E+2}$, as well as $(\tilde{g}^{\gamma^k}, \tilde{g}^{\alpha\gamma^{-k}})_{k=0}^C \in \tilde{\mathbb{G}}^{2S}$ and a pair $(g^{z_1}, h^{z_2}) \in \mathbb{G}^2$, it is hard to decide whether $z_1 = z_2 = \delta + \chi\gamma^D/\alpha$ or (z_1, z_2) is random.

Definition A.6. [n-DHE [15]] Let $(p, \mathbb{G}, \tilde{\mathbb{G}}, \mathbb{G}_t, e, g, \tilde{g}) \leftarrow \mathcal{G}(1^k)$ and $\alpha \leftarrow \mathbb{Z}_p$. Given $(p, \mathbb{G}, \tilde{\mathbb{G}}, \mathbb{G}_t, e, g, \tilde{g})$ and a tuple $(g_1, \tilde{g}_1, \ldots, g_n, \tilde{g}_n, g_{n+2}, \ldots, g_{2n})$ such that $g_i = g^{(\alpha^i)}$ and $\tilde{g}_i = \tilde{g}^{(\alpha^i)}$, for any p.p.t. adversary $\mathcal{A}$, $\Pr[g^{(\alpha^{n+1})} \leftarrow \mathcal{A}(p, \mathbb{G}, \tilde{\mathbb{G}}, \mathbb{G}_t, e, g, \tilde{g}, g_1, \tilde{g}_1, \ldots, g_n, \tilde{g}_n, g_{n+2}, \ldots, g_{2n})] \leq \epsilon(k)$.

A.3 Zero-Knowledge Arguments of Knowledge

Informally speaking, a zero-knowledge argument of knowledge is a two-party protocol between a prover and a verifier with two properties. First, it should be a proof of knowledge, i.e., there should exist a knowledge extractor that extracts the secret input from a successful prover with all but negligible probability. Second, it should be zero-knowledge, i.e., for all possible verifiers there exists a simulator that, without knowledge of the secret input, yields a distribution that cannot be distinguished from the interaction with a real prover.

To express a zero-knowledge argument of knowledge, we follow the notation introduced by Camenisch and Stadler [17], i.e., we denote as $\mathsf{ZK}\{(w) : y = f(w)\}$ a "*zero-knowledge proof of knowledge of the secret input w such that $y = f(w)$*", where w is a secret input, while y and the function f are publicly known.

Let $\mathcal{L}$ be a language in NP. We can associate to any NP-language $\mathcal{L}$ a polynomial time recognizable relation $\mathcal{R}_{\mathcal{L}}$ defining $\mathcal{L}$ as $\mathcal{L} = \{x : \exists w \text{ s.t. } (x, w) \in \mathcal{R}_{\mathcal{L}}\}$, where $|w| \leq \text{poly}(|x|)$. The string w is called a witness for membership of $x \in \mathcal{L}$.

A protocol $\Sigma = (\mathcal{P}, \mathcal{V})$ for an NP-language $\mathcal{L}$ is an interactive proof system. The prover $\mathcal{P}$ and the verifier $\mathcal{V}$ know an instance x of the language $\mathcal{L}$. The prover $\mathcal{P}$ also knows a witness w for membership of $x \in \mathcal{L}$. Σ-protocols have a 3-move shape where the first message α, called *commitment*, is sent by the prover. The second message β, called *challenge*, is chosen randomly and sent by the verifier. The last message γ, called *response*, is sent by the prover. A Σ-protocol fulfills the properties of completeness, honest-verifier zero-knowledge, and special soundness defined in Faust et al. [29].

In our e-cash schemes, zero-knowledge arguments of knowledge based on the Fiat-Shamir transform [30] are used. The Fiat-Shamir transform removes the interaction between the prover $\mathcal{P}$ and the verifier $\mathcal{V}$ of a Σ protocol by replacing the challenge with a hash value $H(\alpha, x)$ computed by the prover, where H is modeled as a random oracle. (It is possible to include an additional message m as input to H, i.e. $H(\alpha, x, m)$, turning the argument of knowledge into a *signature of knowledge* of the message m.) An argument π consists of $(\alpha, H(\alpha, x), \gamma)$. The Fiat-Shamir system is denoted by $(\mathcal{P}^H, \mathcal{V}^H)$ and fulfills the properties of zero-knowledge and weak simulation extractability defined in Faust et al. [29], which we recall below.

Definition A.7 (Zero-Knowledge). Define the zero knowledge simulator $\mathcal{S}$ as follows. $\mathcal{S}$ is a stateful algorithm that can operate in two modes: $(h_i, st) \leftarrow \mathcal{S}(1, st, q_i)$ answers random oracle queries q_i, while $(\pi, st) \leftarrow \mathcal{S}(2, st, x)$ outputs a simulated proof π for an instance x. $\mathcal{S}(1, \cdots)$ and $\mathcal{S}(2, \cdots)$ share the state st that is updated after each operation.

Let $\mathcal{L}$ be a language in NP. Denote with $(\mathcal{S}_1, \mathcal{S}_2)$ the oracles such that $\mathcal{S}_1(q_i)$ returns the first output of $(h_i, st) \leftarrow \mathcal{S}(1, st, q_i)$ and $\mathcal{S}_2(x, w)$ returns the first output of $(\pi, st) \leftarrow \mathcal{S}(2, st, x)$ if $(x, w) \in \mathcal{R}_{\mathcal{L}}$. A protocol $(\mathcal{P}^H, \mathcal{V}^H)$ is a non-interactive zero-knowledge proof for the language $\mathcal{L}$ in the random oracle model if there exists a ppt simulator $\mathcal{S}$ such that for all ppt distinguishers $\mathcal{D}$ we have

$$\Pr[\mathcal{D}^{H(\cdot), \mathcal{P}^H(\cdot,\cdot)}(1^k) = 1] \approx \Pr[\mathcal{D}^{\mathcal{S}_1(\cdot), \mathcal{S}_2(\cdot,\cdot)}(1^k) = 1],$$

where both $\mathcal{P}$ and $\mathcal{S}_2$ oracles output $\perp$ if $(x, w) \notin \mathcal{R}_{\mathcal{L}}$.

Definition A.8 (Weak Simulation Extractability). Let $\mathcal{L}$ be a language in NP. Consider a non-interactive zero-knowledge proof system $(\mathcal{P}^H, \mathcal{V}^H)$ for $\mathcal{L}$ with zero-knowledge simulator $\mathcal{S}$. Let $(\mathcal{S}_1, \mathcal{S}_2')$ be oracles returning the first output of $(h_i, st) \leftarrow \mathcal{S}(1, st, q_i)$ and $(\pi, st) \leftarrow \mathcal{S}(2, st, x)$ respectively. $(\mathcal{P}^H, \mathcal{V}^H)$ is weakly simulation extractable with extraction error ν and with respect to $\mathcal{S}$ in the random oracle model, if for all ppt adversaries $\mathcal{A}$ there exists an efficient algorithm $\mathcal{E}_{\mathcal{A}}$ with access to the answers $(\mathcal{T}_H, \mathcal{T})$ of $(\mathcal{S}_1, \mathcal{S}_2')$ respectively such that the following holds. Let

$$\begin{aligned}
&\text{acc} = \Pr[(x^*, \pi^*) \leftarrow \mathcal{A}^{\mathcal{S}_1(\cdot), \mathcal{S}_2'(\cdot)}(1^k; \rho) : \\
&(x^*, \pi^*) \notin \mathcal{T}; \mathcal{V}^{\mathcal{S}_1}(x^*, \pi^*) = 1] \\
&\text{ext} = \Pr[(x^*, \pi^*) \leftarrow \mathcal{A}^{\mathcal{S}_1(\cdot), \mathcal{S}_2'(\cdot)}(1^k; \rho); \\
&\quad w^* \leftarrow \mathcal{E}_{\mathcal{A}}(x^*, \pi^*; \rho, \mathcal{T}_H, \mathcal{T}) : (x^*, \pi^*) \notin \mathcal{T}; (x^*, w^*) \in \mathcal{R}_{\mathcal{L}}],
\end{aligned}$$

where the probability space in both cases is over the random choices of $\mathcal{S}$ and the adversary's random tape ρ. Then, there exists a constant $d > 0$ and a polynomial p such that whenever $\text{acc} \geq \nu$, we have $\text{ext} \geq (1/p)(\text{acc} - \nu)^d$.

Types of proofs. We use known results for computing ZK proofs of discrete logarithms [17]. A protocol proving knowledge of exponents $(w_1, \ldots, w_n)$ that satisfy the formula $\phi(w_1, \ldots, w_n)$ is described as

$$\mathsf{ZK}\{(w_1, \ldots, w_n) : 1 = \phi(w_1, \ldots, w_n)\} \tag{1}$$

The formula $\phi(w_1, \ldots, w_n)$ consists of conjunctions and disjunctions of "atoms". An atom expresses *group relations*, such as

$$\prod_{j=1}^{k} g_j^{f_j} = 1$$

where the g_j's are elements of prime order groups and the f_j's are polynomials in the variables $(w_1, \ldots, w_n)$.

A proof system for (1) can be transformed into a proof system for more expressive statements about secret exponents *sexps* and secret bases *sbases*:

$$\mathsf{ZK}\{(sexps, sbases) : 1 = \phi(sexps, bases \cup sbases)\} \tag{2}$$

The transformation adds an additional base h to the public bases. For each $g_j \in sbases$, the transformation picks a random exponent ρ_j and computes a blinded base $g_j' = g_j h^{\rho_j}$. The transformation adds g_j' to the public bases *bases*, ρ_j to the secret exponents *sexps*, and rewrites $g_j^{f_j}$ into $g_j'^{f_j} h^{-f_j \rho_j}$.

The proof system supports pairing product equations

$$\prod_{j=1}^{k} e(g_j, \tilde{g}_j)^{f_j} = 1 \tag{3}$$

in groups of prime order with a bilinear map e, by treating the target group $\mathbb{G}_t$ as the group of the proof system. The embedding for secret bases is unchanged, except for the case in which both bases in a pairing are secret. In this case, $e(g_j, \tilde{g}_j)^{f_j}$ must be transformed into $e(g_j', \tilde{g}_j')^{f_j} e(g_j', \tilde{h})^{-f_j \tilde{\rho}_j} e(h, \tilde{g}_j')^{-f_j \rho_j} e(h, \tilde{h})^{f_j \rho_j \tilde{\rho}_j}$.

A.4 Commitment Schemes

A commitment scheme consists of algorithms CSetup, Com and VfCom. The algorithm $\text{CSetup}(1^k)$ generates the parameters of the

commitment scheme par_c, which include a description of the message space $\mathcal{M}$. $\mathsf{Com}(par_c, x)$ outputs a commitment com to x and auxiliary information $open$. A commitment is opened by revealing $(x, open)$ and checking whether $\mathsf{VfCom}(par_c, com, x, open)$ outputs 1 or 0.

A commitment scheme should fulfill the *correctness*, *hiding* and *binding* properties. We recall the definitions of those properties below.

Definition A.9 (Correctness). Correctness requires that VfCom accepts all commitments created by algorithm Com, i.e., for all $x \in \mathcal{M}$

$$\Pr\left[\begin{array}{c} par_c \leftarrow \mathsf{CSetup}(1^k);\ (com, open) \leftarrow \mathsf{Com}(par_c, x): \\ 1 = \mathsf{VfCom}(par_c, com, x, open) \end{array}\right] = 1\ .$$

Definition A.10 (Hiding Property). The hiding property ensures that a commitment com to x does not reveal any information about x. For any ppt adversary $\mathcal{A}$, the hiding property is defined as follows:

$$\Pr\left[\begin{array}{l} par_c \leftarrow \mathsf{CSetup}(1^k); \\ (x_0, x_1, st) \leftarrow \mathcal{A}(par_c); \\ b \leftarrow \{0,1\};\ (com, open) \leftarrow \mathsf{Com}(par_c, x_b); \\ b' \leftarrow \mathcal{A}(st, com): \\ x_0 \in \mathcal{M}\ \wedge\ x_1 \in \mathcal{M}\ \wedge\ b = b' \end{array}\right] \leq \frac{1}{2} + \epsilon(k)\ .$$

Definition A.11 (Binding Property). The binding property ensures that com cannot be opened to another value x'. For any ppt adversary $\mathcal{A}$, the binding property is defined as follows:

$$\Pr\left[\begin{array}{l} par_c \leftarrow \mathsf{CSetup}(1^k); \\ (com, x, open, x', open') \leftarrow \mathcal{A}(par_c): \\ x \in \mathcal{M}\ \wedge\ x' \in \mathcal{M}\ \wedge x \neq x' \\ \wedge\ 1 = \mathsf{VfCom}(par_c, com, x, open) \\ \wedge\ 1 = \mathsf{VfCom}(par_c, com, x', open') \end{array}\right] \leq \epsilon(k)\ .$$

Our e-cash schemes use the commitment scheme by Pedersen [47] to commit to elements $x \in \mathbb{Z}_p$, where p is a prime. This commitment scheme is perfectly hiding and computationally binding under the discrete logarithm assumption. The Pedersen commitment scheme consists of the following algorithms.

- $\mathsf{CSetup}(1^k)$. On input the security parameter 1^k, pick random generators g, h of a group $\mathbb{G}_p$ of prime order p. Output $par_c = (g, h, \mathcal{M})$, where $\mathcal{M} = \mathbb{Z}_p$.
- $\mathsf{Com}(par_c, x)$. Check that $x \in \mathcal{M}$. Pick random value $open \in \mathbb{Z}_p$, compute $com = g^{open} h^x$, and output com.
- $\mathsf{VfCom}(par_c, com, x', open')$. Recompute $com' = g^{open'} h^{x'}$. If $com = com'$ then output 1 else 0.

When committing to a tuple of messages, the Pedersen commitment scheme works as follows.

- $\mathsf{CSetup}(1^k, l)$. On input the security parameter 1^k and an upper bound l on the number of elements to be committed, pick $l+1$ random generators $h_1, \ldots, h_l, g$ of a group $\mathbb{G}_p$ of prime order p. Output $par_c = (h_1, \ldots, h_l, g, \mathcal{M})$, where $\mathcal{M} = \mathbb{Z}_p^l$.
- $\mathsf{Com}(par_c, \langle x_1, \ldots, x_l \rangle)$. Pick random value $open \leftarrow \mathbb{Z}_p$, compute $com = g^{open} \prod_{i=1}^{l} h_i^{x_i}$ and output com.
- $\mathsf{VfCom}(par_c, com, \langle x'_1, \ldots, x'_l \rangle, open')$. Recompute commitment $com' = g^{open'} \prod_{i=1}^{l} h_i^{x'_i}$. If it is the case that $com = com'$ then output 1 else 0.

A.5 Signature Schemes

A signature scheme consists of the algorithms KeyGen, Sign, and VfSig. $\mathsf{KeyGen}(1^k)$ outputs a secret key sk and a public key pk, which include a description of the message space $\mathcal{M}$. $\mathsf{Sign}(sk, m)$ outputs a signature σ on message $m \in \mathcal{M}$. $\mathsf{VfSig}(pk, \sigma, m)$ outputs 1 if σ is a valid signature on m and 0 otherwise. This definition can be extended to blocks of messages $(m_1, \ldots, m_q)$. In this case, $\mathsf{KeyGen}(1^k, q)$ receives the maximum number of messages as input. A signature scheme must fulfill the correctness and existential unforgeability properties [32], which we recall below.

Definition A.12 (Correctness). Correctness ensures that the algorithm VfSig accepts the signatures created by the algorithm Sign on input a secret key computed by algorithm KeyGen. More formally, correctness is defined as follows.

$$\Pr\left[\begin{array}{l} (sk, pk) \leftarrow \mathsf{KeyGen}(1^k);\ m \leftarrow \mathcal{M}; \\ \sigma \leftarrow \mathsf{Sign}(sk, m):\ 1 = \mathsf{VfSig}(pk, \sigma, m) \end{array}\right] = 1$$

Definition A.13 (Existential Unforgeability). The property of existential unforgeability ensures that it is not feasible to output a signature on a message without knowledge of the secret key or of another signature on that message. Let O_s be an oracle that, on input sk and a message $m \in \mathcal{M}$, outputs $\mathsf{Sign}(sk, m)$, and let S_s be a set that contains the messages sent to O_s. More formally, for any ppt adversary $\mathcal{A}$, existential unforgeability is defined as follows.

$$\Pr\left[\begin{array}{l} (sk, pk) \leftarrow \mathsf{KeyGen}(1^k);\ (m, \sigma) \leftarrow \mathcal{A}(pk)^{O_s(sk, \cdot)}: \\ 1 = \mathsf{VfSig}(pk, \sigma, m)\ \wedge\ m \in \mathcal{M}\ \wedge\ m \notin S_s \end{array}\right] \leq \epsilon(k)$$

Pointcheval-Sanders (PS) signatures. The PS signature scheme is defined as follows [48].

$\mathsf{KeyGen}(1^k, q)$. Run $\mathcal{G}(1^k)$ to obtain a pairing group setup $\theta = (p, \mathbb{G}, \tilde{\mathbb{G}}, \mathbb{G}_t, e, g, \tilde{g})$. Pick random secret key $(x, y_1, \ldots, y_q) \leftarrow \mathbb{Z}_p^{q+1}$. Output the secret key $sk = (\theta, x, y_1, \ldots y_q)$ and the public key $pk = (\theta, \tilde{\alpha}, \beta_1, \tilde{\beta}_1, \ldots, \beta_q, \tilde{\beta}_q) \leftarrow (\theta, \tilde{g}^x, g^{y_1}, \tilde{g}^{y_1}, \ldots, g^{y_q}, \tilde{g}^{y_q})$.

$\mathsf{Sign}(sk, m_1, \ldots, m_q)$. Parse sk as $(\theta, x, y_1, \ldots, y_q)$. Pick up random $r \leftarrow \mathbb{Z}_p$ and set $h \leftarrow g^r$. Output the signature $\sigma = (h, s) \leftarrow (h, h^{x + y_1 m_1 + \ldots + y_q m_q})$.

$\mathsf{VfSig}(pk, \sigma, m_1, \ldots, m_q)$. Output 1 if $e(h, \tilde{\alpha} \prod_{j=1}^{q} \tilde{\beta}_j^{m_j}) = e(s, \tilde{g})$ and $h \neq 1$. Otherwise output 0.

This signature scheme is randomizable. To randomize a signature $\sigma = (h, s)$, pick random $r' \leftarrow \mathbb{Z}_p$ and compute $\sigma' = (h^{r'}, s^{r'})$. The elements $(\beta_1, \ldots, \beta_q)$ in the public key are needed for the blind signature issuance protocol in [48], as well as for the issuance protocols of Coconut and of our e-cash schemes.

Pointcheval-Sanders signatures in the random oracle model. Coconut and our e-cash schemes use a variant of PS signatures in which, in algorithm Sign, the random generator h is computed via a hash function, which is modeled as a random oracle (RO). This variant has been formalized in [50] as PS signatures in the RO model.

In [50], the syntax of algorithm Sign is as follows. Sign uses a random oracle $H : \mathcal{M}_{ro} \rightarrow \mathcal{S}$. $\mathsf{Sign}(sk, m_1, \ldots m_q, r, st)$ receives as input a secret key sk, a tuple of messages $(m_1, \ldots, m_q)$, a value $r \in \mathcal{M}_{ro}$ and state information st, which stores tuples of the form $(m_1, \ldots, m_q, r)$. Sign outputs a signature σ on $(m_1, \ldots, m_q)$ if st does

not contain a tuple $(m'_1, \ldots, m'_q, r')$ such that $(m_1, \ldots, m_q) \neq (m'_1, \ldots, m'_q)$ and $r = r'$. Sign also outputs updated state information st'.

The PS signature scheme in the RO model scheme works as follows. The algorithms KeyGen and VfSig remain unmodified.

Sign$(sk, m_1, \ldots, m_q, r, st)$. Parse sk as $(\theta, x, y_1, \ldots, y_q)$. If st contains a tuple $(m'_1, \ldots, m'_q, r')$ such that $(m_1, \ldots, m_q) \neq (m'_1, \ldots, m'_q)$ and $r = r'$, output $\sigma = \perp$ and $st' = st$. Otherwise compute $h \leftarrow H(r)$ and output the signature $\sigma = (h, s) \leftarrow (h, h^{x+y_1 m_1 + \ldots + y_q m_q})$ and the updated state information $st' = st \cup \{(m_1, \ldots, m_q, r)\}$.

We recall the definition of the existential unforgeability property in the RO model below.

Definition A.14 (Existential Unforgeability in the RO [50]). For any ppt adversary $\mathcal{A}$, existential unforgeability in the RO model is defined as follows.

$$\Pr\left[\begin{array}{l}(sk, pk) \leftarrow \mathsf{KeyGen}(1^k);\\ (m, \sigma) \leftarrow \mathcal{A}(pk)^{O_s(sk,\cdot,\cdot),H(\cdot)}:\\ 1 = \mathsf{VfSig}(pk, \sigma, m) \ \wedge \ m \in \mathcal{M} \ \wedge \ m \notin S_s\end{array}\right] \leq \epsilon(k)$$

$O_s(sk, \cdot, \cdot)$ works as follows. On input sk, a message $m = (m_1, \ldots, m_q)$ and the value r, O_s runs $(\sigma, st') \leftarrow \mathsf{Sign}(sk, m_1, \ldots, m_q, r, st)$. O_s replaces st by st' and returns σ to $\mathcal{A}$. (st is empty in the first invocation of O_s.) S_s is a set that contains the messages sent to O_s.

In comparison to the definition of existential unforgeability (see Definition A.13), $\mathcal{A}$ has access to the random oracle H, and the signing oracle is modified to follow the new syntax. The PS scheme in the RO model is existentially unforgeable under the generalized PS assumption proposed in [39, 40].

Structure-Preserving Signature (SPS) scheme. In a SPS scheme, the public key, the messages, and the signatures are group elements in $\mathbb{G}$ and $\tilde{\mathbb{G}}$, and verification must consist purely in the checking of pairing product equations. Our divisible e-cash scheme uses the SPS scheme in [1]. In this SPS scheme, a elements in $\mathbb{G}$ and b elements in $\tilde{\mathbb{G}}$ are signed.

KeyGen(grp, a, b). Let $grp \leftarrow (p, \mathbb{G}, \tilde{\mathbb{G}}, \mathbb{G}_t, e, g, \tilde{g})$ be the bilinear map parameters. Pick at random $u_1, \ldots, u_b, y, w_1, \ldots w_a, z \leftarrow \mathbb{Z}_p^*$ and compute $U_i = g^{u_i}$, $i \in [1..b]$, $Y = \tilde{g}^y$, $W_i = \tilde{g}^{w_i}$, $i \in [1..a]$ and $Z = \tilde{g}^z$. Return the verification key $pk \leftarrow (grp, U_1, \ldots, U_b, Y, W_1, \ldots, W_a, Z)$ and the signing key $sk \leftarrow (pk, u_1, \ldots, u_b, y, w_1, \ldots, w_a, z)$.

Sign$(sk, \langle m_1, \ldots, m_{a+b} \rangle)$. Pick $r \leftarrow \mathbb{Z}_p^*$, and set

$$R \leftarrow g^r, \quad S \leftarrow g^{z - ry} \prod_{i=1}^{a} m_i^{-w_i}, \quad T \leftarrow (\tilde{g} \prod_{i=1}^{b} m_{a+i}^{-u_i})^{1/r},$$

and output the signature $\sigma \leftarrow (R, S, T)$.

VfSig$(pk, \sigma, \langle m_1, \ldots, m_{a+b} \rangle)$. Output 1 if it is satisfied that

$$e(R, Y) e(S, \tilde{g}) \prod_{i=1}^{a} e(m_i, W_i) = e(g, Z)$$

and

$$e(R, T) \prod_{i=1}^{b} e(U_i, m_{a+i}) = e(g, \tilde{g})$$

A.6 Pseudorandom Functions

Pseudorandom functions (PF) [11, 31] are a family of indexed functions $F = \{F_s\}$ such that: (1) given the index s, F_s can be efficiently evaluated on all inputs; (2) no probabilistic polynomial-time algorithm without s can distinguish evaluations $F_s(x_i)$ for inputs x_i of its choice from random values. We recall the definition of pseudorandom functions in [11].

Definition A.15 (Pseudorandom Function Family). A family of functions $\mathcal{F} = \{F_s\}_{s \in S}$, indexed by a set S, and where $F_s : D \rightarrow R$ for all s, is a pseudorandom function (PRF) family if for a randomly chosen s, and all PPT $\mathcal{A}$, the distinguishing advantage $\Pr_{s \leftarrow S}[\mathcal{A}^{f_s(\cdot)} = 1] - \Pr_{f \leftarrow (D \rightarrow R)}[\mathcal{A}^{f(\cdot)} = 1]$ is negligible, where $(D \rightarrow R)$ denotes the set of all functions from D to R.

Our compact e-cash scheme uses the PF in [14], which works as follows. For every n, a function f is defined by the tuple $(\mathbb{G}, p, g, s)$, where $\mathbb{G}$ is a group of order p, p is an n-bit prime, g is a generator of $\mathbb{G}$, and s is a seed in $\mathbb{Z}_p$. For any input $x \in \mathbb{Z}_p$ (except for $x = -1 \bmod p$), the function $f_{\mathbb{G},p,g,s}(\cdot)$, which we denote as $f_{g,s}$ for fixed values of $(\mathbb{G}, p, g)$, is defined as $f_{g,s}(x) = g^{1/(s+x+1)}$. This PF is secure under the y-DDHI assumption in $\mathbb{G}$, which we recall in §A.2. This PF is based on the verifiable random function in [28], which is secure under the y-DBDHI assumption.

A.7 Notation

In Table 4, we summarize the notation used in our paper.

B IDEAL-WORLD/REAL-WORLD PARADIGM

The security of a protocol φ is analyzed by comparing the view of an environment $\mathcal{Z}$ in a real execution of φ against that of $\mathcal{Z}$ in the ideal protocol defined in $\mathcal{F}_\varphi$. $\mathcal{Z}$ chooses the inputs of the parties and collects their outputs. In the real world, $\mathcal{Z}$ can communicate freely with an adversary $\mathcal{A}$ who controls both the network and any corrupt parties. In the ideal world, $\mathcal{Z}$ interacts with dummy parties, who simply relay inputs and outputs between $\mathcal{Z}$ and $\mathcal{F}_\varphi$, and a simulator $\mathcal{S}$. We say that a protocol φ securely realizes $\mathcal{F}_\varphi$ if $\mathcal{Z}$ cannot distinguish the real world from the ideal world, i.e., $\mathcal{Z}$ cannot distinguish whether it is interacting with $\mathcal{A}$ and parties running protocol φ or with $\mathcal{S}$ and dummy parties relaying to $\mathcal{F}_\varphi$

A protocol $\varphi^{\mathcal{G}}$ securely realizes $\mathcal{F}$ in the $\mathcal{G}$-hybrid model when φ is allowed to invoke the ideal functionality $\mathcal{G}$. Therefore, for any protocol ψ that securely realizes $\mathcal{G}$, the composed protocol φ^{ψ}, which is obtained by replacing each invocation of an instance of $\mathcal{G}$ with an invocation of an instance of ψ, securely realizes $\mathcal{F}$.

In the ideal functionalities described in this paper, we consider static corruptions. When describing ideal functionalities, we use the conventions introduced in [13], which are summarised in B.

Interface Naming Convention. An ideal functionality can be invoked by using one or more interfaces. The name of a message in an interface consists of three fields separated by dots, e.g., ec.setup.ini in $\mathcal{F}_{\text{EC}}$ in §3.3. The first field indicates the name of the functionality and is the same in all interfaces of the functionality. This field is useful for distinguishing between invocations of different functionalities in a hybrid protocol that uses two or more different functionalities. The second field indicates the kind of action performed by the functionality and is the same in all messages that

Table 4: Table of symbols

Symbol	Meaning
Bilinear Maps	
e	Bilinear map
$\mathcal{G}$	Bilinear setup
p	Prime number
$\mathbb{G}$	Group of order p
$\tilde{\mathbb{G}}$	Group of order p
$\mathbb{G}_t$	Group of order p
g	Generator of $\mathbb{G}$
$\tilde{g}$	Generator of $\tilde{\mathbb{G}}$
$\mathbb{Z}_p$	Integers modulo p
Security Definitions	
$\mathcal{A}$	Adversary
$\mathcal{Z}$	Environment
$\mathcal{S}$	Simulator
$\mathcal{F}$	Ideal Functionality
pid	Party identifier
sid	Session identifier
qid	Query identifier
Pr	Probability
$\mathcal{O}$	Oracle
k	Security parameter
ϵ	Negligible function
E-cash	
EC	Threshold issuance offline anonymous e-cash
$\mathcal{U}$	User
$\mathcal{P}$	Provider
$\mathcal{V}$	Authority
n	Number of authorities
t	Threshold
L	Number of coins in a full wallet
V	Number of coins spent in a payment
params	Parameters
$sk_{\mathcal{V}}$	Authority secret key
$pk_{\mathcal{V}}$	Authority public key
$sk_{\mathcal{U}}$	User secret key
$pk_{\mathcal{U}}$	User public key
req	Withdrawal request
res	Withdrawal response
W	Wallet
pay	Payment
payinfo	Payment information
BB	Bulletin Board

the functionality exchanges within the same interface. The third field distinguishes between the messages that belong to the same interface, and can take the following different values. A message ec.setup.ini is the incoming message received by the functionality, i.e., the message through which the interface is invoked. A message ec.setup.end is the outgoing message sent by the functionality, i.e., the message that ends the execution of the interface. The message ec.setup.sim is used by the functionality to send a message to $\mathcal{S}$, and the message ec.setup.rep is used to receive a message from $\mathcal{S}$.

Network vs local communication. The identity of an interactive Turing machine instance (ITI) consists of a party identifier *pid* and a session identifier *sid*. A set of parties in an execution of a system of interactive Turing machines is a protocol instance if they have the same session identifier *sid*. ITIs can pass direct inputs to and outputs from "local" ITIs that have the same *pid*. An ideal functionality $\mathcal{F}$ has $pid = \perp$ and is considered local to all parties. An instance of $\mathcal{F}$ with the session identifier *sid* only accepts inputs from and passes outputs to machines with the same session identifier *sid*. Some functionalities require the session identifier to have some structure. Those functionalities check whether the session identifier possesses the required structure in the first message that invokes the functionality. For the subsequent messages, the functionality implicitly checks that the session identifier equals the session identifier used in the first message. Communication between ITIs with different party identifiers must take place over the network. The network is controlled by $\mathcal{A}$, meaning that he can arbitrarily delay, modify, drop, or insert messages.

Query identifiers. Some interfaces in a functionality can be invoked more than once. When the functionality sends a message ec.setup.sim to $\mathcal{S}$ in such an interface, a query identifier *qid* is included in the message. The query identifier must also be included in the response ec.setup.rep sent by $\mathcal{S}$. The query identifier is used to identify the message ec.setup.sim to which $\mathcal{S}$ replies with a message ec.setup.rep. We note that, typically, $\mathcal{S}$ in the security proof may not be able to provide an immediate answer to the functionality after receiving a message ec.setup.sim. The reason is that $\mathcal{S}$ typically needs to interact with the copy of $\mathcal{A}$ it runs in order to produce the message ec.setup.rep, but $\mathcal{A}$ may not provide the desired answer or may provide a delayed answer. In such cases, when the functionality sends more than one message ec.setup.sim to $\mathcal{S}$, $\mathcal{S}$ may provide delayed replies, and the order of those replies may not follow the order of the messages received.

Aborts. When an ideal functionality $\mathcal{F}$ aborts after being activated with a message sent by a party, we mean that $\mathcal{F}$ halts the execution of its program and sends a special abortion message to the party that invoked the functionality. When an ideal functionality $\mathcal{F}$ aborts after being activated with a message sent by $\mathcal{S}$, we mean that $\mathcal{F}$ halts the execution of its program and sends a special abortion message to the party that receives the outgoing message from $\mathcal{F}$ after $\mathcal{F}$ is activated by $\mathcal{S}$.

C DEFINITIONS OF IDEAL FUNCTIONALITIES

C.1 Secure Message Transmission

Our e-cash schemes use the functionality $\mathcal{F}_{\text{SMT}}$ for secure message transmission described in [21]. $\mathcal{F}_{\text{SMT}}$ interacts with a sender $\mathcal{T}$ and a receiver $\mathcal{R}$, and consists of one interface smt.send. $\mathcal{T}$ uses the smt.send interface to send a message m to $\mathcal{F}_{\text{SMT}}$. $\mathcal{F}_{\text{SMT}}$ leaks $l(m)$, where $l : \mathcal{M} \to \mathbb{N}$ is a function that leaks the message length, to the simulator $\mathcal{S}$. After receiving a response from $\mathcal{S}$, $\mathcal{F}_{\text{SMT}}$ sends m to $\mathcal{R}$. $\mathcal{S}$ cannot modify m. The session identifier *sid* contains the identities of $\mathcal{T}$ and $\mathcal{R}$.

Ideal Functionality $\mathcal{F}_{\text{SMT}}$. $\mathcal{F}_{\text{SMT}}$ is parameterized by a message space $\mathcal{M}$ and by a leakage function $l : \mathcal{M} \to \mathbb{N}$, which leaks the message length.

(1) On input (smt.send.ini, *sid*, m) from a party $\mathcal{T}$:
- Abort if $sid \neq (\mathcal{T}, \mathcal{R}, sid')$ or if $m \notin \mathcal{M}$.
- Create a fresh *qid* and store $(qid, \mathcal{R}, m)$.
- Send (smt.send.sim, *sid*, *qid*, $l(m)$) to $\mathcal{S}$.

S. On input (smt.send.rep, *sid*, *qid*) from $\mathcal{S}$:
- Abort if $(qid, \mathcal{R}, m)$ is not stored.

- Delete the record $(qid, \mathcal{R}, m)$.
- Send (smt.send.end, sid, m) to $\mathcal{R}$.

C.2 Key Genearation

In our e-cash constructions, the setup phase generates the parameters $params$ through algorithm $\mathsf{Setup}(1^k, L)$. Moreover, algorithm $\mathsf{KeyGenV}(params, t, n)$ generates a key pair for the Pointcheval-Sanders signature scheme in such a way that the shares of the secret key are given to n authorities, so that $t \leq n$ authorities are needed to produce a signature.

To simplify our security analysis, in a manner similar to [50], we define an ideal functionality $\mathcal{F}_{\text{KG}}$ that runs both algorithms. In our construction in §4, $\mathcal{F}_{\text{KG}}$ gives $params$ and the public keys pk and $(pk_{\mathcal{V}_i})_{i\in[1,n]}$ to any party running the protocol, while each authority $\mathcal{V}_i$ also receives his secret key $sk_{\mathcal{V}_i}$.

$\mathcal{F}_{\text{KG}}$ could be replaced by an ideal functionality for distributed key generation (DKG) [34, 38]. DKG would avoid the need of a trusted party to generate the keys. With that replacement, our constructions would realize a modified version of our functionality in §3.3, where authorities cannot finalize the execution of the setup interface without involvement of other authorities. The security analysis of the remaining phases of our e-cash schemes is not affected by the fact that the authorities keys are generated by a trusted party or through a DKG protocol.

$\mathcal{F}_{\text{KG}}$ interacts with n authorities $(\mathcal{A}_1, \ldots, \mathcal{A}_n)$. $\mathcal{F}_{\text{KG}}$ consists of two interfaces kg.getkey and kg.retrieve. The interface kg.getkey is used by $\mathcal{A}_i$ to obtain its public key $pk_{\mathcal{V}_i}$ and secret key $sk_{\mathcal{V}_i}$, as well as the public key pk and the parameters $params$. The interface kg.retrieve is used by any party $\mathcal{P}$ to obtain pk, $params$ and the public keys $\langle pk_{\mathcal{V}_i}\rangle_{i=1}^{n}$ of each of the authorities.

Ideal Functionality $\mathcal{F}_{\text{KG}}$. $\mathcal{F}_{\text{KG}}$ is parameterized by probabilistic algorithms Setup and KeyGenV, a security parameter 1^k, a threshold t and a number L of coins in a wallet.

(1) On input (kg.getkey.ini, sid) from an authority $\mathcal{A}_i$:
- Abort if $sid \neq (\mathcal{A}_1, \ldots, \mathcal{A}_n, sid')$, or if $n < t$.
- If $(sid, params, pk, \langle sk_{\mathcal{V}_i}, pk_{\mathcal{V}_i}\rangle_{i\in[1,n]})$ is not stored, do the following:
  - Run $params \leftarrow \mathsf{Setup}(1^k, L)$.
  - Run $(pk, \langle sk_{\mathcal{V}_i}, pk_{\mathcal{V}_i}\rangle_{i\in[1,n]}) \leftarrow \mathsf{KeyGenV}(params, t, n)$.
  - Store $(sid, params, pk, \langle sk_{\mathcal{V}_i}, pk_{\mathcal{V}_i}\rangle_{i\in[1,n]})$.
- Create a fresh qid and store $(qid, \mathcal{A}_i)$.
- Send (kg.getkey.sim, sid, qid, $params$, pk, $\langle pk_{\mathcal{V}_i}\rangle_{i\in[1,n]}$) to $\mathcal{S}$.

S. On input (kg.getkey.rep, sid, qid) from the simulator $\mathcal{S}$:
- Abort if $(qid, \mathcal{A}_i)$ such that $qid \neq qid'$ is not stored.
- Delete $(qid, \mathcal{A}_i)$.
- Send (kg.getkey.end, sid, $params$, pk, $sk_{\mathcal{V}_i}$, $pk_{\mathcal{V}_i}$) to $\mathcal{A}_i$.

(2) On input (kg.retrieve.ini, sid) from any party $\mathcal{P}$:
- Abort if $sid \neq (\mathcal{A}_1, \ldots, \mathcal{A}_n, sid')$.
- If the tuple $(sid, params, pk, \langle sk_{\mathcal{V}_i}, pk_{\mathcal{V}_i}\rangle_{i\in[1,n]})$ is stored, set $v \leftarrow (params, pk, \langle pk_{\mathcal{V}_i}\rangle_{i\in[1,n]})$, else set $v \leftarrow \perp$.
- Create a fresh qid and store $(qid, \mathcal{P}, v)$.
- Send (kg.retrieve.sim, sid, qid, v) to $\mathcal{S}$.

S. On input (kg.retrieve.rep, sid, qid) from $\mathcal{S}$:
- Abort if $(qid', \mathcal{P}, v)$ such that $qid' \neq qid$ is not stored.
- Delete the record $(qid, \mathcal{P}, v)$.
- Send (kg.retrieve.end, sid, v) to $\mathcal{P}$.

C.3 Registration

Our protocol uses the functionality $\mathcal{F}_{\text{REG}}$ for key registration by Canetti [21]. $\mathcal{F}_{\text{REG}}$ interacts with any party $\mathcal{T}$ that registers a message v and with any party $\mathcal{P}$ that retrieves the registered message. $\mathcal{F}_{\text{REG}}$ consists of two interfaces reg.register and reg.retrieve. The interface reg.register is used by $\mathcal{T}$ to register a message v with $\mathcal{F}_{\text{REG}}$. A party $\mathcal{P}$ uses reg.retrieve to retrieve v from $\mathcal{F}_{\text{REG}}$.

Ideal Functionality $\mathcal{F}_{\text{REG}}$. $\mathcal{F}_{\text{REG}}$ is parameterized by a message space $\mathcal{M}$.

(1) On input (reg.register.ini, sid, v) from a party $\mathcal{T}$:
- Abort if $sid \neq (\mathcal{T}, sid')$, or if $v \notin \mathcal{M}$ or if there is a tuple $(sid, v', 0)$ stored.
- Store $(sid, v, 0)$.
- Send (reg.register.sim, sid, v) to $\mathcal{S}$.

S. On input (reg.register.rep, sid) from the simulator $\mathcal{S}$:
- Abort if $(sid, v, 0)$ is not stored or if $(sid, v, 1)$ is already stored.
- Store $(sid, v, 1)$ and parse sid as $(\mathcal{T}, sid')$.
- Send (reg.register.end, sid) to $\mathcal{T}$.

(2) On input (reg.retrieve.ini, sid) from any party $\mathcal{P}$:
- If $(sid, v, 1)$ is stored, set $v' \leftarrow v$; else set $v' \leftarrow \perp$.
- Create a fresh qid and store $(qid, \mathcal{P}, v')$.
- Send (reg.retrieve.sim, sid, qid, v') to $\mathcal{S}$.

S. On input (reg.retrieve.rep, sid, qid) from $\mathcal{S}$:
- Abort if $(qid, \mathcal{P}, v')$ is not stored.
- Delete the record $(qid, \mathcal{P}, v')$.
- Send (reg.retrieve.end, sid, v') to $\mathcal{P}$.

C.4 Pseudonymous Channel

Our e-cash schemes use the functionality $\mathcal{F}_{\text{NYM}}$ for a secure idealized pseudonymous channel. We use $\mathcal{F}_{\text{NYM}}$ to describe our e-cash schemes for simplicity, in order to hide the details of real-world pseudonymous channels. $\mathcal{F}_{\text{NYM}}$ is similar to the functionality for anonymous secure message transmission in [16]. $\mathcal{F}_{\text{NYM}}$ interacts with senders $\mathcal{T}$ and receivers $\mathcal{R}$. $\mathcal{F}_{\text{NYM}}$ is parameterized by a message space $\mathcal{M}$, a security parameter k, a universe of pseudonyms $\mathbb{U}_p$, and a leakage function l, which leaks the message length. $\mathcal{F}_{\text{NYM}}$ consists of one interfaces nym.send. $\mathcal{T}$ uses the nym.send interface to send a message $m \in \mathcal{M}$, a pseudonym $P \in \mathbb{U}_p$ and a receiver identifier $\mathcal{R}$ to $\mathcal{F}_{\text{NYM}}$. $\mathcal{F}_{\text{NYM}}$ sends $l(m)$ to the simulator $\mathcal{S}$. After receiving a response from $\mathcal{S}$, $\mathcal{F}_{\text{NYM}}$ sends m and P to $\mathcal{R}$.

$\mathcal{R}$ does not learn the identifier $\mathcal{T}$. Instead $\mathcal{R}$ learns a pseudonym P chosen by $\mathcal{T}$. $\mathcal{T}$ can choose different pseudonyms to make the messages sent unlinkable towards $\mathcal{R}$.

Ideal Functionality $\mathcal{F}_{\text{NYM}}$. $\mathcal{F}_{\text{NYM}}$ is parameterized by a message space $\mathcal{M}$, a security parameter k, a universe of pseudonyms $\mathbb{U}_p$, and a leakage function l, which leaks the message length.

(1) On input (nym.send.ini, sid, m, P, $\mathcal{R}$) from $\mathcal{T}$:
- Abort if $m \notin \mathcal{M}$, or if $P \notin \mathbb{U}_p$.
- Create a fresh qid and store $(qid, P, \mathcal{T}, m, \mathcal{R})$.
- Send (nym.send.sim, sid, qid, $l(m)$) to $\mathcal{S}$.

S. On input (nym.send.rep, sid, qid) from $\mathcal{S}$:
- Abort if $(qid', P, \mathcal{T}, m, \mathcal{R})$ such that $qid = qid'$ is not stored.
- Delete the record $(qid, P, \mathcal{T}, m, \mathcal{R})$.
- Send (nym.send.end, sid, m, P) to $\mathcal{R}$.

C.5 Authenticated Bulletin Board

Our e-cash schemes use the functionality $\mathcal{F}_{\text{BB}}$ for an authenticated bulletin board BB [56]. A BB is used to store the payments deposited by providers, and to verify that there are not double spendings or double deposits. $\mathcal{F}_{\text{BB}}$ interacts with writers $\mathcal{W}_j$ and readers $\mathcal{R}_k$. $\mathcal{W}_j$ uses the bb.write interface to send a message m to $\mathcal{F}_{\text{BB}}$. $\mathcal{F}_{\text{BB}}$ increments a counter ct of the number of messages stored in BB and appends $[ct, \mathcal{W}_j, m]$ to BB. $\mathcal{R}_k$ uses the bb.read interface on input an index i. If $i \in [1, ct]$, $\mathcal{F}_{\text{BB}}$ takes the tuple $[i, \mathcal{W}_j, m]$ in BB and sends $(\mathcal{W}_j, m)$ to $\mathcal{R}_k$.

Ideal Functionality $\mathcal{F}_{\text{BB}}$. $\mathcal{F}_{\text{BB}}$ is parameterized by a universe of messages $\mathbb{U}_m$. $\mathcal{F}_{\text{BB}}$ interacts with writers $\mathcal{W}_j$ and readers $\mathcal{R}_k$.

(1) On input (bb.write.ini, *sid*, *m*) from $\mathcal{W}_j$:
- Abort if $m \notin \mathbb{U}_m$.
- Create a fresh *qid* and store $(qid, \mathcal{W}_j, m)$.
- Send (bb.write.sim, *sid*, *qid*) to $\mathcal{S}$.

S. On input (bb.write.rep, *sid*, *qid*) from $\mathcal{S}$:
- Abort if $(qid', \mathcal{W}_j, m)$ such that $qid' = qid$ is not stored.
- If (*sid*, BB, *ct*) is not stored, set BB $\leftarrow \perp$ and $ct \leftarrow 0$.
- Increment *ct*, append $[ct, \mathcal{W}_j, m]$ to BB and update *ct* and BB in (*sid*, BB, *ct*).
- Delete $(qid, \mathcal{W}_j, m)$.
- Send (bb.write.end, *sid*) to $\mathcal{W}_j$.

(2) On input (bb.read.ini, *sid*, *i*) from $\mathcal{R}_k$:
- Create a fresh *qid* and store $(qid, \mathcal{R}_k, i)$.
- Send (bb.read.sim, *sid*, *qid*) to $\mathcal{S}$.

S. On input (bb.read.rep, *sid*, *qid*) from $\mathcal{S}$:
- Abort if $(qid', \mathcal{R}_k, i)$ such that $qid' = qid$ is not stored.
- If (*sid*, BB, *ct*) is stored and $i \in [1, ct]$, take $[i, \mathcal{W}_j, m]$ from BB and set $m' \leftarrow (\mathcal{W}_j, m)$, else set $m' \leftarrow \perp$.
- Send (bb.read.end, *sid*, m') to $\mathcal{R}_k$.

C.6 Ideal Functionality $\mathcal{F}^R_{\text{SZK}}$

In our schemes, for efficiency reasons, we use the Fiat-Shamir heuristic to compute signatures of knowledge. This implies that our schemes are not UC secure. To allow for UC security, our schemes can make use of the ideal functionality $\mathcal{F}^R_{\text{SZK}}$ for signatures of knowledge, which we define below. To be realized, this functionality requires a signature of knowledge scheme that provides straight-line extraction.

Functionality $\mathcal{F}^R_{\text{SZK}}$ interacts with any parties $\mathcal{P}$ that obtain the parameters *par* and compute and verify signatures of knowledge. The interaction between the functionality $\mathcal{F}^R_{\text{SZK}}$ and those parties takes place through the following interfaces:

(1) Any party $\mathcal{P}$ employs the szk.setup interface to obtain the parameters *par*.
(2) Any party $\mathcal{P}$ employs the szk.prove interface to obtain a signature of knowledge π on input a witness *wit*, an instance *ins* and a message *mes* such that $(wit, ins) \in R$.
(3) Any party $\mathcal{P}$ employs the szk.verify interface to verify a signature of knowledge π for an instance *ins* and a message *mes*.

$\mathcal{F}^R_{\text{SZK}}$ uses a table Tbl. Tbl consists of entries of the form $[ins, mes, \pi, u]$, where *ins* is an instance, *mes* is a message, π is a signature of knowledge, and u is a bit that indicates whether π is a valid signature of knowledge for the instance *ins* or not. $\mathcal{F}^R_{\text{SZK}}$ also uses a set $\mathbb{S}$ that contains all the parties that have obtained the parameters *par*.

$\mathcal{F}^R_{\text{SZK}}$ is similar to the ideal functionality for non-interactive zero-knowledge in [35]. $\mathcal{F}^R_{\text{SZK}}$ asks the simulator $\mathcal{S}$ to provide simulation and extraction algorithms at setup. In [35], the functionality asks the simulator to provide simulated signatures of knowledge and to extract witnesses when the szk.prove and szk.verify interfaces are invoked. We choose the first alternative because it hides from the simulator the instances and messages and also when the parties compute and verify signatures of knowledge. In comparison to the functionality in [35], $\mathcal{F}^R_{\text{SZK}}$ adds the message to be signed.

The functionality $\mathcal{F}^R_{\text{SZK}}$ does not allow the computation and verification of signatures of knowledge using parameters *par* that were not generated by the functionality. As can be seen, the interfaces szk.prove and szk.verify do not receive the parameters of the scheme as input and, in order to compute and verify signatures of knowledge, the functionality employs the parameters that it stores. Therefore, a construction that realizes this functionality must ensure that the honest parties employ the same parameters. To achieve this, some form of trusted setup is required. We depict $\mathcal{F}^R_{\text{SZK}}$ below.

Ideal Functionality $\mathcal{F}^R_{\text{SZK}}$. SZK.SimProve and SZK.Extract are ppt algorithms. $\mathcal{F}^R_{\text{SZK}}$ is parameterized with a description of a relation R. $\mathcal{F}^R_{\text{SZK}}$ interacts with any parties $\mathcal{P}$ that obtain the parameters *par* and compute and verify signatures of knowledge.

(1) On input (szk.setup.ini, *sid*) from any party $\mathcal{P}$:
- Generate a random *qid* and store $(qid, \mathcal{P})$.
- If the tuple (*sid*, *par*, *td*, SZK.SimProve, SZK.Extract) is not stored, send the message (szk.setup.req, *sid*, *qid*, $\mathcal{P}$) to $\mathcal{S}$, else send (szk.setup.sim, *sid*, *qid*, $\mathcal{P}$) to $\mathcal{S}$.

S. On input from $\mathcal{S}$ the message (szk.setup.alg, *sid*, *qid*, *par*, *td*, SZK.SimProve, SZK.Extract):
- Abort if $(qid, \mathcal{P})$ is not stored.
- If (*sid*, *par*, *td*, SZK.SimProve, SZK.Extract) is not stored, do the following:
  - Store (*sid*, *par*, *td*, SZK.SimProve, SZK.Extract).
  - Create an empty set $\mathbb{S}$.
- Set $\mathbb{S} \leftarrow \mathbb{S} \cup \mathcal{P}$.
- Delete $(qid, \mathcal{P})$.
- Send (szk.setup.end, *sid*, *par*) to $\mathcal{P}$.

S. On input (szk.setup.rep, *sid*, *qid*) from $\mathcal{S}$:
- Abort if $(qid, \mathcal{P})$ is not stored or if (*sid*, *par*, *td*, SZK.SimProve, SZK.Extract) is not stored.
- Set $\mathbb{S} \leftarrow \mathbb{S} \cup \mathcal{P}$.
- Delete $(qid, \mathcal{P})$.
- Send (szk.setup.end, *sid*, *par*) to $\mathcal{P}$.

(2) On input (szk.prove.ini, *sid*, *wit*, *ins*, *mes*) from a party $\mathcal{P}$:
- Abort if $\mathcal{P} \notin \mathbb{S}$ or if $(wit, ins) \notin R$.
- Run $\pi \leftarrow$ SZK.SimProve(*sid*, *par*, *td*, *ins*, *mes*).
- Append $[ins, mes, \pi, 1]$ to Table Tbl.
- Send (szk.prove.end, *sid*, π) to $\mathcal{P}$.

(3) On input (szk.verify.ini, *sid*, *ins*, *mes*, π) from a party $\mathcal{P}$:
- Abort if $\mathcal{P} \notin \mathbb{S}$.
- If there is an entry $[ins, mes, \pi, u]$ in Tbl, set $v \leftarrow u$.
- Else, do the following:

- Extract $wit \leftarrow \mathsf{SZK.Extract}(sid, par, td, \pi, ins, mes)$.
- If $(wit, ins) \notin R$, set $v \leftarrow 0$, else set $v \leftarrow 1$.
- Append $[ins, mes, \pi, v]$ to Tbl.

• Send $(\text{szk.verify.end}, sid, v)$ to $\mathcal{P}$.

We discuss the three interfaces of $\mathcal{F}_{\text{SZK}}^{R}$.

(1) The szk.setup.ini message is sent by any party $\mathcal{P}$. If the algorithms are not stored, $\mathcal{F}_{\text{SZK}}^{R}$ asks the simulator $\mathcal{S}$ to send the parameters *par*, the trapdoor *td* and the algorithms SZK.SimProve and SZK.Extract. Else, $\mathcal{F}_{\text{SZK}}^{R}$ asks the simulator to allow the setup phase to be completed for party $\mathcal{P}$. When the simulator provides the parameters, trapdoor and algorithms, the functionality stores them if they were not stored before. Both when the simulator provides the algorithms or when it simply prompts the completion of the setup phase for party $\mathcal{P}$, the functionality records that $\mathcal{P}$ obtains the parameters and sends the parameters to $\mathcal{P}$.

(2) The szk.prove.ini message is sent by any honest party $\mathcal{P}$ on input a witness *wit*, an instance *ins* and a message *mes*. $\mathcal{F}_{\text{SZK}}^{R}$ aborts if $\mathcal{P}$ did not obtain the parameters. This is required because $\mathcal{F}_{\text{SZK}}^{R}$ enforces that the computation of a signature of knowledge is a local process (note that $\mathcal{F}_{\text{SZK}}^{R}$ does not communicate with the simulator when computing a signature of knowledge), so parties in the real world must have obtained the parameters before computing a signature of knowledge. $\mathcal{F}_{\text{SZK}}^{R}$ runs the algorithm SZK.SimProve on input *par*, *td* and *ins* to get a simulated signature of knowledge π. SZK.SimProve does not receive the witness *wit* as input, and therefore a signature of knowledge scheme that realizes this functionality must fulfill the zero-knowledge property. $\mathcal{F}_{\text{SZK}}^{R}$ stores $[ins, mes, \pi, 1]$ in Tbl and sends π to $\mathcal{P}$.

(3) The szk.verify.ini message is sent by any honest party $\mathcal{P}$ on input an instance *ins*, a message *mes*, and a signature of knowledge π. $\mathcal{F}_{\text{SZK}}^{R}$ aborts if $\mathcal{P}$ did not obtain the parameters because the verification of a signature of knowledge is enforced to be a local process. If there is an entry $[ins, mes, \pi, u]$ already stored in Tbl, then the functionality returns the bit u. Therefore, any signature of knowledge scheme that realizes this functionality must be consistent. Else, the functionality runs the algorithm SZK.Extract to get a witness *wit* such that $(wit, ins) \in R$. Therefore, any scheme that realizes $\mathcal{F}_{\text{SZK}}^{R}$ must be extractable. If extraction fails, the functionality sets the verification result to 0, i.e., extraction must succeed unless the signature of knowledge is incorrect. The functionality records the verification result in Tbl and returns that result.

D SECURITY PROOF FOR OUR COMPACT E-CASH SCHEME

To prove that construction Π_{EC}, instantiated with the algorithms of the compact e-cash scheme in §5.1, securely realizes the ideal functionality $\mathcal{F}_{\text{EC}}$, we have to show that for any environment $\mathcal{Z}$ and any adversary $\mathcal{A}$ there exists a simulator $\mathcal{S}$ such that $\mathcal{Z}$ cannot distinguish between whether it is interacting with $\mathcal{A}$ and the protocol in the real world or with $\mathcal{S}$ and $\mathcal{F}_{\text{EC}}$. The simulator thereby plays the role of all honest parties in the real world and interacts with $\mathcal{F}_{\text{EC}}$ for all corrupt parties in the ideal world.

$\mathcal{S}$ runs a copy of any adversary $\mathcal{A}$, which is used to provide to $\mathcal{Z}$ a view that is indistinguishable from the view given by $\mathcal{A}$ in the real world. To achieve that, $\mathcal{S}$ must simulate the real-world protocol towards the copy of $\mathcal{A}$, in such a way that $\mathcal{A}$ cannot distinguish an interaction with $\mathcal{S}$ from an interaction with the real-world protocol. $\mathcal{S}$ uses the information provided by $\mathcal{F}_{\text{EC}}$ to provide a simulation of the real-world protocol.

Our simulator $\mathcal{S}$ runs copies of the functionalities $\mathcal{F}_{\text{SMT}}$, $\mathcal{F}_{\text{NYM}}$, $\mathcal{F}_{\text{KG}}$, $\mathcal{F}_{\text{REG}}$ and $\mathcal{F}_{\text{BB}}$. When any of the copies of these functionalities aborts, $\mathcal{S}$ implicitly forwards the abortion message to the adversary if the functionality sends the abortion message to a corrupt party.

$\mathcal{S}$ also runs copies of the extractors $\mathcal{E}_s$ and $\mathcal{E}_v$ and simulators $\mathcal{S}_s$ and $\mathcal{S}_v$ for the non-interactive zero-knowledge arguments of knowledge π_s and π_v, which are computed through the Fiat-Shamir transform. We remark that they involve calls to the random oracle and rewinding of the adversary. For simplicity, we omit those details.

Challenges in the proof. Our compact e-cash scheme with threshold issuance is based on an existing compact e-cash scheme [14]. We instantiate that scheme with the Pointcheval-Sanders signature scheme, and then we use an existing threshold issuance protocol for Pointcheval-Sanders signatures [50] to provide threshold issuance. Nevertheless, the security proof does not follow straightforwardly from the proofs in [14, 50].

First, in the blind issuance protocol of the compact e-cash scheme in [14] one of the coin secrets is chosen jointly between the user and the bank, i.e. both the bank and the user add randomness to compute the secret. In a threshold issuance setting, it would be necessary that all the authorities add the same randomness without communicating between them. While adding a mechanism to do that is possible, in order to improve efficiency we decided to have the secret be chosen by the user only. This implies that, when a corrupt user picks up the secrets of two wallets, the user could pick up the secrets in such a way that two coins have the same serial number. Consequently, when those coins are spent, there is a situation in which double spending is detected (because serial numbers are the same), although no double spending has happened. In that case, we prove that the identification algorithm cannot output the public key of an honest user, and we are able to do that under the discrete logarithm assumption.

Second, to improve efficiency, we use only one coin secret instead of the two coins secrets used in [14]. When using two coin secrets, it is straightforward to show that serial numbers and double spending tags do not reveal any information about the secrets by using the pseudorandomness property of the pseudorandom function. However, when using the same secret for both, but a different generator, we need to add an additional reduction to the XDH assumption.

Simulator. We describe the simulator $\mathcal{S}$ for the case in which a subset of users $\mathcal{U}_j$, a subset of providers $\mathcal{P}_k$ and up to $t-1$ authorities are corrupt. $\mathcal{S}$ simulates the honest parties in the protocol Π_{EC} and runs copies of the ideal functionalities involved.

Honest authority $\mathcal{V}_i$ starts setup. When the functionality $\mathcal{F}_{\text{EC}}$ sends the message $(\text{ec.setup.sim}, sid, \mathcal{V}_i)$, the simulator $\mathcal{S}$ runs a copy of the functionality $\mathcal{F}_{\text{KG}}$ on input $(\text{kg.getkey.ini}, sid)$. When

the functionality $\mathcal{F}_{\text{KG}}$ sends the message (kg.getkey.sim, *sid*, *qid*, *params*, *pk*, $\langle pk_{\mathcal{V}_i}\rangle_{i\in[1,n]}$), $\mathcal{S}$ forwards that message to $\mathcal{A}$.

Honest authority $\mathcal{V}_i$ ends setup. When the adversary $\mathcal{A}$ sends the message (kg.getkey.rep, *sid*, *qid*), $\mathcal{S}$ runs a copy of $\mathcal{F}_{\text{KG}}$ on input that message. When $\mathcal{F}_{\text{KG}}$ sends (kg.getkey.end, *sid*, *params*, *pk*, $sk_{\mathcal{V}_i}$, $pk_{\mathcal{V}_i}$), $\mathcal{S}$ sends (ec.setup.rep, *sid*, $\mathcal{V}_i$) to $\mathcal{F}_{\text{EC}}$.

Corrupt authority $\tilde{\mathcal{V}}_i$ starts setup. When a corrupt authority $\tilde{\mathcal{V}}_i$ sends the message (kg.getkey.ini, *sid*), $\mathcal{S}$ runs a copy of $\mathcal{F}_{\text{KG}}$ on input that message. When $\mathcal{F}_{\text{KG}}$ sends (kg.getkey.sim, *sid*, *qid*, *params*, *pk*, $\langle pk_{\mathcal{V}_i}\rangle_{i\in[1,n]}$), $\mathcal{S}$ forwards that message to $\tilde{\mathcal{V}}_i$.

Corrupt authority $\tilde{\mathcal{V}}_i$ ends setup. When $\tilde{\mathcal{V}}_i$ sends the message (kg.getkey.rep, *sid*, *qid*), $\mathcal{S}$ runs a copy of $\mathcal{F}_{\text{KG}}$ on input that message. When $\mathcal{F}_{\text{KG}}$ sends (kg.getkey.end, *sid*, *params*, *pk*, $sk_{\tilde{\mathcal{V}}_i}$, $pk_{\tilde{\mathcal{V}}_i}$), $\mathcal{S}$ sends (ec.setup.ini, *sid*) to the functionality $\mathcal{F}_{\text{EC}}$. When $\mathcal{F}_{\text{EC}}$ sends (ec.setup.sim, *sid*, $\tilde{\mathcal{V}}_i$), the simulator $\mathcal{S}$ sends the message (ec.setup.rep, *sid*, $\tilde{\mathcal{V}}_i$) to $\mathcal{F}_{\text{EC}}$. When $\mathcal{F}_{\text{EC}}$ sends (ec.setup.end, *sid*), $\mathcal{S}$ sends the message (kg.getkey.end, *sid*, *params*, *pk*, $sk_{\tilde{\mathcal{V}}_i}$, $pk_{\tilde{\mathcal{V}}_i}$) to $\tilde{\mathcal{V}}_i$.

Honest user (or provider) requests keys. When $\mathcal{F}_{\text{EC}}$ sends the message (ec.register.sim, *sid*, $\mathcal{U}_j$), $\mathcal{S}$ runs a copy of $\mathcal{F}_{\text{KG}}$ on input (kg.retrieve.ini, *sid*). When the functionality $\mathcal{F}_{\text{KG}}$ sends the message (kg.retrieve.sim, *sid*, *qid*, *v*), $\mathcal{S}$ forwards that message to $\mathcal{A}$.

Honest user (or provider) receives and registers keys. When $\mathcal{A}$ sends (kg.retrieve.rep, *sid*, *qid*), $\mathcal{S}$ runs $\mathcal{F}_{\text{KG}}$ on input that message. When the functionality $\mathcal{F}_{\text{KG}}$ sends (kg.retrieve.end, *sid*, *v*), the simulator $\mathcal{S}$ runs $(sk_{\mathcal{U}_j}, pk_{\mathcal{U}_j}) \leftarrow \mathsf{KeyGenU}(params)$, sets $sid_{\text{REG}} \leftarrow (\mathcal{U}_j, sid')$ and runs a copy of $\mathcal{F}_{\text{REG}}$ on input the message (reg.register.ini, sid_{REG}, $pk_{\mathcal{U}_j}$). When $\mathcal{F}_{\text{REG}}$ sends the message (reg.register.sim, sid_{REG}, $pk_{\mathcal{U}_j}$), $\mathcal{S}$ forwards that message to $\mathcal{A}$.

Honest user (or provider) finalizes the registration of keys. When $\mathcal{A}$ sends the message (reg.register.rep, sid_{REG}), $\mathcal{S}$ runs $\mathcal{F}_{\text{REG}}$ on input that message. When $\mathcal{F}_{\text{REG}}$ sends (reg.register.end, sid_{REG}), $\mathcal{S}$ sends (ec.register.rep, *sid*, $\mathcal{U}_j$) to $\mathcal{F}_{\text{EC}}$.

Corrupt user (or provider) requests keys. When $\mathcal{A}$ sends the message (kg.retrieve.ini, *sid*), $\mathcal{S}$ runs $\mathcal{F}_{\text{KG}}$ on input that message. When $\mathcal{F}_{\text{KG}}$ sends (kg.retrieve.sim, *sid*, *qid*, *v*), $\mathcal{S}$ forwards that message to $\mathcal{A}$.

Corrupt user (or provider) receives keys. When $\mathcal{A}$ sends the message (kg.retrieve.rep, *sid*, *qid*), $\mathcal{S}$ runs $\mathcal{F}_{\text{KG}}$ on input that message. When $\mathcal{F}_{\text{KG}}$ sends (kg.retrieve.end, *sid*, *v*), $\mathcal{S}$ forwards that message to $\mathcal{A}$.

Corrupt user (or provider) registers keys. When $\mathcal{A}$ sends the message (reg.register.ini, sid_{REG}, $pk_{\mathcal{U}_j}$), $\mathcal{S}$ runs $\mathcal{F}_{\text{REG}}$ on input that message. When the functionality $\mathcal{F}_{\text{REG}}$ sends (reg.register.sim, sid_{REG}, $pk_{\mathcal{U}_j}$), $\mathcal{S}$ forwards that message to $\mathcal{A}$.

Corrupt user (or provider) finishes the registration of keys. When $\mathcal{A}$ sends (reg.register.rep, sid_{REG}), $\mathcal{S}$ runs $\mathcal{F}_{\text{REG}}$ on input that message. When $\mathcal{F}_{\text{REG}}$ sends (reg.register.end, sid_{REG}), the simulator $\mathcal{S}$, acting as the corrupt user $\mathcal{U}_j$, sends (ec.register.ini, *sid*) to $\mathcal{F}_{\text{EC}}$. When the functionality $\mathcal{F}_{\text{EC}}$ sends (ec.register.sim, *sid*, $\mathcal{U}_j$), the simulator $\mathcal{S}$ sends (ec.register.rep, *sid*, $\mathcal{U}_j$) to $\mathcal{F}_{\text{EC}}$. When $\mathcal{F}_{\text{EC}}$ sends the message (ec.register.end, *sid*), $\mathcal{S}$ sends (reg.register.end, sid_{REG}) to $\mathcal{A}$.

Honest user sends a request to honest authority. When $\mathcal{F}_{\text{EC}}$ sends (ec.request.sim, *sid*, *qid*, $\mathcal{U}_j$, $\mathcal{V}_i$), the simulator $\mathcal{S}$ parses the session identifier *sid* as $(\mathcal{V}_1, \ldots, \mathcal{V}_n, sid')$, sets $sid_{\text{SMT}} \leftarrow (\mathcal{U}_j, \mathcal{V}_i, sid')$ and sends (smt.send.sim, sid_{SMT}, *qid*, $l(m)$) to $\mathcal{A}$, where $l(m)$ is equal to the length of the message that the honest user sends to an authority in the ec.request interface.

Adversary forwards request. When the adversary $\mathcal{A}$ sends the message (smt.send.rep, sid_{SMT}, *qid*) to reply to (ec.request.sim, *sid*, *qid*, $\mathcal{U}_j$, $\mathcal{V}_i$), $\mathcal{S}$ checks if a tuple (*sid*, $\mathcal{U}_j$, $\mathcal{V}_i$) is stored. If not, $\mathcal{S}$ runs the procedure in Figure 3. After that, $\mathcal{S}$ sends the message (ec.request.rep, *sid*, *qid*) to $\mathcal{F}_{\text{EC}}$.

Honest user sends a request to corrupt authority. When $\mathcal{F}_{\text{EC}}$ sends (ec.request.sim, *sid*, *qid*, $\mathcal{U}_j$, $\mathcal{V}_i$, *wid*), $\mathcal{S}$ checks if a tuple (*sid*, $\mathcal{U}'_j$, *wid*′, *wn*) such that $\mathcal{U}'_j = \mathcal{U}_j$ and *wid*′ = *wid* is stored. If not, $\mathcal{S}$ does the following:

- Pick up the stored tuple (*sid*, $\mathcal{U}'_j$, *wct*) such that $\mathcal{U}'_j = \mathcal{U}_j$. (This tuple is initialized to *wct* = 0.)
- Set *wct* ← *wct* + 1.
- Set *wn* ← *wct*.
- Store (*sid*, $\mathcal{U}_j$, *wid*, *wn*).
- Update *wct* in the tuple (*sid*, $\mathcal{U}_j$, *wct*).

$\mathcal{S}$ parses the session identifier *sid* as $(\mathcal{V}_1, \ldots, \mathcal{V}_n, sid')$, sets sid_{SMT} ← $(\mathcal{U}_j, \mathcal{V}_i, sid')$ and sends (smt.send.sim, sid_{SMT}, *qid*, $l(m)$) to $\mathcal{A}$, where $l(m)$ is equal to the length of the message that the honest user sends to an authority in the ec.request interface.

Corrupt authority receives request from honest user. When $\mathcal{F}_{\text{EC}}$ sends (ec.request.end, *sid*, $\mathcal{U}_j$, *reqid*) to a corrupt authority $\tilde{\mathcal{V}}_i$, $\mathcal{S}$ retrieves the stored tuple (*sid*, $\mathcal{U}_j$, *wid*, *wn*) such that $\mathcal{U}'_j = \mathcal{U}_j$ and *wid* was received before in the message (ec.request.sim, ...) associated with the message (ec.request.end, ...) received now. $\mathcal{S}$ runs a copy of a user on input (ec.request.ini, *sid*, $\tilde{\mathcal{V}}_i$, *reqid*, *wn*). (We remark that *wn* used a input to the copy of the user is not necessarily equal to the value of *wn* received by the honest user from the environment. However, it is enough to ensure that the same *wn* is used for each *wid* received from $\mathcal{F}_{\text{EC}}$, so that the copy of the user produces the same request for each *wid*.) When the copy of the user runs algorithm Request, $\mathcal{S}$ uses the public key $pk_{\mathcal{U}_j}$ that was previously computed for user $\mathcal{U}_j$. $\mathcal{S}$ also uses the simulator $\mathcal{S}_s$ to compute a simulated proof π_s, and sets *com*, com_1 and com_2 to be commitments to random messages. When the copy of the user sends the message (smt.send.ini, sid_{SMT}, ⟨*reqid*, *req*⟩) to $\mathcal{F}_{\text{SMT}}$, $\mathcal{S}$ runs $\mathcal{F}_{\text{SMT}}$ on input that message. When $\mathcal{F}_{\text{SMT}}$ sends the message (smt.send.sim, *sid*, *qid*, $l(m)$), $\mathcal{S}$ runs $\mathcal{F}_{\text{SMT}}$ on input (smt.send.rep, *sid*, *qid*). When $\mathcal{F}_{\text{SMT}}$ sends (smt.send.end, *sid*, ⟨*reqid*, *req*⟩), $\mathcal{S}$ forwards that message to $\tilde{\mathcal{V}}_i$.

Corrupt authority requests user keys. When the adversary $\mathcal{A}$ sends (reg.retrieve.ini, sid_{REG}), $\mathcal{S}$ runs a copy of $\mathcal{F}_{\text{REG}}$ on input that message. When $\mathcal{F}_{\text{REG}}$ sends (reg.register.sim, *sid*, *v*), $\mathcal{S}$ forwards that message to $\mathcal{A}$.

Corrupt authority receives user keys. When the adversary $\mathcal{A}$ sends (reg.register.rep, *sid*), $\mathcal{S}$ runs $\mathcal{F}_{\text{REG}}$ on input that message. When $\mathcal{F}_{\text{REG}}$ sends (reg.retrieve.end, sid_{REG}, *v*), $\mathcal{S}$ forwards that message to $\mathcal{A}$.

Corrupt user requests credential. When the adversary $\mathcal{A}$ sends the message (smt.send.ini, sid_{SMT}, ⟨*reqid*, *req*⟩), $\mathcal{S}$ runs $\mathcal{F}_{\text{SMT}}$ on

Procedure: Retrieve user key

- The simulator $\mathcal{S}$ sets $sid_{\text{REG}} \leftarrow (\mathcal{U}_j, sid')$ and runs the functionality $\mathcal{F}_{\text{REG}}$ on input (reg.retrieve.ini, sid_{REG}). When $\mathcal{F}_{\text{REG}}$ sends (reg.retrieve.sim, sid_{REG}, qid, v'), $\mathcal{S}$ forwards that message to $\mathcal{A}$.
- When $\mathcal{A}$ sends (reg.retrieve.rep, sid_{REG}, qid), $\mathcal{S}$ runs $\mathcal{F}_{\text{REG}}$ on input that message. When $\mathcal{F}_{\text{REG}}$ sends (reg.retrieve.end, sid_{REG}, v'), if $v' = \perp$, $\mathcal{S}$ aborts, else $\mathcal{S}$ stores $(sid, \mathcal{U}_j, \mathcal{V}_i)$.

Figure 3: Procedure for simulating user key retrieval.

input that message. When $\mathcal{F}_{\text{SMT}}$ sends (smt.send.sim, sid_{SMT}, qid, $l(m)$), $\mathcal{S}$ sends that message to $\mathcal{A}$.

Honest authority receives request from corrupt user. When the adversary $\mathcal{A}$ sends (smt.send.rep, sid_{SMT}, qid), $\mathcal{S}$ runs $\mathcal{F}_{\text{SMT}}$ on input that message. When $\mathcal{F}_{\text{SMT}}$ sends the message (smt.send.end, sid_{SMT}, m), $\mathcal{S}$ parses the message m as $\langle reqid, req\rangle$ and sid_{SMT} as $(\mathcal{U}_j, \mathcal{V}_i, sid')$. $\mathcal{S}$ does the following:

- Abort if the authority $\mathcal{V}_i$ did not end the setup.
- Check if a tuple $(sid, \mathcal{U}_j, \mathcal{V}_i)$ is stored. If not, $\mathcal{S}$ runs the procedure in Figure 3.
- Abort if there is a stored tuple $(\mathcal{U}'_j, wn, \mathbb{R}, req, \langle m_1, m_2, o, o_1, o_2\rangle)$ such that $\mathcal{U}'_j = \mathcal{U}_j$ and $\{reqid, \mathcal{V}_i\} \in \mathbb{R}$. Here $\mathcal{S}$ aborts because there is a request pending with the same request identifier $reqid$ from $\mathcal{U}_j$ for $\mathcal{V}_i$, like it is done in the real protocol.
- If there is a stored tuple $(\mathcal{U}'_j, wn, \mathbb{R}, req', \langle m_1, m_2, o, o_1, o_2\rangle)$ such that $\mathcal{U}'_j = \mathcal{U}_j$ and $req' = req$, then update the set $\mathbb{R} \leftarrow \mathbb{R} \cup \{reqid, \mathcal{V}_i\}$ in that tuple and take wn from that tuple. Else do the following:
  - Follow the steps of an honest authority in Π_{EC} to verify req. This involves verifying the request by running the algorithm $b \leftarrow \mathsf{RequestVf}(params, req, pk_{\mathcal{U}_j})$. ($\mathcal{S}$ checked before that the corrupt user registered $pk_{\mathcal{U}_j}$ through the procedure in Figure 3.) Abort if $b = 0$.
  - Parse req as $(h, com, com_1, com_2, \pi_s)$. Run the extractor $\mathcal{E}_s$ to extract the witness $\langle m_1, m_2, o, o_1, o_2\rangle$ from π_s.
  - If there is a tuple stored $(\mathcal{U}_j, wn, \mathbb{R}, \langle h, com', com_1, com_2, \pi_s\rangle, \langle m'_1, m'_2, o, o_1, o_2\rangle)$ such that $com' = com$ but $(m'_1, m'_2) \neq (m_1, m_2)$, $\mathcal{S}$ outputs failure.
  - If there is a stored tuple $(\mathcal{U}_j, wn, \mathbb{R}, req, \langle m_1, m_2, o, o_1, o_2\rangle)$ such that $req = \langle h, com', com_1, com_2, \pi_s\rangle$ and $com' = com$, then take wn from that tuple and store a tuple $(\mathcal{U}_j, wn, \mathbb{R}, req, \langle m_1, m_2, o, o_1, o_2\rangle)$, where $\mathbb{R} \leftarrow \{reqid, \mathcal{V}_i\}$ and $\langle m_1, m_2, o, o_1, o_2\rangle$ is the witness that was output by the extractor $\mathcal{E}_s$. (We remark that wn is the same because, if com is the same in two requests, the partial signatures obtained from two different authorities can be aggregated, even if the values (com_1, com_2, π_s) are different.) Else do the following:
    - Pick up the stored tuple $(sid, \mathcal{U}'_j, wct)$ such that $\mathcal{U}'_j = \mathcal{U}_j$. (This tuple is initialized to $wct = 0$.)
    - Set $wct \leftarrow wct + 1$.
    - Set $wn \leftarrow wct$.
    - Store $(\mathcal{U}_j, wn, \mathbb{R}, req, \langle m_1, m_2, o, o_1, o_2\rangle)$, where $\mathbb{R} \leftarrow \{reqid, \mathcal{V}_i\}$.
    - Update wct in the tuple $(sid, \mathcal{U}_j, wct)$.

  $\mathcal{S}$ sends (ec.request.ini, $sid, \mathcal{V}_i, reqid, wn$) to the functionality $\mathcal{F}_{\text{EC}}$. When $\mathcal{F}_{\text{EC}}$ sends (ec.request.sim, sid, $qid, \mathcal{U}_j, \mathcal{V}_i$), $\mathcal{S}$ sends (ec.request.rep, sid, qid) to $\mathcal{F}_{\text{EC}}$.

Honest authority issues attribute. When the functionality $\mathcal{F}_{\text{EC}}$ sends the message (ec.issue.sim, sid, $qid, \mathcal{V}_i, \mathcal{U}_j$), $\mathcal{S}$ parses sid as $(\mathcal{V}_1, \ldots, \mathcal{V}_n, sid')$, sets $sid_{\text{SMT}} \leftarrow (\mathcal{V}_i, \mathcal{U}_j, sid')$ and sends the message (smt.send.sim, sid_{SMT}, qid, $l(m)$), where $l(m)$ is the length of the message that an honest authority sends in the ec.issue interface.

Adversary forwards issuance. When the adversary $\mathcal{A}$ sends the message (smt.send.rep, sid_{SMT}, qid), $\mathcal{S}$ sends (ec.issue.rep, sid, qid) to $\mathcal{F}_{\text{EC}}$.

Corrupt user receives issuance. When $\mathcal{F}_{\text{EC}}$ sends the message (ec.issue.end, sid, $reqid, \mathcal{V}_i$) to a corrupt user $\mathcal{U}_j$, the simulator $\mathcal{S}$ finds the stored tuple $(\mathcal{U}'_j, wn, \mathbb{R}, req, \langle m_1, m_2, o, o_1, o_2\rangle)$ such that $\mathcal{U}'_j = \mathcal{U}_j$ and there is element $\{reqid', \mathcal{V}'_i\} \in \mathbb{R}$ such that $reqid' = reqid$ and $\mathcal{V}'_i = \mathcal{V}_i$. $\mathcal{S}$ parses req as $(h, com, com_1, com_2, \pi_s)$. $\mathcal{S}$ parses the secret key $sk_{\mathcal{V}_i}$ as $(x_i, y_{i,1}, y_{i,2})$, computes $c = h^{x_i} \prod_{j=1}^{2} com_j^{y_{i,j}}$ and sets the blinded signature share $\hat{\sigma}_i \leftarrow (h, c)$ as in Π_{EC}. If a tuple $(sid, \mathcal{U}'_j, wn', \mathbb{S})$ such that $\mathcal{U}'_j = \mathcal{U}_j$ and $wn' = wn$ is not stored, $\mathcal{S}$ stores a tuple $(sid, \mathcal{U}_j, wn, \{i\})$, else updates $\mathbb{S} \leftarrow \mathbb{S} \cup \{i\}$ in the tuple $(sid, \mathcal{U}_j, wn, \mathbb{S})$. After that, $\mathcal{S}$ removes $\{reqid, \mathcal{V}_i\}$ from the set $\mathbb{R}$ in the tuple $(\mathcal{U}_j, wn, \mathbb{R}, req, \langle m_1, m_2, o, o_1, o_2\rangle)$. Finally, $\mathcal{S}$ sets $sid_{\text{SMT}} \leftarrow (\mathcal{V}_i, \mathcal{U}_j, sid')$, sets $res \leftarrow \hat{\sigma}_i$ and sends (smt.send.end, sid_{SMT}, $\langle reqid, res\rangle$) to $\mathcal{A}$.

Corrupt authority issues attribute. When a corrupt authority $\tilde{\mathcal{V}}_i$ sends the message (smt.send.ini, sid_{SMT}, $\langle reqid, res\rangle$), $\mathcal{S}$ runs $\mathcal{F}_{\text{SMT}}$ on input that message. When the functionality $\mathcal{F}_{\text{SMT}}$ sends (smt.send.sim, sid, qid, $l(m)$), $\mathcal{S}$ sends that message to $\mathcal{A}$.

Honest user receives issuance from corrupt authority. After the adversary $\mathcal{A}$ sends the message (smt.send.rep, sid, qid), the simulator $\mathcal{S}$ runs $\mathcal{F}_{\text{SMT}}$ on input that message. When $\mathcal{F}_{\text{SMT}}$ sends (smt.send.end, sid_{SMT}, $\langle reqid, res\rangle$), the simulator $\mathcal{S}$ parses sid_{SMT} as $(\mathcal{V}_i, \mathcal{U}_j, sid')$ and runs the copy of the user $\mathcal{U}_j$ on input that message. We remark that the copy of the user finds the request identifier $reqid$ of the request associated with this issuance message, or aborts if it is not found. When the copy of the user outputs (ec.issue.end, sid, $reqid, \mathcal{V}_i$), $\mathcal{S}$ sends (ec.issue.ini, $sid, \mathcal{U}_j, reqid$) to $\mathcal{F}_{\text{EC}}$. When $\mathcal{F}_{\text{EC}}$ sends (ec.issue.sim, sid, $qid, \mathcal{V}_i, \mathcal{U}_j$), $\mathcal{S}$ sends (ec.issue.rep, sid, qid) to $\mathcal{F}_{\text{EC}}$.

Honest user begins spending. When $\mathcal{F}_{\text{EC}}$ sends (ec.spend.sim, sid, qid), $\mathcal{S}$ sends (nym.send.sim, sid, qid, $l(m)$) to $\mathcal{A}$, where $l(m)$ is the length of the message $\langle pay, payinfo\rangle$ sent by an honest user in the spend interface. Here we consider that the length does not change depending on the number of spent coins. This can be achieved by setting a maximum number of coins to be spent and sending always messages whose length is the length of the message when the maximum number of coins is spent.

Adversary forwards spending. When $\mathcal{A}$ sends (nym.send.rep, sid, qid), $\mathcal{S}$ sends (ec.spend.rep, sid, qid) to $\mathcal{F}_{\text{EC}}$.

Corrupt provider receives spending. When $\mathcal{F}_{\text{EC}}$ sends the message (ec.spend.end, $payid, V$, $payinfo, P$), $\mathcal{S}$ sets the payment $pay \leftarrow (\kappa, \sigma', \langle S_k, T_k, A_k\rangle_{k\in[0,V-1]}, V, C, \pi_v)$ by running the procedure

in Figure 4. $\mathcal{S}$ stores $(sid, payid, pay, payinfo)$ and sends the message $(\text{nym.send.end}, sid, \langle pay, payinfo\rangle, P)$ to $\mathcal{A}$.

Corrupt user begins spending. When the adversary $\mathcal{A}$ sends $(\text{nym.send.ini}, sid, \langle pay, payinfo\rangle, P, \mathcal{P}_k)$, $\mathcal{S}$ runs $\mathcal{F}_{\text{NYM}}$ on input that message. When $\mathcal{F}_{\text{NYM}}$ sends $(\text{nym.reply.sim}, sid, qid, l(m))$, $\mathcal{S}$ sends that message to $\mathcal{A}$.

Honest provider receives spending from corrupt user. When $\mathcal{A}$ sends the message $(\text{nym.reply.rep}, sid, qid)$, the simulator $\mathcal{S}$ runs $\mathcal{F}_{\text{NYM}}$ on input that message. When the functionality $\mathcal{F}_{\text{NYM}}$ sends $(\text{nym.send.end}, sid, \langle pay, payinfo\rangle, P)$, $\mathcal{S}$ runs $b \leftarrow \text{SpendVf}(pk, pay, payinfo)$. If $b = 0$, $\mathcal{S}$ aborts. Otherwise $\mathcal{S}$ runs the procedure in Figure 5 to check the payment and store a tuple $(pay, payinfo, \langle sk_{\mathcal{U}_j}, v, r, o_c, \langle l_k, o_{a_k}, \mu_k, o_{\mu_k}\rangle_{k=0}^{V-1}\rangle)$. After that, $\mathcal{S}$ checks that the algorithm for identification of double spenders identifies the double spender properly by running the procedure in Figure 6. $\mathcal{S}$ sets $\mathbb{D} \leftarrow \{l_k | k \in [0, V-1]\}$ and sends $(\text{ec.spend.ini}, sid, wn, V, \mathbb{D}, payinfo, P, \mathcal{P}_k)$ to $\mathcal{F}_{\text{EC}}$. When the functionality $\mathcal{F}_{\text{EC}}$ sends the message $(\text{ec.spend.sim}, sid, qid)$, $\mathcal{S}$ sends $(\text{ec.spend.rep}, sid, qid)$ to $\mathcal{F}_{\text{EC}}$.

Honest provider begins deposit of honest user $\mathcal{U}_j$ payment. If there are not corrupt authorities, $\mathcal{F}_{\text{EC}}$ sends $(\text{ec.deposit.sim}, sid, qid)$ and $\mathcal{S}$ picks up random $payinfo \leftarrow U_{info}$ and $V \leftarrow [1, L]$. If there is at least one corrupt authority, $\mathcal{F}_{\text{EC}}$ sends the message $(\text{ec.deposit.sim}, sid, qid, V, payinfo)$.

After receiving the message from $\mathcal{F}_{\text{EC}}$, $\mathcal{S}$ runs the procedure in Figure 4 to simulate an honest user payment pay. $\mathcal{S}$ runs a copy of $\mathcal{F}_{\text{BB}}$ on input $(\text{bb.write.ini}, sid, \langle pay, payinfo\rangle)$. When $\mathcal{F}_{\text{BB}}$ sends $(\text{bb.write.sim}, sid, qid)$, $\mathcal{S}$ forwards that message to $\mathcal{A}$.

Honest provider starts deposit of corrupt user $\mathcal{U}_j$ payment. When $\mathcal{F}_{\text{EC}}$ sends the message $(\text{ec.deposit.sim}, sid, qid, \mathcal{U}_j, wn, V, \mathbb{D}, payinfo, P, \mathcal{P}_k)$, $\mathcal{S}$ finds the tuple $(pay, payinfo', \langle sk_{\mathcal{U}_j}, v, r, o_c, \langle l_k, o_{a_k}, \mu_k, o_{\mu_k}\rangle_{k=0}^{V-1}\rangle)$, where pay is $(\kappa, \sigma', \langle S_k, T_k, A_k\rangle_{k\in[0,V'-1]}, V', C, \pi_v)$, such that $sk_{\mathcal{U}_j}$ is the secret key corresponding to the public key registered by the corrupt user $\mathcal{U}_j$, v is the serial number associated with the wallet wn, $V' = V$, $\langle l_k\rangle_{k\in[0,V-1]} = \mathbb{D}$, and $payinfo' = payinfo$. $\mathcal{S}$ runs $\mathcal{F}_{\text{BB}}$ on input $(\text{bb.write.ini}, sid, \langle pay, payinfo\rangle)$. When $\mathcal{F}_{\text{BB}}$ sends $(\text{bb.write.sim}, sid, qid)$, $\mathcal{S}$ forwards that message to $\mathcal{A}$. We remark that $\mathcal{F}_{\text{EC}}$ sends $(\text{ec.deposit.sim}, sid, qid, \mathcal{U}_j, wn, V, \mathbb{D}, payinfo, P, \mathcal{P}_k)$ instead $(\text{ec.deposit.sim}, sid, qid)$ when the deposit corresponds to a payment made by a corrupt user and when at least one authority is corrupt. When all the authorities are honest, the adversary in the real world does not have access to the information in $\mathcal{F}_{\text{BB}}$, and thus the simulator does not need to learn from $\mathcal{F}_{\text{EC}}$ whether a payment made by a corrupt user has been deposited.

Honest provider ends deposit. When the adversary $\mathcal{A}$ sends the message $(\text{bb.write.rep}, sid, qid)$, $\mathcal{S}$ runs $\mathcal{F}_{\text{BB}}$ on input that message. When $\mathcal{F}_{\text{BB}}$ sends $(\text{bb.write.end}, sid)$, $\mathcal{S}$ sends $(\text{ec.deposit.rep}, sid, qid)$ to $\mathcal{F}_{\text{EC}}$.

Corrupt provider begins deposit. When $\mathcal{A}$ sends $(\text{bb.write.ini}, sid, \langle pay, payinfo\rangle)$, the simulator $\mathcal{S}$ checks if a tuple $(sid, payid, pay', payinfo')$ such that $pay' = pay$ and $payinfo' = payinfo$ is stored. If it is stored, which means that this is a payment that was sent to $\mathcal{A}$ by the simulator acting as an honest user, $\mathcal{S}$ takes $payid$ from that tuple and proceeds with step 2 below. Otherwise we are in the case of a payment that a corrupt user sent to a corrupt provider, and thus $\mathcal{S}$ did not receive it during the spending phase. $\mathcal{S}$ proceeds with step 1.

Step 1. $\mathcal{S}$ proceeds as follows:

- $\mathcal{S}$ runs $b \leftarrow \text{SpendVf}(pk, pay, payinfo)$. If $b = 0$, the simulator $\mathcal{S}$ aborts.
- $\mathcal{S}$ runs the procedure in Figure 5 to store $(pay, payinfo, \langle sk_{\mathcal{U}_j}, v, r, o_c, \langle l_k, o_{a_k}, \mu_k, o_{\mu_k}\rangle_{k=0}^{V-1}\rangle)$.
- The simulator $\mathcal{S}$ checks that the algorithm for identification of double spenders identifies the double spender properly by running the procedure in Figure 6.

$\mathcal{S}$ sets $\mathbb{D} \leftarrow \{l_k | k \in [0, V-1]\}$, picks a random pseudonym $P \leftarrow \mathbb{U}_p$, takes the identity $\mathcal{P}_k$ of the corrupt provider that sent the message $(\text{bb.write.ini}, \ldots)$, and sends $(\text{ec.spend.ini}, sid, wn, V, \mathbb{D}, payinfo, P, \mathcal{P}_k)$ to $\mathcal{F}_{\text{EC}}$. When $\mathcal{F}_{\text{EC}}$ sends the message $(\text{ec.spend.sim}, sid, qid)$,

Procedure: Simulate honest user payment

- Pick random $r \leftarrow \mathbb{Z}_p$ and $r' \leftarrow \mathbb{Z}_p$.
- Compute $\sigma' = (h', s') \leftarrow (g^{r'}, g^{rr'})$.
- Compute $\kappa \leftarrow \tilde{g}^r$.
- For $k \in [0, V-1]$, pick random $S_k \leftarrow \mathbb{G}$, $T_k \leftarrow \mathbb{G}$ and $A_k \leftarrow \mathbb{G}$.
- Pick random $C \leftarrow \mathbb{G}$.
- Run the simulator $\mathcal{S}_v$ to compute a simulated proof π_v.

Figure 4: Procedure for simulating an honest user payment.

Procedure: Check Corrupt User Payment

- $\mathcal{S}$ parses pay as $(\kappa, \sigma', \langle S_k, T_k, A_k\rangle_{k\in[0,V-1]}, V, C, \pi_v)$.
- $\mathcal{S}$ runs the extractor $\mathcal{E}_v$ to extract the witness $(sk_{\mathcal{U}_j}, v, r, o_c, \langle l_k, o_{a_k}, \mu_k, o_{\mu_k}\rangle_{k=0}^{V-1})$ from the proof π_v.
- $\mathcal{S}$ parses σ' as (h', s') and computes $\hat{\sigma} = (\hat{h}, \hat{s}) = (h', s'(h')^{-r})$.
- $\mathcal{S}$ runs the verification equation $e(\hat{h}, \tilde{\alpha}\tilde{\beta}_1^{sk_{\mathcal{U}_j}}\tilde{\beta}_2^{v}) = e(\hat{s}, \tilde{g})$ of the Pointcheval-Sanders signature scheme. If the verification equation does not hold, $\mathcal{S}$ outputs failure.
- $\mathcal{S}$ finds a tuple $(\mathcal{U}_j, wn, \mathbb{R}, \langle h, com, com_1, com_2, \pi_s\rangle, \langle m_1, m_2, o, o_1, o_2\rangle)$ such that $m_1 = sk_{\mathcal{U}_j}$ and $m_2 = v$. If such a tuple is not stored, $\mathcal{S}$ outputs failure, else $\mathcal{S}$ takes $\mathcal{U}_j$ and wn from that tuple. We remark that, in this tuple, $\mathcal{U}_j$ is always the identity of a corrupt user.
- $\mathcal{S}$ finds a tuple $(sid, \mathcal{U}'_j, wn', \mathbb{S})$ such that $\mathcal{U}'_j = \mathcal{U}_j$ and $wn' = wn$. $\mathcal{S}$ outputs failure if such a tuple is not stored, or if $|\mathbb{S}| < t - \tilde{t}$, where $\tilde{t}$ is the number of corrupt authorities.
- For $k \in [0, V-1]$, $\mathcal{S}$ checks whether $S_k = \delta^{1/(v+l_k+1)}$ and $T_k = g^{sk_{\mathcal{U}_j}+R_k/(v+l_k+1)}$, where $R_k \leftarrow H'(payinfo, k)$. If any of those checks fails, $\mathcal{S}$ outputs failure.
- $\mathcal{S}$ stores the tuple $(pay, payinfo, \langle sk_{\mathcal{U}_j}, v, r, o_c, \langle l_k, o_{a_k}, \mu_k, o_{\mu_k}\rangle_{k=0}^{V-1}\rangle)$.

Figure 5: Procedure for checking a corrupt user payment.

Procedure: Check Double Spending

- $\mathcal{S}$ takes the tuple $(pay, payinfo, \langle sk_{\mathcal{U}_j}, v, r, o_c, \langle l_k, o_{a_k}, \mu_k, o_{\mu_k}\rangle_{k=0}^{V-1}\rangle)$ for the payment made by a corrupt user.
- For all the stored tuples $(pay', payinfo', \langle sk'_{\mathcal{U}_j}, v', r', o'_c, \langle l'_k, o'_{a_k}, \mu'_k, o'_{\mu_k}\rangle_{k=0}^{V'-1}\rangle)$, $\mathcal{S}$ checks whether $sk'_{\mathcal{U}_j} = sk_{\mathcal{U}_j}$, $v' = v$ and $\langle l'_k\rangle_{k\in[0,V'-1]} \cap \langle l_k\rangle_{k\in[0,V-1]} \neq \emptyset$, which means that there is a double spending. $\mathcal{S}$ sets PK to contain all the registered public keys and runs the algorithm $c \leftarrow$ Identify$(params, PK, pay, pay', payinfo, payinfo')$. Then $\mathcal{S}$ proceeds as follows:
  - If there is double spending, $\mathcal{S}$ does the following:
    - If $payinfo = payinfo'$ and $c \neq payinfo$, $\mathcal{S}$ outputs failure. We note that authorities verify that $payinfo$ contains an identifier of the provider that deposited the payment. This avoids that an honest provider can be found guilty when an adversarial provider e.g. double deposits a payment with payment information $payinfo$ that contains the identifier of an honest provider.
    - If $payinfo \neq payinfo'$ and $c \neq pk_{\mathcal{U}_j}$, where $pk_{\mathcal{U}_j}$ is the public key associated with secret key $sk_{\mathcal{U}_j}$, $\mathcal{S}$ outputs failure.
  - If there is not double spending, and if $c = pk_{\mathcal{U}_j}$, where $pk_{\mathcal{U}_j}$ is a public key associated with an honest user, $\mathcal{S}$ outputs failure. We remark that it is possible for the adversary to produce two payments where there is not double-spending, and yet the algorithm Identify detects double spending. When that happens, we must ensure that algorithm Identify does not output the public key of an honest user. We recall that authorities verify that $payinfo$ contains an identifier of the provider that deposited the payment, which also avoids an honest provider from being framed in this case.

Figure 6: Procedure for checking double spending.

$\mathcal{S}$ sends (ec.spend.rep, *sid*, *qid*) to $\mathcal{F}_{\text{EC}}$. When $\mathcal{F}_{\text{EC}}$ sends the message (ec.spend.end, *payid*, V, *payinfo*, P), $\mathcal{S}$ proceeds with step 2.

Step 2. $\mathcal{S}$ sends (ec.deposit.ini, *sid*, *payid*) to $\mathcal{F}_{\text{EC}}$. When $\mathcal{F}_{\text{EC}}$ sends (ec.deposit.sim, *sid*, *qid*), $\mathcal{S}$ runs a copy of $\mathcal{F}_{\text{BB}}$ on input the message (bb.write.ini, *sid*, $\langle pay, payinfo\rangle$). When $\mathcal{F}_{\text{BB}}$ sends the message (bb.write.sim, *sid*, *qid*), $\mathcal{S}$ forwards that message to $\mathcal{A}$.

Corrupt provider ends deposit. When the adversary $\mathcal{A}$ sends the message (bb.write.rep, *sid*, *qid*), $\mathcal{S}$ runs the copy of $\mathcal{F}_{\text{BB}}$ on input that message. When $\mathcal{F}_{\text{BB}}$ sends (bb.write.end, *sid*), $\mathcal{S}$ sends (ec.deposit.rep, *sid*, *qid*) to $\mathcal{F}_{\text{EC}}$. When $\mathcal{F}_{\text{EC}}$ sends (ec.deposit.end, *sid*, *payid*), $\mathcal{S}$ sends (bb.write.end, *sid*) to $\mathcal{A}$.

Honest authority runs deposit verification. When $\mathcal{F}_{\text{EC}}$ sends the message (ec.depvf.sim, *sid*, *qid*, $\mathbb{U}'$, d), for all $\mathcal{U}_j \in \mathbb{U}'$, $\mathcal{S}$ runs the procedure in Figure 3. After that, for $i = 1$ to d, $\mathcal{S}$ does the following:

- Send (bb.read.sim, *sid*, *qid*) to $\mathcal{A}$.
- Receive (bb.read.rep, *sid*, *qid*) from $\mathcal{A}$.

$\mathcal{S}$ sends (ec.depvf.rep, *sid*, *qid*) to $\mathcal{F}_{\text{EC}}$.

Corrupt authority starts reading of bulletin board. When $\mathcal{A}$ sends (bb.read.ini, *sid*, *i*), $\mathcal{S}$ runs $\mathcal{F}_{\text{BB}}$ on input that message. When $\mathcal{F}_{\text{BB}}$ sends (bb.read.sim, *sid*, *qid*), $\mathcal{S}$ forwards that message to $\mathcal{A}$.

Corrupt authority ends reading of bulletin board. When the adversary $\mathcal{A}$ sends (bb.read.rep, *sid*, *qid*), $\mathcal{S}$ runs $\mathcal{F}_{\text{BB}}$ on input that message. When $\mathcal{F}_{\text{BB}}$ sends (bb.read.end, *sid*, m'), $\mathcal{S}$ parses m' as $(\mathcal{P}_k, pay, payinfo)$. If $(pay, payinfo)$ is a payment that was computed by $\mathcal{S}$, $\mathcal{S}$ parses pay as $(\kappa, \sigma', \langle S_k, T_k, A_k\rangle_{k\in[0,V-1]}, V, C, \pi_v)$ and checks whether any serial number S_k is equal to any of the other serial numbers stored in the payments in $\mathcal{F}_{\text{BB}}$. If that is the case, $\mathcal{S}$ outputs failure. Otherwise $\mathcal{S}$ sends (bb.read.end, *sid*, m') to $\mathcal{A}$. We remark that payments computed by the adversary were already checked before by $\mathcal{S}$ through the procedure in Figure 6.

Theorem D.1. *When a subset of users $\mathcal{U}_j$, a subset of providers $\mathcal{P}_k$ and up to $t-1$ authorities $\mathcal{V}_i$ are corrupt, Π_{EC} securely realizes $\mathcal{F}_{\text{EC}}$ in the random oracle model and in the $(\mathcal{F}_{\text{SMT}}, \mathcal{F}_{\text{NYM}}, \mathcal{F}_{\text{KG}}, \mathcal{F}_{\text{REG}}, \mathcal{F}_{\text{BB}})$-hybrid model if the non-interactive proof of knowledge scheme is zero-knowledge and provides weak simulation extractability, the signature scheme by Pointcheval-Sanders in the RO model is unforgeable, the commmitment scheme is hiding and binding, the function in §A.6 is pseudorandom, and the hash function H' is collision-resistant.*

Proof of Theorem D.1. We show by means of a series of hybrid games that the environment $\mathcal{Z}$ cannot distinguish between the ensemble $\text{REAL}_{\Pi_{\text{EC}},\mathcal{A},\mathcal{Z}}$ and the ensemble $\text{IDEAL}_{\mathcal{F}_{\text{EC}},\mathcal{S},\mathcal{Z}}$ with non-negligible probability. We denote by $\Pr[\textbf{Game } i]$ the probability that the environment distinguishes **Game** i from the real-world protocol.

Game 0**:** This game corresponds to the execution of the real-world protocol. Therefore, $\Pr[\textbf{Game } 0] = 0$.

Game 1**:** This game proceeds as **Game** 0, except that **Game** 1 runs the extractor $\mathcal{E}_s$ for the non-interactive ZK proofs of knowledge π_s sent by the adversary. Under the weak simulation extractability property of the proof system (Definition A.8), we have that $|\Pr[\textbf{Game } 1] - \Pr[\textbf{Game } 0]| \leq \text{Adv}^{\text{ext}}_{\mathcal{A}}$.

Game 2**:** This game proceeds as **Game** 1, except that **Game** 2 outputs failure if two request messages were received from the adversary with commitments com' and com and proofs π'_s and π_s such that $com' = com$ but, after extraction of the witnesses $(m'_1, m'_2, o', o'_1, o'_2)$ from π'_s and (m_1, m_2, o, o_1, o_2) from π_s, $(m'_1, m'_2) \neq (m_1, m_2)$. Under the binding property of the commitment scheme, we have that $|\Pr[\textbf{Game } 2] - \Pr[\textbf{Game } 1]| \leq \text{Adv}^{\text{bin}}_{\mathcal{A}}$.

Proof. Given an adversary that makes **Game** 2 output failure with non-negligible probability, we construct an algorithm B that breaks the binding property of the commitment scheme with non-negligible probability. B works as follows. B receives the parameters of the Pedersen commitment scheme $par_c = (g, \gamma_1, \gamma_2)$ from the challenger. When running $\mathcal{F}_{\text{KG}}$, B uses those parameters to set the values (g, γ_1, γ_2) in the parameters $params \leftarrow (p, \mathbb{G}, \tilde{\mathbb{G}}, \mathbb{G}_t, e, g, \tilde{g}, \gamma_1, \gamma_2, \delta, L)$. When the adversary sends two requests with commitments com' and com and proofs π'_s and π_s such that $com' = com$ but, after extraction of the witnesses $(m'_1, m'_2, o', o'_1, o'_2)$ from π'_s and (m_1, m_2, o, o_1, o_2) from π_s, it holds that $(m'_1, m'_2) \neq (m_1, m_2)$, B sends $(com, m_1, m_2, o, m'_1, m'_2, o')$ to the challenger. □

Game 3**:** This game proceeds as **Game** 2, except that **Game** 3 runs the extractor $\mathcal{E}_v$ for the non-interactive ZK proofs of knowledge

π_v sent by the adversary. Under the weak simulation extractability property of the proof system (Definition A.8), we have that $|\Pr[\textbf{Game } 3] - \Pr[\textbf{Game } 2]| \leq \mathrm{Adv}^{\mathrm{ext}}_{\mathcal{A}}$.

Game 4: This game proceeds as **Game** 3, except that, after extracting the witness $(sk_{\mathcal{U}_j}, v, r, o_c, \langle l_k, o_{a_k}, \mu_k, o_{\mu_k}\rangle_{k=0}^{V-1})$ from a proof π_v, **Game** 4 takes the payment $pay = (\kappa, \sigma', \langle S_k, T_k, A_k\rangle_{k\in[0,V-1]}, V, C, \pi_v)$ that contains π_v and parses σ' as (h', s'). **Game** 4 computes $\hat{\sigma} = (\hat{h}, \hat{s}) = (h', s'(h')^{-r})$. Then **Game** 4 outputs failure if $\hat{\sigma}$ is not a valid signature. As shown below, the computation $\hat{\sigma} = (\hat{h}, \hat{s}) = (h', s'(h')^{-r})$ always produces a valid signature, and thus $|\Pr[\textbf{Game } 4] - \Pr[\textbf{Game } 3]| = 0$.

Proof. We follow the proof in [50]. We observe that, after extraction from π_v is successful, κ is of the form $\kappa \leftarrow \tilde{\alpha}\tilde{\beta}_1^{sk_{\mathcal{U}_j}}\tilde{\beta}_2^{v}\tilde{g}^{r}$. We also know that the following equality holds

$$e(h', \kappa) = e(s', \tilde{g})$$

If we replace κ by $\tilde{\alpha}\tilde{\beta}_1^{sk_{\mathcal{U}_j}}\tilde{\beta}_2^{v}\tilde{g}^{r}$, we have that

$$e(h', \tilde{\alpha}\tilde{\beta}_1^{sk_{\mathcal{U}_j}}\tilde{\beta}_2^{v}\tilde{g}^{r}) = e(s', \tilde{g})$$

If we now multiply the two sides of the equality by $e(h', \tilde{g}^{-r})$, we have that

$$e(h', \tilde{\alpha}\tilde{\beta}_1^{sk_{\mathcal{U}_j}}\tilde{\beta}_2^{v}\tilde{g}^{r})e(h', \tilde{g}^{-r}) = e(s', \tilde{g})e(h', \tilde{g}^{-r})$$

and this gives us

$$e(h', \tilde{\alpha}\tilde{\beta}_1^{sk_{\mathcal{U}_j}}\tilde{\beta}_2^{v}) = e(s'(h')^{-r}, \tilde{g})$$

which is the verification equation of the Pointcheval-Sanders signature scheme for the signature $(h', s'(h')^{-r})$. Therefore, the computation $\hat{\sigma} = (\hat{h}, \hat{s}) = (h', s'(h')^{-r})$ always produces a valid signature. □

Game 5: This game proceeds as **Game** 4, except that **Game** 5 outputs failure if, after computing the signature $\hat{\sigma} = (\hat{h}, \hat{s}) = (h', s'(h')^{-r})$ on $(sk_{\mathcal{U}_j}, v)$, it is the case that the adversary was not issued at least $t-\tilde{t}$ signatures from $t-\tilde{t}$ different authorities on $(sk_{\mathcal{U}_j}, v)$. Under the unforgeability property of Pointcheval-Sanders signatures in the random oracle model, we have that $|\Pr[\textbf{Game } 5] - \Pr[\textbf{Game } 4]| \leq \mathrm{Adv}^{\mathrm{unf}}_{\mathcal{A}} \cdot ((n-\tilde{t})!/((t-1-\tilde{t})!(n-t+1)!))$, where n is the number of authorities, t is the threshold and $\tilde{t}$ is the number of corrupt authorities.

Proof. We follow the proof in [50]. We construct an algorithm B that interacts with the challenger of the existential unforgeability game in the RO model (Definition A.14) and the adversary $\mathcal{A}$ and that shows that, if $\mathcal{A}$ makes **Game** 5 output failure with non-negligible probability, then $\mathcal{A}$ can be used by B to break the existential unforgeability property in the RO model of Pointcheval-Sanders signatures.

B receives a public key $(\theta, \tilde{\alpha}, \beta_1, \tilde{\beta}_1, \beta_2, \tilde{\beta}_2)$ from the challenger. To set up the keys when running functionality $\mathcal{F}_{\mathrm{KG}}$, B proceeds as follows.

- B uses the bilinear map setup θ to set *params*. B uses the public key received from the challenger to set the verification key $pk = (params, \tilde{\alpha}, \beta_1, \tilde{\beta}_1, \beta_2, \tilde{\beta}_2)$.
- Let $\mathbb{T}$ be the set of indices of corrupt authorities. Let $\mathbb{U}$ be a set of indices of size $t - 1 - |\mathbb{T}|$ picked at random from $[1, n] \setminus \mathbb{T}$. Let $\mathbb{S}' \leftarrow \mathbb{T} \cup \mathbb{U}$. To compute the secret keys and public keys of the authorities $\mathcal{V}_i$ such that $i \in \mathbb{S}'$, B picks random $(x_i, y_{i,1}, y_{i,2}) \leftarrow \mathbb{Z}_p$ and computes $pk_{\mathcal{V}_i} = (\tilde{\alpha}_i, \beta_{i,1}, \tilde{\beta}_{i,1}, \beta_{i,2}, \tilde{\beta}_{i,2}) \leftarrow (\tilde{g}^{x_i}, g^{y_{i,1}}, \tilde{g}^{y_{i,1}}, g^{y_{i,2}}, \tilde{g}^{y_{i,2}})$.
- Let $\mathbb{S} \leftarrow \mathbb{S}' \cup \{0\}$ and let $\mathbb{D} = [1, n] \setminus \mathbb{S}'$. To compute the public keys of the remaining authorities, i.e. the authorities in the set $\mathbb{D}$, B does the following. Let $(\tilde{\alpha}_0, \beta_{0,1}, \tilde{\beta}_{0,1}, \beta_{0,2}, \tilde{\beta}_{0,2}) \leftarrow (\tilde{\alpha}, \beta_1, \tilde{\beta}_1, \beta_2, \tilde{\beta}_2)$. For all $d \in \mathbb{D}$:
  - For all $i \in \mathbb{S}$, evaluate at d the Lagrange basis polynomials

$$l_i = [\prod_{j\in\mathbb{S}, j\neq i}(d-j)][\prod_{j\in\mathbb{S}, j\neq i}(i-j)]^{-1} \bmod p$$

  - For all $i \in \mathbb{S}$, take $(\tilde{\alpha}_i, \beta_{i,1}, \tilde{\beta}_{i,1}, \beta_{i,2}, \tilde{\beta}_{i,2})$ and then do

$$pk_{\mathcal{V}_d} = (\tilde{\alpha}_d, \beta_{d,1}, \tilde{\beta}_{d,1}, \beta_{d,2}, \tilde{\beta}_{d,2}) = (\prod_{i\in\mathbb{S}}\tilde{\alpha}_i^{l_i}, \prod_{i\in\mathbb{S}}\beta_{i,1}^{l_i}, \prod_{i\in\mathbb{S}}\tilde{\beta}_{i,1}^{l_i}, \prod_{i\in\mathbb{S}}\beta_{i,2}^{l_i}, \prod_{i\in\mathbb{S}}\tilde{\beta}_{i,2}^{l_i}).$$

To reply the random oracle queries $H(com)$ of the adversary $\mathcal{A}$, B forwards the query *com* to the random oracle provided by the challenger and sends $\mathcal{A}$ the response h given by the challenger.

When $\mathcal{A}$ sends a valid request $req \leftarrow (h, com, com_1, com_2, \pi_s)$, B runs the extractor $\mathcal{E}_s$ to extract the witness (m_1, m_2, o, o_1, o_2) from π_s. B outputs failure if two request messages were received with commitments com' and *com* and proofs π_s' and π_s such that com' = *com* but, after extraction of the witnesses from π_s' and π_s, $(m_1', m_2') \neq (m_1, m_2)$. As shown in **Game** 2, the probability that B fails is negligible if the commitment scheme is binding. This guarantees that *com* is different for each tuple of messages (m_1', m_2'), which is necessary when querying the signing oracle.

If the request is sent to an authority $\mathcal{V}_i$ such that $i \in \mathbb{U}$, B computes an issuance message by following Π_{EC} and stores $(m_1, m_2, \mathcal{V}_i)$. (We note that in this case B knows the secret key of the authority.) If the request is sent to $\mathcal{V}_d$ such that $d \in \mathbb{D}$, B proceeds as follows:

- B submits the message tuple (m_1, m_2) that was extracted by $\mathcal{E}_s$ and the commitment *com* to the signing oracle provided by the challenger. The challenger sends a signature $\sigma_0 = (h, s_0)$ and state information st'.
- For all $i \in \mathbb{S}'$, B computes a signature $\sigma_i = (h, s_i)$ by using the secret keys of authorities in $\mathbb{S}'$.
- For all $i \in \mathbb{S}$, B evaluates at d the Lagrange basis polynomials

$$l_i = [\prod_{j\in\mathbb{S}, j\neq i}(d-j)][\prod_{j\in\mathbb{S}, j\neq i}(i-j)]^{-1} \bmod p$$

- B computes the signature $\sigma_d = (h, s_d) \leftarrow (h, \prod_{i\in\mathbb{S}} s_i^{l_i})$. We note that in this computation the signature sent by the challenger is used.
- B computes $\hat{\sigma}_d = (h, s_d\beta_{d,1}^{o_1}\beta_{d,2}^{o_2})$ and includes it in the issuance message sent to $\mathcal{A}$. B stores $(m_1, m_2, \mathcal{V}_d)$.

After the adversary $\mathcal{A}$ sends a valid payment $pay \leftarrow (\kappa, \sigma', \langle S_k, T_k, A_k\rangle_{k\in[0,V-1]}, V, C, \pi_v)$, B proceeds as follows.

- B runs the extractor $\mathcal{E}_v$ to extract the witness $(sk_{\mathcal{U}_j}, v, r, o_c, \langle l_k, o_{a_k}, \mu_k, o_{\mu_k}\rangle_{k=0}^{V-1})$ from the proof π_v.
- B parses σ' as (h', s') and computes $\hat{\sigma} = (\hat{h}, \hat{s}) = (h', s'(h')^{-r})$. B runs the verification equation $e(\hat{h}, \tilde{\alpha}\tilde{\beta}_1^{sk_{\mathcal{U}_j}}\tilde{\beta}_2^{v}) = e(\hat{s}, \tilde{g})$ of the Pointcheval-Sanders signature scheme. If for any signature $\hat{\sigma}_l$ the

verification equation does not hold, B outputs failure. As shown in **Game** 4, the probability that B outputs failure is 0.

- B checks that there are at least $t - |\mathbb{T}|$ tuples $(m_1, m_2, \mathcal{V}_i)$ stored for $t - |\mathbb{T}|$ honest authorities. If that is the case, B does nothing because $\mathcal{A}$ was issued enough signatures to compute the payment. Else, if the adversary $\mathcal{A}$ received less than $t - |\mathbb{T}|$ signatures from honest authorities, but $\mathcal{A}$ did receive a signature from an authority $\mathcal{V}_d$ such that $d \in \mathbb{D}$, B fails because B had to the query signing oracle to issue that signature to the adversary and therefore he cannot use $\hat{\sigma}$ as a forgery. However, if $\mathcal{A}$ received less than $t - |\mathbb{T}|$ signatures from honest authorities, and all those authorities $\mathcal{V}_i$ are such that $i \in \mathbb{U}$, B sends $\hat{\sigma}$ to the challenger to win the existential unforgeability game.

Finally, the probability that B fails can be bound as follows. B needs to query the signing oracle of the challenger whenever $\mathcal{A}$ requests a signature from an authority $\mathcal{V}_d$ such that $d \in \mathbb{D}$. Therefore, when $\mathcal{A}$ is able to show a signature without receiving $t - |\mathbb{T}|$ signatures shares from $t - |\mathbb{T}|$ different honest authorities, B fails whenever $\mathcal{A}$ did request a signature from an authority $\mathcal{V}_d$ such that $d \in \mathbb{D}$. In the worst case, $\mathcal{A}$ received $t - 1 - |\mathbb{T}|$ signatures from $t - 1 - |\mathbb{T}|$ honest authorities. In that worst case, B only succeeds when those $t - 1 - |\mathbb{T}|$ authorities are those authorities $\mathcal{V}_i$ such that $i \in \mathbb{U}$. The probability that B succeeds, i.e. the probability that $\mathcal{A}$ picks those $t - 1 - |\mathbb{T}|$ authorities from the set of $n - |\mathbb{T}|$ authorities is given by the inverse of the number of $(t - 1 - |\mathbb{T}|)$-element combinations of $n - |\mathbb{T}|$ objects taken without repetition

$$\frac{(t - 1 - |\mathbb{T}|)!(n - t + 1!)}{(n - |\mathbb{T}|)!}$$

We remark that, in the frequent case in which $t = n$, then B succeeds with probability $1/(t - |\mathbb{T}|)$. □

Game 6**:** This game proceeds as **Game** 5, except that, after extracting the witness $(sk_{\mathcal{U}_j}, v, r, o_c, \langle l_k, o_{a_k}, \mu_k, o_{\mu_k}\rangle_{k=0}^{V-1})$ from a proof π_v, **Game** 6 takes the payment $pay = (\kappa, \sigma', \langle S_k, T_k, A_k\rangle_{k\in[0,V-1]}, V, C, \pi_v)$ that contains π_v and the associated payment information *payinfo* and computes the serial numbers $S'_k \leftarrow \delta^{1/(v+l_k+1)}$ and the double spending tags $T'_k \leftarrow g^{sk_{\mathcal{U}_j}+R_k/(v+l_k+1)}$, where $R_k = H'(payinfo, k)$. Then **Game** 6 outputs failure if, for any $k \in [0, V-1]$, $S'_k \neq S_k$ or $T'_k \neq T_k$. Under the hardness of the discrete logarithm problem, $|\Pr[\textbf{Game } 6] - \Pr[\textbf{Game } 5]| \leq \mathrm{Adv}^{\mathrm{dlog}}_{\mathcal{A}}$.

Proof. We have an adversary $\mathcal{A}$ that, with non-negligible probability, sends payment information *payinfo* and a payment $pay = (\kappa, \sigma', \langle S_k, T_k, A_k\rangle_{k\in[0,V-1]}, V, C, \pi_v)$ such that, after extracting from π_v the witness $(sk_{\mathcal{U}_j}, v, r, o_c, \langle l_k, o_{a_k}, \mu_k, o_{\mu_k}\rangle_{k=0}^{V-1})$ and computing $S'_k \leftarrow \delta^{1/(v+l_k+1)}$ and $T'_k \leftarrow g^{sk_{\mathcal{U}_j}+R_k/(v+l_k+1)}$ for all $k \in [0, V-1]$, we have that $S'_k \neq S_k$ or $T'_k \neq T_k$ for some $k \in [0, V-1]$. We construct an algorithm B that uses that adversary to solve the discrete logarithm problem.

B works as follows. Given an instance (h, h^x) of the discrete logarithm problem, when running the functionality $\mathcal{F}_{\mathrm{KG}}$ to set up the parameters $params \leftarrow (p, \mathbb{G}, \tilde{\mathbb{G}}, \mathbb{G}_t, e, g, \tilde{g}, \gamma_1, \gamma_2, \delta, L)$, B sets $g \leftarrow h$ and $\gamma_1 \leftarrow h^x$. When the adversary outputs a payment pay that fulfills the condition described above for a certain $k \in [0, V-1]$, B outputs

$$x \leftarrow \frac{(o_{a_k} + o_c)\mu_k + o_{\mu_k}}{1 - ((v + l_k + 1)\mu_k)}$$

We show that the discrete logarithm x is computed correctly as follows. The non-interactive ZK proof of knowledge π_v is described by

$$\begin{aligned}\pi_v = \mathrm{NIZK}\{&(sk_{\mathcal{U}_j}, v, r, o_c, \langle l_k, o_{a_k}, \mu_k, o_{\mu_k}\rangle_{k=0}^{V-1}) : \\ &\kappa = \tilde{\alpha}\tilde{\beta}_1^{sk_{\mathcal{U}_j}}\tilde{\beta}_2^{v}\tilde{g}^{r} \ \wedge\ C = g^{o_c}\gamma_1^{v} \wedge \\ &\langle A_k = g^{o_{a_k}}\gamma_1^{l_k} \ \wedge\ l_k \in [0, L-1] \wedge \\ &S_k = \delta^{\mu_k} \ \wedge\ \gamma_1 = (A_k C\gamma_1)^{\mu_k} g^{o_{\mu_k}} \wedge \\ &T_k = g^{sk_{\mathcal{U}_j}}(g^{R_k})^{\mu_k} \ \rangle_{k\in[0,V-1]}\}\end{aligned}$$

After successful extraction of the witness $(sk_{\mathcal{U}_j}, v, r, o_c, \langle l_k, o_{a_k}, \mu_k, o_{\mu_k}\rangle_{k=0}^{V-1})$, we know that the statements proven by π_v hold. In the case that $S'_k \neq S_k$ for some $k \in [0, V-1]$, we have that

$$\delta^{1/(v+l_k+1)} \neq \delta^{\mu_k}$$

and thus $1/(v + l_k + 1) \neq \mu_k$. In the case that $T'_k \neq T_k$ for some $k \in [0, V-1]$, we have that

$$g^{sk_{\mathcal{U}_j}+R_k/(v+l_k+1)} \neq g^{sk_{\mathcal{U}_j}}(g^{R_k})^{\mu_k}$$

and thus we can also deduce that $1/(v + l_k + 1) \neq \mu_k$.

We also have the following:

$$\begin{aligned}\gamma_1 &= (A_k C\gamma_1)^{\mu_k} g^{o_{\mu_k}} \\ &= (g^{o_{a_k}}\gamma_1^{l_k} g^{o_c}\gamma_1^{v}\gamma_1)^{\mu_k} g^{o_{\mu_k}} \\ &= \gamma_1^{(v+l_k+1)\mu_k} g^{(o_{a_k}+o_c)\mu_k + o_{\mu_k}}\end{aligned}$$

and therefore

$$\gamma_1 = g^{\frac{(o_{a_k}+o_c)\mu_k + o_{\mu_k}}{1-((v+l_k+1)\mu_k)}}$$

This equation shows that, when $1/(v+l_k+1) \neq \mu_k$, we can compute the discrete logarithm x as described above. □

Game 7**:** This game proceeds as **Game** 6, except that in **Game** 7 the non-interactive ZK proofs of knowledge π_v that are sent to the adversary are replaced by simulated proofs computed by the simulator $\mathcal{S}_v$. Under the zero-knowledge property of the proof system (see Definition A.7), we have that $|\Pr[\textbf{Game } 7] - \Pr[\textbf{Game } 6]| \leq \mathrm{Adv}^{\mathrm{zk}}_{\mathcal{A}}$.

Game 8**:** This game proceeds as **Game** 7, except that in **Game** 8, for the payments $pay \leftarrow (\kappa, \sigma', \langle S_k, T_k, A_k\rangle_{k\in[0,V-1]}, V, C, \pi_v)$ that are sent to the adversary, the values κ and σ' are computed as follows:

- Pick random $t \leftarrow \mathbb{Z}_p$ and $t' \leftarrow \mathbb{Z}_p$.
- Compute $\sigma' = (h', s') \leftarrow (g^{t'}, g^{tt'})$.
- Compute $\kappa \leftarrow \tilde{g}^{t}$.

As shown below, $|\Pr[\textbf{Game } 8] - \Pr[\textbf{Game } 7]| = 0$.

Proof. This proof follows the proof in [50]. We show that values κ and σ' follow the same distribution as the ones computed by the honest user in the real-world protocol. Observe that the honest user computes the following:

- Pick random $r \leftarrow \mathbb{Z}_p$ and $r' \leftarrow \mathbb{Z}_p$.

- Set $\sigma' = (h', s') \leftarrow (h^{r'}, s^{r'}(h')^r)$, where we have that

$$(h^{r'}, s^{r'}(h')^r) = (h^{r'}, h^{(x+sk_{\mathcal{U}_j} y_1 + v y_2 + r) r'}) = (g^{ur'}, g^{(x+sk_{\mathcal{U}_j} y_1 + v y_2 + r) u r'})$$

- Set $\kappa \leftarrow \alpha \beta_1^{sk_{\mathcal{U}_j}} \beta_2^{v} \tilde{g}^{r} = \tilde{g}^{x+sk_{\mathcal{U}_j} y_1 + v y_2 + r}$.

Therefore, t corresponds to $(x + sk_{\mathcal{U}_j} y_1 + v y_2 + r_l)$ and t' corresponds to ur', where u is a random value such that $h = g^u$. Both $(x + sk_{\mathcal{U}_j} y_1 + v y_2 + r)$ and ur' are random. Observe as well that the verification equation $e(h', \kappa) = e(s', \tilde{g})$ still holds because $e(g^{t'}, \tilde{g}^t) = e(g^{tt'}, \tilde{g})$. □

Game 9**:** This game proceeds as **Game** 8, except that in **Game** 9 the non-interactive ZK proofs of knowledge π_s that are sent to the adversary are replaced by simulated proofs computed by the simulator $\mathcal{S}_s$. Under the zero-knowledge property of the proof system (see Definition A.7), we have that $|\Pr[\textbf{Game } 9] - \Pr[\textbf{Game } 8]| \leq \mathrm{Adv}^{\mathsf{zk}}_{\mathcal{A}}$.

Game 10**:** This game proceeds as **Game** 9, except that in **Game** 10, in each request that is sent to the adversary, the values (com_1, com_2) are replaced by $com_1 \leftarrow g^{o_1} h^{m_1}$ and $com_2 \leftarrow g^{o_2} h^{m_2}$, where (o_1, m_1, o_2, m_2) are random values in $\mathbb{Z}_p$. At this point, the non-interactive ZK proofs of knowledge π_s are simulated proofs of false statements. We also remark that, after **Game** 8, the computation of payment messages does not use the signatures obtained in the issuance phase. Since the values (com_1, com_2) are uniformly distributed at random, this change does not alter the view of the environment and we have that $|\Pr[\textbf{Game } 10] - \Pr[\textbf{Game } 9]| = 0$.

Game 11**:** This game proceeds as **Game** 10, except that in **Game** 11, in each request that is sent to the adversary, the value com is replaced by picking random $com \leftarrow \mathbb{G}$. Under the hiding property of the commitment scheme, $|\Pr[\textbf{Game } 11] - \Pr[\textbf{Game } 10]| \leq N \cdot \mathrm{Adv}^{\mathsf{hid}}_{\mathcal{A}}$, where N is the number of commitments com sent to the adversary. Since the Pedersen commitment scheme is perfectly hiding, $|\Pr[\textbf{Game } 11] - \Pr[\textbf{Game } 10]| = 0$.

Proof. The proof uses a sequence of games **Game** 10.i, for $i = 0$ to N. **Game** 10.0 is equal to **Game** 10, whereas **Game** 10.N is equal to **Game** 11. In **Game** 10.i, the first i commitments sent to the adversary are set to random values, whereas the remaining ones are set as in **Game** 10.

Given an adversary that distinguishes between **Game** 10.i and **Game** 10.$(i+1)$ with non-negligible probability, we construct an algorithm B that breaks the hiding property of the commitment scheme. B works as follows. B receives the parameters of the Pedersen commitment scheme $par_c = (g, \gamma_1, \gamma_2)$ from the challenger. When running $\mathcal{F}_{\mathrm{KG}}$, B uses those parameters to set the values (g, γ_1, γ_2) in the parameters $params \leftarrow (p, \mathbb{G}, \tilde{\mathbb{G}}, \mathbb{G}_t, e, g, \tilde{g}, \gamma_1, \gamma_2, \delta, L)$. To compute the first i commitments, B sets com to random. To compute the commitment $i+1$, B sends the messages $(sk_{\mathcal{U}_j}, v)$ to the challenger. B sets com to the challenge commitment received from the challenger. As can be seen, if the challenge commitment commits to $(sk_{\mathcal{U}_j}, v)$, then we are in **Game** 10.i, whereas if the challenge commitment commits to a random message, then we are in **Game** 10.$(i+1)$. The remaining commitments are computed as in **Game** 10. B sends the adversarial guess to distinguish between **Game** 10.i and **Game** 10.$(i+1)$ to the challenger of the hiding game. □

Game 12**:** This game proceeds as **Game** 11, except that in **Game** 12, for the payments $pay \leftarrow (\kappa, \sigma', \langle S_k, T_k, A_k \rangle_{k \in [0, V-1]}, V, C, \pi_v)$ that are sent to the adversary, for all $k \in [0, V-1]$, the values $A_k \leftarrow \mathbb{G}$ are set to random elements in $\mathbb{G}$. The value C is also set to a random element in $\mathbb{G}$. Under the hiding property of the commitment scheme, we have that $|\Pr[\textbf{Game } 12] - \Pr[\textbf{Game } 11]| \leq N \cdot \mathrm{Adv}^{\mathsf{hid}}_{\mathcal{A}}$, where N is the number of commitments C and A_k sent to the adversary. Since the Pedersen commitment scheme is perfectly hiding, $|\Pr[\textbf{Game } 12] - \Pr[\textbf{Game } 11]| = 0$. We omit the proof, which is similar to the proof of indistinguishability between **Game** 10 and **Game** 11.

Game 13**:** This game proceeds as **Game** 12, except that in **Game** 13, for the payments $pay \leftarrow (\kappa, \sigma', \langle S_k, T_k, A_k \rangle_{k \in [0, V-1]}, V, C, \pi_v)$ that are sent to the adversary, for all $k \in [0, V-1]$, the values S_k and T_k are computed by doing $S_k \leftarrow \delta^{r_k}$ and $T_k \leftarrow g^{sk_{\mathcal{U}_j}} (g^{r_k})^{R_k}$, where $r_k \leftarrow \mathbb{Z}_p$ is picked up randomly. Under the pseudorandomness property of the pseudorandom function, we have that $|\Pr[\textbf{Game } 13] - \Pr[\textbf{Game } 12]| \leq \mathrm{Adv}^{\mathsf{pseu}}_{\mathcal{A}}$.

Proof. Given an adversary that is able to distinguish **Game** 12 from **Game** 13 with non-negligible probability, we construct an algorithm B that breaks the pseudorandomness property of the pseudorandom function $f_{\mathbb{G},p,g,s}(\cdot)$ described in §A.6. B receives from the challenger the parameters $(\mathbb{G}, p, g)$. When running $\mathcal{F}_{\mathrm{KG}}$, B picks up random $a \in \mathbb{Z}_p$, computes $\delta \leftarrow g^{1/a}$ and sets the parameters $params \leftarrow (p, \mathbb{G}, \tilde{\mathbb{G}}, \mathbb{G}_t, e, g, \tilde{g}, \gamma_1, \gamma_2, \delta, L)$. We remark that, after **Game** 11, request messages are computed without requiring knowledge of the coin secret v, and so B does not need to know the secret s of the pseudorandom function to compute them. Similarly, after **Game** 12, payment messages are computed without requiring knowledge of the coin secret v. To compute a payment $pay \leftarrow (\kappa, \sigma', \langle S_k, T_k, A_k \rangle_{k \in [0, V-1]}, V, C, \pi_v)$ to be sent to the adversary, B follows the changes described up to **Game** 12 and, additionally, to compute $\langle S_k, T_k \rangle_{k \in [0, V-1]}$, B does the following. For all $k \in [0, V-1]$, B sends the coin index l_k to the oracle of the challenger, which provides a response Z_k. B sets $T_k \leftarrow g^{sk_{\mathcal{U}_j}} (Z_k)^{R_k}$ and $S_k \leftarrow Z_k^{1/a}$. As can be seen, if $Z_k = f_{\mathbb{G},p,g,s}(l_k)$, T_k and S_k are computed as in **Game** 12, whereas if Z_k is random, Z_k are computed as in **Game** 13. Therefore, B uses the guess of the adversary to distinguish between **Game** 12 and **Game** 13 in order to break the pseudorandomness property of the pseudorandom function. □

Game 14**:** This game proceeds as **Game** 13, except that in **Game** 14, for the payments $pay \leftarrow (\kappa, \sigma', \langle S_k, T_k, A_k \rangle_{k \in [0, V-1]}, V, C, \pi_v)$ that are sent to the adversary, for all $k \in [0, V-1]$, the values S_k and T_k are computed by picking random $S_k \leftarrow \mathbb{G}$ and $T_k \leftarrow \mathbb{G}$. Under the external Diffie-Hellman (XDH) assumption in $\mathbb{G}$, we have that $|\Pr[\textbf{Game } 14] - \Pr[\textbf{Game } 13]| \leq N_s \cdot \mathrm{Adv}^{\mathsf{xdh}}_{\mathcal{A}}$, where N_s is the number of serial numbers and double spending tags sent to the adversary.

Proof. The proof uses a sequence of games **Game** 13.i, for $i = 0$ to N_s. **Game** 13.0 is equal to **Game** 13, whereas **Game** 13.(N_s) is equal to **Game** 14. In **Game** 13.i, the values S_k and T_k of the first i payments sent to the adversary are set to random values, whereas in the remaining payments they are set as in **Game** 13.

Given an adversary that distinguishes between **Game** 13.i and **Game** 13.$(i+1)$ with non-negligible probability, we construct an

algorithm B that uses that adversary to solve the XDH problem with non-negligible probability. B works as follows. Given an instance (h, h^a, h^b, Z) of the XDH problem in $\mathbb{G}$, when running $\mathcal{F}_{\text{KG}}$ to set up the parameters $params \leftarrow (p, \mathbb{G}, \tilde{\mathbb{G}}, \mathbb{G}_t, e, g, \tilde{g}, \gamma_1, \gamma_2, \delta, L)$, B sets $g \leftarrow h^a$ and $\delta \leftarrow h$. When setting the $i+1$ serial number and double spending tag sent to the adversary, B sets $S_k \leftarrow h^b$ and $T_k \leftarrow g^{sk_{\mathcal{U}_j}}(Z)^{R_k}$. As can be seen, when Z is random, S_k and T_k are random values and we are thus in **Game** 13.$(i+1)$. In contrast, when $Z = h^{ab}$, we have that $S_k = h^b = \delta^b$ and $T_k = (h^a)^{sk_{\mathcal{U}_j}}(h^{ab})^{R_k} = (g)^{sk_{\mathcal{U}_j}}(g^b)^{R_k}$, and thus the distribution is equal to that of **Game** 13.i. Therefore, B can use the guess of the adversary to distinguish between **Game** 13.i and **Game** 13.$(i+1)$ in order to solve the XDH problem in $\mathbb{G}$ with non-negligible probability. □

Game 15**:** This game proceeds as **Game** 14, except that **Game** 15 checks that algorithm Identify identifies the double spender when there is double spending. To do that, when the adversary sends two valid payments $(pay, payinfo)$ and $(pay', payinfo')$, after extracting the witness $\langle sk_{\mathcal{U}_j}, v, r, o_c, \langle l_k, o_{a_k}, \mu_k, o_{\mu_k}\rangle_{k=0}^{V-1}\rangle$ from the proofs $\pi_v \in pay$ and $\langle sk'_{\mathcal{U}_j}, v', r', o'_c, \langle l'_k, o'_{a_k}, \mu'_k, o'_{\mu_k}\rangle_{k=0}^{V'-1}\rangle$ from the proof $\pi'_v \in pay'$, **Game** 15 checks whether $sk'_{\mathcal{U}_j} = sk_{\mathcal{U}_j}$, $v' = v$ and $\langle l'_k\rangle_{k\in[0,V'-1]} \cap \langle l_k\rangle_{k\in[0,V-1]} \neq \emptyset$, which means that there is a double spending. In that case **Game** 15 sets PK to contain all the registered public keys and runs the algorithm $c \leftarrow$ Identify$(params, PK, pay, pay', payinfo, payinfo')$. Then **Game** 15 does the following:

- If $payinfo = payinfo'$ and $c \neq payinfo$, **Game** 15 outputs failure.
- If $payinfo \neq payinfo'$ and $c \neq pk_{\mathcal{U}_j}$, where $pk_{\mathcal{U}_j}$ is the public key associated with secret key $sk_{\mathcal{U}_j}$, **Game** 15 outputs failure.

The probability that **Game** 15 fails is negligible under the collision-resistance property of the hash function H', i.e. we have that $|\Pr[\textbf{Game } 15] - \Pr[\textbf{Game } 14]| \leq \text{Adv}^{\text{col-res}}_{\mathcal{A}}$.

Proof. In **Game** 6, we have shown that, if a payment is valid, under the hardness of the discrete logarithm assumption, the serial numbers S_k and the double-spending tags T_k are correctly computed. Hence, if there are two payments with witnesses $\langle sk_{\mathcal{U}_j}, v, r, o_c, \langle l_k, o_{a_k}, \mu_k, o_{\mu_k}\rangle_{k=0}^{V-1}\rangle$ and $\langle sk'_{\mathcal{U}_j}, v', r', o'_c, \langle l'_k, o'_{a_k}, \mu'_k, o'_{\mu_k}\rangle_{k=0}^{V'-1}\rangle$ such that $sk'_{\mathcal{U}_j} = sk_{\mathcal{U}_j}$, $v' = v$ and $\langle l'_k\rangle_{k\in[0,V'-1]} \cap \langle l_k\rangle_{k\in[0,V-1]} \neq \emptyset$, the proof in **Game** 6 guarantees that, for those coin indices such that $l'_{k'} = l_k$ (where $k' \in [0, V'-1]$ and $k \in [0, V-1]$), it is the case that $S'_{k'} = S_k = \delta^{1/(v+l_k+1)}$. Therefore, the algorithm Identify always detects double spending.

After detecting double spending, the algorithm Identify checks if $payinfo = payinfo'$ and in that case sets $c = payinfo$. Therefore, the first condition under which **Game** 15 fails never happens.

If $payinfo \neq payinfo'$, algorithm Identify computes

$$pk_{\mathcal{U}_j} \leftarrow ((T'_{k'})^{R_k} / T_k^{R'_{k'}})^{(R_k - R'_{k'})^{-1}}$$

We have that $R_k = H'(payinfo, k)$ and $R'_{k'} = H'(payinfo', k')$. The proof in **Game** 6 also guarantees that the double spending tags are correctly computed. Hence, we know $T_k = g^{sk_{\mathcal{U}_j}}(g^{1/(v+l_k+1)})^{R_k}$ and $T'_{k'} = g^{sk_{\mathcal{U}_j}}(g^{1/(v+l_k+1)})^{R'_{k'}}$. Let $Z = g^{1/(v+l_k+1)}$. We have that

$$\begin{aligned}
pk_{\mathcal{U}_j} &= ((T'_{k'})^{R_k} / T_k^{R'_{k'}})^{(R_k - R'_{k'})^{-1}} \\
&= ((g^{sk_{\mathcal{U}_j}}(Z)^{R'_{k'}})^{R_k} / (g^{sk_{\mathcal{U}_j}}(Z)^{R_k})^{R'_{k'}})^{(R_k - R'_{k'})^{-1}} \\
&= ((g^{sk_{\mathcal{U}_j} R_k}(Z)^{R'_{k'} R_k}) / (g^{sk_{\mathcal{U}_j} R'_{k'}}(Z)^{R_k R'_{k'}}))^{(R_k - R'_{k'})^{-1}} \\
&= (g^{sk_{\mathcal{U}_j} R_k} / g^{sk_{\mathcal{U}_j} R'_{k'}})^{(R_k - R'_{k'})^{-1}} \\
&= (g^{sk_{\mathcal{U}_j}(R_k - R'_{k'})})^{(R_k - R'_{k'})^{-1}} \\
&= g^{sk_{\mathcal{U}_j}}
\end{aligned}$$

Therefore, algorithm Identify outputs the public key of the double spender, except when $R_k = R'_{k'}$. Given that $R_k = H'(payinfo, k)$ and $R'_{k'} = H'(payinfo', k')$, if $R_k = R'_{k'}$, a collision for the hash function H' has been found. □

Game 16**:** This game proceeds as **Game** 15, except that, even if there is not double spending, **Game** 16 sets PK to contain all the registered public keys and runs the algorithm $c \leftarrow$ Identify$(params, PK, pay, pay', payinfo, payinfo')$. If it is the case that $c = pk_{\mathcal{U}_j}$, where $pk_{\mathcal{U}_j}$ is a public key associated with an honest user, **Game** 16 outputs failure. We show that **Game** 16 outputs failure with negligible probability under the hardness of the discrete logarithm problem, i.e. $|\Pr[\textbf{Game } 16] - \Pr[\textbf{Game } 15]| \leq N_u \cdot \text{Adv}^{\text{dlog}}_{\mathcal{A}}$, where N_u is the number of public keys of honest users.

Proof. Given an adversary that makes **Game** 16 fail with non-negligible probability, we construct an algorithm B that solves the discrete logarithm problem with non-negligible probability. B works as follows. B receives an instance (h, h^x) of the discrete logarithm problem from the challenger. When running $\mathcal{F}_{\text{KG}}$, B sets $g \leftarrow h$ and sets the parameters $params \leftarrow (p, \mathbb{G}, \tilde{\mathbb{G}}, \mathbb{G}_t, e, g, \tilde{g}, \gamma_1, \gamma_2, \delta, L)$. B picks randomly an honest user $\mathcal{U}_j$ and, when that user registers her public key, B sets $pk_{\mathcal{U}_j} \leftarrow h^x$. We remark that, since **Game** 9, knowledge of $sk_{\mathcal{U}_j}$ is not needed to compute request messages. We also remark that, since **Game** 8, knowledge of $sk_{\mathcal{U}_j}$ is not needed to compute payment messages. Therefore, B can simulate those messages without knowledge of $sk_{\mathcal{U}_j}$.

At some point B receives from the adversary two payments $(pay, payinfo)$ and $(pay', payinfo')$ that make **Game** 16 fail. If the public key $pk_{\mathcal{U}_j}$ obtained after running Identify is different from the value h^x received from the challenger, B fails. Otherwise B computes $sk_{\mathcal{U}_j}$ as follows. First, B extracts the witness $\langle sk, v, r, o_c, \langle l_k, o_{a_k}, \mu_k, o_{\mu_k}\rangle_{k=0}^{V-1}\rangle$ from the proofs $\pi_v \in pay$ and $\langle sk', v', r', o'_c, \langle l'_k, o'_{a_k}, \mu'_k, o'_{\mu_k}\rangle_{k=0}^{V'-1}\rangle$ from the proof $\pi'_v \in pay'$. Given that double spending has been detected, and that in **Game** 6 we proved that serial numbers and double spending tags are correctly computed, we know that there is $S_k \in pay$ and $S'_{k'} \in pay'$ such that $S_k = S'_{k'} = \delta^b$, where $b = (1/(v + l_k + 1)) = (1/(v' + l'_{k'} + 1))$. Therefore, we also know that the double spending tags are of the form $T_k = g^{sk + bR_k}$ and $T'_{k'} = g^{sk' + bR'_{k'}}$. From the computation of algorithm Identify,

we have that

$$\begin{aligned} pk_{\mathcal{U}_j} &= ((T'_{k'})^{R_k} / T_k^{R'_{k'}})^{(R_k - R'_{k'})^{-1}} \\ &= ((g^{sk' + bR'_{k'}})^{R_k} / (g^{sk + bR_k})^{R'_{k'}})^{(R_k - R'_{k'})^{-1}} \\ &= ((g^{R_k sk' + bR_k R'_{k'}}) / (g^{R'_{k'} sk + bR_k R'_{k'}}))^{(R_k - R'_{k'})^{-1}} \\ &= (g^{R_k sk' - R'_{k'} sk})^{(R_k - R'_{k'})^{-1}} \\ &= (g^{(R_k sk' - R'_{k'} sk)/(R_k - R'_{k'})}) \end{aligned}$$

Therefore, B computes $x \leftarrow (R_k sk' - R'_{k'} sk)/(R_k - R'_{k'})$ to solve the discrete logarithm problem. If the adversary succeeds with probability α, B succeeds with probability α/N_u, where N_u is the number of public keys of honest users. □

Game 17**:** This game proceeds as **Game** 16, except that **Game** 17 outputs failure when the serial number of a payment computed by an honest user is equal to a serial number of another payment. The probability that two serial numbers have the same value is bounded by $N_s/|\mathbb{G}|$, where N_s is the number of serial numbers and $|\mathbb{G}|$ is the size of $\mathbb{G}$. Additionally, we show below that the probability that an adversarial user computes a payment with a serial number that is equal to a serial number in a payment computed by an honest user is negligible thanks to the hardness of the discrete logarithm problem. Therefore, we have that $|\Pr[\textbf{Game } 17] - \Pr[\textbf{Game } 16]| \leq N_s/|\mathbb{G}| + N_s \cdot \mathrm{Adv}^{\mathsf{dlog}}_{\mathcal{A}}$.

Proof. Given an adversary that makes **Game** 17 fail with non-negligible probability, we construct an algorithm B that solves the discrete logarithm problem with non-negligible probability. B works as follows. B receives an instance (h, h^x) of the discrete logarithm problem from the challenger. When running $\mathcal{F}_{\mathrm{KG}}$, B sets $\delta \leftarrow h$ and sets the parameters $params \leftarrow (p, \mathbb{G}, \tilde{\mathbb{G}}, \mathbb{G}_t, e, g, \tilde{g}, \gamma_1, \gamma_2, \delta, L)$. B picks randomly a serial number in a payment computed by an honest user and sets $S_k \leftarrow h^x$. We recall that, since **Game** 14, serial numbers in payments computed by honest users are random values in $\mathbb{G}$.

At some point, B finds that a payment received from the adversary has a serial number that is equal to a serial number in a payment computed by an honest user. If that serial number is not equal to h^x, B fails. Otherwise B extracts the witness $\langle sk, v, r, o_c, \langle l_k, o_{a_k}, \mu_k, o_{\mu_k} \rangle_{k=0}^{V-1} \rangle$ from the proof π_v in the payment pay sent by the adversary. Thanks to the proof in **Game** 6, we know that the serial number $S_k = h^x$ in pay is of the form $S_k = \delta^{1/(v+l_k+1)}$. Therefore, B outputs $x \leftarrow 1/(v+l_k+1)$. If the adversary succeeds with probability α, B succeeds with probability α/N_s, where N_s is the number of serial numbers. □

The distribution of **Game** 17 is identical to that of our simulation. In **Game** 17, the request message is computed without knowledge of the values $sk_{\mathcal{U}_j}$ and v. The payment is computed without knowledge of the signatures and without knowledge of the signed messages $sk_{\mathcal{U}_j}$ and v. Additionally, it is guaranteed that the adversary cannot compute a payment on a wallet defined by $sk_{\mathcal{U}_j}$ and v unless the adversary obtained enough signatures from honest authorities. It is also guaranteed that, if the adversary double spends coins, then an adversarial user or provider will be identified. Moreover, it is guaranteed that an honest user or provider will not be found guilty of double spending. The overall advantage of the environment to distinguish between the real and the ideal protocol is $|\Pr[\textbf{Game } 17] - \Pr[\textbf{Game } 0]| \leq \mathrm{Adv}^{\mathsf{ext}}_{\mathcal{A}} + \mathrm{Adv}^{\mathsf{bin}}_{\mathcal{A}} + \mathrm{Adv}^{\mathsf{unf}}_{\mathcal{A}} \cdot ((n - \tilde{t})!/((t-1-\tilde{t})!(n-t+1)!)) + \mathrm{Adv}^{\mathsf{zk}}_{\mathcal{A}} + (N_u + N_s + 1) \cdot \mathrm{Adv}^{\mathsf{dlog}}_{\mathcal{A}} + N_s/|\mathbb{G}| + \mathrm{Adv}^{\mathsf{pseu}}_{\mathcal{A}} + N_s \cdot \mathrm{Adv}^{\mathsf{xdh}}_{\mathcal{A}} + \mathrm{Adv}^{\mathsf{col-res}}_{\mathcal{A}}$, where N_u is the number of users and N_s is the number of serial numbers. The discrete logarithm assumption are implied by the DDHI assumption, which is used to prove that the function in §A.6 is pseudorandom. This concludes the proof of Theorem D.1.

E SECURITY PROOF FOR OUR DIVISIBLE E-CASH SCHEME

In §D, we provide a detailed security proof for our compact EC scheme. In this section, we analyze the security of our divisible EC scheme, but we omit a full proof. Instead, we discuss the points where the security analysis of our divisible EC scheme differs from the analysis of our compact EC scheme.

For the interfaces ec.setup, ec.register, ec.request, and ec.issue, the simulator for the divisible EC scheme is equal to that of our compact EC scheme. We recall that both schemes use the same algorithms for the ec.register, ec.request and ec.issue interfaces. For the ec.setup interface, although algorithm Setup is different in our schemes, the algorithm is run in both cases by $\mathcal{F}_{\mathrm{KG}}$, as specified in our simulation.

In the spending phase, the simulator for the divisible EC scheme, like in the compact EC scheme, outputs failure when the adversary submits a payment such that the simulator fails to extract a valid PS signature, or when the adversary did not receive signatures from $t - \tilde{t}$ authorities on the signed messages. Additionally, the simulator for the divisible EC scheme outputs failure if it is unable to extract a SPS signature on $(\varsigma_{l+V-1}, \theta_{l+V-1})$, where $(\varsigma_{l+V-1}, \theta_{l+V-1})$ are part of the public parameters. It also outputs failure if the extracted (ς_l, θ_l) are not part of the public parameters. In the security proof, the probability that extraction fails is negligible thanks to the weak simulation extractability property of the ZK argument π_v, the existential unforgeability of the SPS signature scheme, and the BDHI assumption. All these arguments ensure that $\phi_{V,l}$ and $\varphi_{V,l}$ are computed correctly.

In the spending phase, to simulate a payment $(\kappa, \sigma', \phi_{V,l}, \varphi_{V,l}, R, \pi_v, V)$, the values κ and σ' for the proof of possession of a PS signature are simulated like in the simulator for our compact EC scheme (see Figure 4). The proof π_v is also simulated by using the simulator S_v. The values $\phi_{V,l}$ and $\varphi_{V,l}$ are set to random. In the security proof, it can be shown that a simulated payment is indistinguishable from an honestly computed payment under the N-MXDH' assumption. The proof is similar to one given in [49]. We remark that, in the proof in [49], an element of the N-MXDH' instance is needed to simulate the request message because the element $U_2 = u_2^v$ is revealed to the bank, whereas in our case that is not needed thanks to the hiding property of the commitment scheme.

In the deposit phase, the behavior of both simulators is similar, taking into account that the serial numbers and double spending tags are computed differently. The simulator also outputs failure if double spending happened but the identification algorithm does not identify the double spender. In the security proof, after it is

ensured, as described above, that the ElGamal encryptions $\phi_{V,l}$ and $\varphi_{V,l}$ sent by the adversary are computed correctly, we know that serial numbers and double spending tags can be retrieved. Then we can prove, as in the case of the compact EC scheme, that the user guilty of double spending can be identified under the collision resistance property of the hash function H'.

The simulator also outputs failure when there is not double spending, but an honest user is found guilty. In the security proof, like in our EC scheme, we can show that the simulator fails with negligible probability under the hardness of the discrete logarithm problem.

F COMPACT E-CASH WITH RANGE PROOF INSTANTIATION

The zero-knowledge argument of knowledge in algorithm Spend involves a statement $l \in [0, L-1]$ to prove the validity of the index of the spent coin. To implement it, one option is to use the set membership proof described in [12]. This proof consists in proving possession of a signature that signs the index l. We use the Pointcheval-Sanders signature scheme to instantiate it. Algorithm Setup is extended to compute a signing key pair and signatures on all the values in the range $[0, L-1]$. Algorithm Spend is extended to include a zero-knowledge argument of knowledge of the signature that signs l, and algorithm SpendVf is extended to verify it. The modified algorithms work as follows:

Setup$(1^k, L)$. Execute the following steps:

- Run $(p, \mathbb{G}, \tilde{\mathbb{G}}, \mathbb{G}_t, e, g, \tilde{g}) \leftarrow \mathcal{G}(1^k)$.
- Pick 3 random generators $(\gamma_1, \gamma_2, \delta) \leftarrow \mathbb{G}$.
- Run $(sk, pk) \leftarrow$ KeyGen$(1^k, 1)$, i.e., pick random secret key $(x, y) \leftarrow \mathbb{Z}_p^2$ and output the secret key $sk = (x, y)$ and the public key $pk = (\tilde{\alpha}_{sm}, \tilde{\beta}_{sm}) \leftarrow (\tilde{g}^x, \tilde{g}^y)$.
- For all $l \in [0, L-1]$, compute $\sigma_l \leftarrow$ Sign(sk, l), i.e., pick random $r_l \leftarrow \mathbb{Z}_p$, set $h_l \leftarrow g^{r_l}$ and output the signature $\sigma_l = (h_l, s_l) \leftarrow (h_l, h_l^{x+yl})$.
- Set the parameters $params \leftarrow (p, \mathbb{G}, \tilde{\mathbb{G}}, \mathbb{G}_t, e, g, \tilde{g}, \gamma_1, \gamma_2, \delta, \tilde{\alpha}_{sm}, \tilde{\beta}_{sm}, \sigma_0, \ldots, \sigma_{L-1}, L)$.
- Output $params$.

Spend$(pk, sk_{\mathcal{U}_j}, W, payinfo, V)$. Execute the following steps:

- Parse W as (σ, v, l). If $l + V - 1 \geq L$, output 0.
- Parse σ as (h, s).
- Parse pk as $(params, \tilde{\alpha}, \beta_1, \tilde{\beta}_1, \beta_2, \tilde{\beta}_2)$.
- Pick random $r \leftarrow \mathbb{Z}_p$ and $r' \leftarrow \mathbb{Z}_p$.
- Compute $\sigma' = (h', s') \leftarrow (h^{r'}, s^{r'}(h')^r)$.
- Compute $\kappa \leftarrow \tilde{\alpha}\tilde{\beta}_1^{sk_{\mathcal{U}_j}}\tilde{\beta}_2^v\tilde{g}^r$.
- Pick random $o_c \leftarrow \mathbb{Z}_p$ and compute the commitment $C \leftarrow g^{o_c}\gamma_1^v$.
- For $k \in [0, V-1]$, compute $R_k \leftarrow H'(payinfo, k)$, where $payinfo$ must contain the identifier of the merchant, and H' is a collision-resistant hash function.
- For $k \in [0, V-1]$, set $l_k \leftarrow l + k$, pick random o_{a_k} and compute $A_k = g^{o_{a_k}}\gamma_1^{l_k}$.
- For $k \in [0, V-1]$, do the following:
  - Compute $S_k \leftarrow f_{\delta,v}(l_k) = \delta^{1/(v+l_k+1)}$.
  - Compute $T_k \leftarrow g^{sk_{\mathcal{U}_j}}(f_{g,v}(l_k))^{R_k} = g^{sk_{\mathcal{U}_j}+R_k/(v+l_k+1)}$.
- For $k \in [0, V-1]$, compute the values $\mu_k \leftarrow 1/(v + l_k + 1)$ and $o_{\mu_k} \leftarrow -(o_{a_k} + o_c)\mu_k$.
- Parse $params$ as the tuple $(p, \mathbb{G}, \tilde{\mathbb{G}}, \mathbb{G}_t, e, g, \tilde{g}, \gamma_1, \gamma_2, \delta, \tilde{\alpha}_{sm}, \tilde{\beta}_{sm}, \sigma_0, \ldots, \sigma_{L-1}, L)$.
- For $k \in [0, V-1]$, pick random $r_k \leftarrow \mathbb{Z}_p$ and $r'_k \leftarrow \mathbb{Z}_p$.
- For $k \in [0, V-1]$, parse σ_{l_k} as (h_{l_k}, s_{l_k}). Compute $\sigma'_{l_k} = (h'_{l_k}, s'_{l_k}) \leftarrow (h_{l_k}^{r'_k}, s_{l_k}^{r'_k}(h'_{l_k})^{r_k})$.
- For $k \in [0, V-1]$, compute $\kappa_k \leftarrow \tilde{\alpha}_{sm}\tilde{\beta}_{sm}^{l_k}\tilde{g}^{r_k}$.
- Compute a ZK argument of knowledge π_v via the Fiat-Shamir heuristic for the following relation:

$$\begin{aligned}\pi_v = \text{NIZK}\{(sk_{\mathcal{U}_j}, v, r, o_c, \langle l_k, r_k, o_{a_k}, \mu_k, o_{\mu_k}\rangle_{k=0}^{V-1}) : \\ \kappa = \tilde{\alpha}\tilde{\beta}_1^{sk_{\mathcal{U}_j}}\tilde{\beta}_2^v\tilde{g}^r \ \wedge \ C = g^{o_c}\gamma_1^v \ \wedge \\ \langle A_k = g^{o_{a_k}}\gamma_1^{l_k} \ \wedge \ \kappa_k = \tilde{\alpha}_{sm}\tilde{\beta}_{sm}^{l_k}\tilde{g}^{r_k} \wedge \\ S_k = \delta^{\mu_k} \ \wedge \ \gamma_1 = (A_k C\gamma_1)^{\mu_k} g^{o_{\mu_k}} \ \wedge \\ T_k = g^{sk_{\mathcal{U}_j}}(g^{R_k})^{\mu_k} \ \rangle_{k\in[0,V-1]}\}\end{aligned}$$

  The equation $\kappa = \tilde{\alpha}\tilde{\beta}_1^{sk_{\mathcal{U}_j}}\tilde{\beta}_2^v\tilde{g}^r$ proves that κ commits to the signed messages. (κ is used as part of the proof of signature possession.) The equation $C = g^{o_c}\gamma_1^v$ proves that C commits to the same value v committed in κ. The equations $A_k = g^{o_{a_k}}\gamma_1^{l_k}$ and $\kappa_k = \tilde{\alpha}_{sm}\tilde{\beta}_{sm}^{l_k}\tilde{g}^{r_k}$ prove that the value l_k committed in A_k is in the valid range $[0, L-1]$ for a wallet with L coins. The equations $S_k = \delta^{\mu_k}, \gamma_1 = (A_k C\gamma_1)^{\mu_k} g^{o_{\mu_k}}$ and $T_k = g^{sk_{\mathcal{U}_j}}(g^{R_k})^{\mu_k}$ prove that the serial numbers S_k and the security tags T_k are correctly computed. This non-interactive argument signs the payment information $payinfo$.
- Output $pay \leftarrow (\kappa, \sigma', \langle S_k, T_k, A_k, \kappa_k, \sigma'_{l+k}\rangle_{k\in[0,V-1]}, V, C, \pi_v)$ as well as an updated wallet $W' \leftarrow (\sigma, v, l + V)$.

SpendVf$(pk, pay, payinfo)$. Execute the following steps:

- Parse pk as $(params, \tilde{\alpha}, \beta_1, \tilde{\beta}_1, \beta_2, \tilde{\beta}_2)$.
- Parse pay as $(\kappa, \sigma', \langle S_k, T_k, A_k, \kappa_k, \sigma'_{l_k}\rangle_{k\in[0,V-1]}, V, C, \pi_v)$.
- Parse σ' as (h', s') and output 0 if $h' = 1$ or if $e(h', \kappa) = e(s', \tilde{g})$ does not hold.
- For $k \in [0, V-1]$, parse σ'_{l_k} as (h'_{l_k}, s'_{l_k}) and output 0 if $h'_{l_k} = 1$ or if $e(h'_{l_k}, \kappa_k) = e(s'_{l_k}, \tilde{g})$ does not hold.
- Output 0 if not all the serial numbers $\langle S_k\rangle_{k\in[0,V-1]}$ are different from each other.
- For $k \in [0, V-1]$, compute $R_k \leftarrow H'(payinfo, k)$.
- Verify π_v by using $payinfo$, pk, $\langle S_k, T_k, A_k, R_k, \kappa_k\rangle_{k\in[0,V-1]}$, C and κ. Output 0 if the proof is not correct, else output V.

Cost reduction by using one secret in the wallet. We quantify the reduction of costs attained by having a wallet that signs one secret instead of two (as done in the compact e-cash scheme in [14]), in addition to the user secret key. First, we analyze the communication and storage costs. Let $|\mathbb{G}|$, $|\tilde{\mathbb{G}}|$ and $|\mathbb{Z}_p|$ denote the bit size of elements in $\mathbb{G}$, $\tilde{\mathbb{G}}$ and $\mathbb{Z}_p$ respectively. The size of the public parameters does not change. The size of public keys of authorities is $2|\mathbb{G}|+3|\tilde{\mathbb{G}}|$ with one secret and $3|\mathbb{G}| + 4|\tilde{\mathbb{G}}|$ with two secrets. The size of the secret key of authorities is $3|\mathbb{Z}_p|$ with one secret and $4|\mathbb{Z}_p|$ with two secrets. In the withdrawal phase, the size of a response does not change, but the size of a request is $8|\mathbb{G}| + 6|\mathbb{Z}_p|$ with one secret

and $10|\mathbb{G}| + 8|\mathbb{Z}_p|$ with two secrets. The wallet size is $2|\mathbb{G}| + 1|\mathbb{Z}_p|$ with one secret and $2|\mathbb{G}| + 2|\mathbb{Z}_p|$ with two secrets. The size of a payment of V coins is $(2 + 5V + 1)|\mathbb{G}| + (1 + V)|\tilde{\mathbb{G}}| + (5 + 5V)|\mathbb{Z}_p|$ with one secret and $(2 + 5V + 2)|\mathbb{G}| + (1 + V)|\tilde{\mathbb{G}}| + (7 + 7V)|\mathbb{Z}_p|$ with two secrets.

Second, we analyze the computation cost. We remark that the number of bilinear map computations does not change. Let $|M|$ and $|E|$ denote the cost of a multi-exponentiation and of a exponentiation respectively. In a withdrawal phase in which the user contacts t authorities, the total cost is $9|M| + (4 + 7t)|E|$ with one secret and $12|M| + (5 + 10t)|E|$ with two secrets. In a spending phase in which V coins are spent, the total cost is $(6 + 11V)|M| + (4 + 5V)|E|$ with one secret and $(9 + 13V)|M| + (5 + 5V)|E|$ with two secrets. The cost of the deposit phase does not change.

G COMPLETE DESCRIPTION OF THE DIVISIBLE E-CASH SCHEME

The zero-knowledge argument of knowledge in algorithm Spend in §5.2 involves proving knowledge of secret bases. For this purpose, the transformation described in §A.3 needs to be applied. In algorithm Setup, the generators $\psi \in \mathbb{G}$ and $\tilde{\psi} \in \tilde{\mathbb{G}}$ are added. In algorithm Spend, the secret bases are blinded, and the zero-knowledge argument is modified accordingly. We describe below the modified algorithms Setup, Spend and SpendVf. Moreover, we also describe the algorithms KeyGenV, KeyGenU, Request, RequestVf, Issue, IssueVf and AggrWallet, which were not depicted in §5.2.

Setup$(1^k, L)$. Execute the following steps:
- Run $grp = (p, \mathbb{G}, \tilde{\mathbb{G}}, \mathbb{G}_t, e, g, \tilde{g}) \leftarrow \mathcal{G}(1^k)$.
- Pick random generators $\eta, \gamma_1, \gamma_2, \psi \in \mathbb{G}$ and $\tilde{\psi} \in \tilde{\mathbb{G}}$.
- Generate random scalars $(z, y) \leftarrow \mathbb{Z}_p$ and, for $l \in [1, L]$, $a_l \leftarrow \mathbb{Z}_p$.
- Compute $(\varsigma, \theta) \leftarrow (g^z, \eta^z)$.
- For $l \in [1, L]$, compute $(\varsigma_l, \theta_l) \leftarrow (\varsigma^{y^l}, \theta^{y^l})$.
- For $k \in [0, L-1]$, compute $\tilde{\delta}_k \leftarrow \tilde{g}^{y^k}$.
- For $l \in [1, L]$, compute $\eta_l \leftarrow g^{a_l}$.
- For $l \in [1, L]$, for $k \in [0, l-1]$, compute $\tilde{\eta}_{l,k} \leftarrow \tilde{g}^{-a_l \cdot y^k}$.
- Run the algorithm $(pk_{sps}, sk_{sps}) \leftarrow$ KeyGen$(grp, 2, 0)$ of the structure-preserving signature scheme in §A.5.
- For $l \in [1, L]$, compute $\tau_l \leftarrow$ Sign$(sk_{sps}, \langle \varsigma_l, \theta_l \rangle)$.
- Set the parameters for users $params_u \leftarrow (p, \mathbb{G}, \tilde{\mathbb{G}}, \mathbb{G}_t, e, g, \tilde{g}, \eta, \gamma_1, \gamma_2, \{\eta_l, \varsigma_l, \theta_l, \tau_l\}_{l=1}^{L}, pk_{sps}, \psi, \tilde{\psi})$. Set the additional parameters for authorities $params_a \leftarrow (\{\tilde{\delta}_k\}_{k=0}^{L-1}, \{\langle \tilde{\eta}_{l,k} \rangle_{k=0}^{l-1}\}_{l=1}^{L-1})$.
- Set the parameters $params \leftarrow (params_u, params_a)$.
- Output $params$.

KeyGenV$(params_u, t, n)$. Execute the following steps:
- Choose $(1 + 2)$ polynomials (v, w_1, w_2) of degree $(t - 1)$ with random coefficients in $\mathbb{Z}_p$.
- Set $(x, y_1, y_2) \leftarrow (v(0), w_1(0), w_2(0))$.
- For $i = 1$ to n, set the secret key $sk_{\mathcal{V}_i}$ of each authority $\mathcal{V}_i$ as $sk_{\mathcal{V}_i} = (x_i, y_{i,1}, y_{i,2}) \leftarrow (v(i), w_1(i), w_2(i))$.
- For $i = 1$ to n, set the verification key $pk_{\mathcal{V}_i}$ of each authority $\mathcal{V}_i$ as $pk_{\mathcal{V}_i} = (\tilde{\alpha}_i, \beta_{i,1}, \tilde{\beta}_{i,1}, \beta_{i,2}, \tilde{\beta}_{i,2}) \leftarrow (\tilde{g}^{x_i}, g^{y_{i,1}}, \tilde{g}^{y_{i,1}}, g^{y_{i,2}}, \tilde{g}^{y_{i,2}})$.
- Compute the verification key $pk = (params_u, \tilde{\alpha}, \beta_1, \tilde{\beta}_1, \beta_2, \tilde{\beta}_2) \leftarrow (params_u, \tilde{g}^x, g^{y_1}, \tilde{g}^{y_1}, g^{y_2}, \tilde{g}^{y_2})$.
- Output $(pk, \langle pk_{\mathcal{V}_i}, sk_{\mathcal{V}_i} \rangle_{i=1}^{n})$.

KeyGenU$(params_u)$. Execute the following steps:
- Pick random $sk_{\mathcal{U}_j} \leftarrow \mathbb{Z}_p$ and compute $pk_{\mathcal{U}_j} \leftarrow g^{sk_{\mathcal{U}_j}}$.
- Output $(sk_{\mathcal{U}_j}, pk_{\mathcal{U}_j})$.

Request$(params_u, sk_{\mathcal{U}_j})$. Execute the following steps:
- Pick random $v \leftarrow \mathbb{Z}_p$ and set $(m_1, m_2) = (sk_{\mathcal{U}_j}, v)$.
- Pick random $o \leftarrow \mathbb{Z}_p$ and compute $com = g^o \prod_{j=1}^{2} \gamma_j^{m_j}$.
- Compute $h \leftarrow H(com)$, where H is modeled as a random oracle.
- Compute commitments to each of the messages. For $j = 1$ to 2, pick random $o_j \leftarrow \mathbb{Z}_p$ and set $com_j = g^{o_j} h^{m_j}$.
- Compute a ZK argument of knowledge π_s via the Fiat-Shamir heuristic for the following relation:

$$\pi_s = \text{NIZK}\{(m_1, m_2, o, o_1, o_2) : com = g^o \prod_{j=1}^{2} \gamma_j^{m_j} \ \wedge \ pk_{\mathcal{U}_j} \leftarrow g^{m_1} \wedge \{com_j = g^{o_j} h^{m_j}\}_{\forall j \in [1,2]}\}$$

- Set $reqinfo \leftarrow (h, o_1, o_2, v)$.
- Set $req \leftarrow (h, com, com_1, com_2, \pi_s)$.
- Output req and $reqinfo$.

RequestVf$(params_u, req, pk_{\mathcal{U}_j})$. Execute the following steps:
- Parse req as $(h, com, com_1, com_2, \pi_s)$.
- Compute $h' \leftarrow H(com)$, where H is modeled as a random oracle. Output 0 if $h \neq h'$.
- Verify π_s by using the tuple $(params_u, h, com, com_1, com_2)$. Output 0 if the proof π_s is not correct, else output 1.

Issue$(params_u, sk_{\mathcal{V}_i}, req)$. Execute the following steps:
- Parse req as $(h, com, com_1, com_2, \pi_s)$.
- Parse $sk_{\mathcal{V}_i}$ as $(x_i, y_{i,1}, y_{i,2})$.
- Compute $c = h^{x_i} \prod_{j=1}^{2} com_j^{y_{i,j}}$.
- Set the blinded signature share $\hat{\sigma}_i \leftarrow (h, c)$.
- Output $res \leftarrow \hat{\sigma}_i$.

IssueVf$(params_u, pk_{\mathcal{V}_i}, sk_{\mathcal{U}_j}, res, reqinfo)$. Execute the following steps:
- Parse $reqinfo$ as (h', o_1, o_2, v).
- Parse res as $\hat{\sigma}_i = (h, c)$. Output 0 if $h \neq h'$.
- Parse $pk_{\mathcal{V}_i}$ as $(\tilde{\alpha}_i, \beta_{i,1}, \tilde{\beta}_{i,1}, \beta_{i,2}, \tilde{\beta}_{i,2})$.
- Compute $\sigma_i = (h, s) \leftarrow (h, c \prod_{j=1}^{2} \beta_{i,j}^{-o_j})$.
- Set $(m_1, m_2) = (sk_{\mathcal{U}_j}, v)$. Output 0 if $e(h, \tilde{\alpha}_i \prod_{j=1}^{2} \tilde{\beta}_{i,j}^{m_j}) = e(s, \tilde{g})$ does not hold.
- Output $W_i \leftarrow (i, \sigma_i, v)$.

AggrWallet$(pk, sk_{\mathcal{U}_j}, \mathbb{S}, \langle W_i \rangle_{i \in \mathbb{S}})$. Execute the following steps:
- If $|\mathbb{S}| \neq t$, output 0.
- For all $i \in \mathbb{S}$, evaluate at 0 the Lagrange basis polynomials

$$l_i = [\prod_{j \in \mathbb{S}, j \neq i} (0 - j)][\prod_{j \in \mathbb{S}, j \neq i} (i - j)]^{-1} \bmod p$$

- For all $i \in \mathbb{S}$, parse W_i as (i, σ_i, v) and σ_i as (h, s_i).
- Compute the signature $\sigma = (h, s) \leftarrow (h, \prod_{i \in \mathbb{S}} s_i^{l_i})$.
- Parse pk as $(params_u, \tilde{\alpha}, \beta_1, \tilde{\beta}_1, \beta_2, \tilde{\beta}_2)$.
- Set $(m_1, m_2) = (sk_{\mathcal{U}_j}, v)$ and output 0 if $e(h, \tilde{\alpha} \prod_{j=1}^{2} \tilde{\beta}_j^{m_j}) = e(s, \tilde{g})$ does not hold, else output $W \leftarrow (\sigma, v, l)$, where l is a counter from 1 to L initialized to 1.

Spend($pk, sk_{\mathcal{U}_j}, W, payinfo, V$). Execute the following steps:

- Parse W as (σ, v, l). If $l + V \geq L$, output 0.
- Parse σ as (h, s).
- Parse pk as $(params_u, \tilde{\alpha}, \beta_1, \tilde{\beta}_1, \beta_2, \tilde{\beta}_2)$.
- Pick random scalars $r \leftarrow \mathbb{Z}_p$ and $r' \leftarrow \mathbb{Z}_p$.
- Compute $\sigma' = (h', s') \leftarrow (h^{r'}, s^{r'}(h')^r)$.
- Compute $\kappa \leftarrow \tilde{\alpha}\tilde{\beta}_1^{sk_{\mathcal{U}_j}}\tilde{\beta}_2^v\tilde{g}^r$.
- Pick random scalars $r_1, r_2 \leftarrow \mathbb{Z}_p$.
- Compute $\phi_{V,l} = (\phi_{V,l}[1], \phi_{V,l}[2]) \leftarrow (g^{r_1}, \varsigma_l^v\eta_V^{r_1})$.
- Set $R \leftarrow H(payinfo)$, where H is a collision-resistant hash function, and set $\varphi_{V,l} = (\varphi_{V,l}[1], \varphi_{V,l}[2]) \leftarrow (g^{r_2}, (g^R)^{sk_{\mathcal{U}_j}}\theta_l^v\eta_V^{r_2})$.
- Take $params_u$ from pk. Take the public key $pk_{sps} = (Y, W_1, W_2, Z)$ and the signature $\tau_{l+V-1} = (R_{l+V-1}, S_{l+V-1}, T_{l+V-1})$.
- Pick random

$$(\rho_{\varsigma l}, \rho_{\theta l}, \rho_{\varsigma l+V-1}, \rho_{\theta l+V-1}, \rho_{R_{l+V-1}}, \rho_{S_{l+V-1}}, \rho_{T_{l+V-1}}) \leftarrow \mathbb{Z}_p$$

- Compute the blinded bases $\varsigma'_l \leftarrow \varsigma_l\psi^{\rho_{\varsigma l}}$, $\theta'_l \leftarrow \theta_l\psi^{\rho_{\theta l}}$, $\varsigma'_{l+V-1} \leftarrow \varsigma_{l+V-1}\psi^{\rho_{\varsigma l+V-1}}$ and $\theta'_{l+V-1} \leftarrow \theta_{l+V-1}\psi^{\rho_{\theta l+V-1}}$, and the blinded bases for the signature $R'_{l+V-1} \leftarrow R_{l+V-1}\psi^{\rho_{R_{l+V-1}}}$, $S'_{l+V-1} \leftarrow S_{l+V-1}\psi^{\rho_{S_{l+V-1}}}$ and $T'_{l+V-1} \leftarrow T_{l+V-1}\tilde{\psi}^{\rho_{T_{l+V-1}}}$.
- Compute $\rho_1 \leftarrow -v\rho_{\varsigma l}$, $\rho_2 \leftarrow -v\rho_{\theta_l}$ and $\rho_3 \leftarrow \rho_{R_{l+V-1}}\rho_{T_{l+V-1}}$.
- Compute a ZK argument of knowledge π_v via the Fiat-Shamir heuristic for the following relation:

$$\begin{aligned}
\pi_v = \text{NIZK}\{&(sk_{\mathcal{U}_j}, v, r, r_1, r_2, \rho_{\varsigma l}, \rho_{\theta_l}, \rho_{\varsigma l+V-1}, \rho_{\theta_{l+V-1}}, \\
&\rho_{R_{l+V-1}}, \rho_{S_{l+V-1}}, \rho_{T_{l+V-1}}, \rho_1, \rho_2, \rho_3) : \\
&\kappa = \tilde{\alpha}\tilde{\beta}_1^{sk_{\mathcal{U}_j}}\tilde{\beta}_2^v\tilde{g}^r \ \wedge && (4)\\
&\phi_{V,l}[1] = g^{r_1} \ \wedge \ \phi_{V,l}[2] = (\varsigma'_l)^v\psi^{\rho_1}\eta_V^{r_1} \wedge && (5)\\
&\varphi_{V,l}[1] = g^{r_2} \ \wedge \ \varphi_{V,l}[2] = (g^R)^{sk_{\mathcal{U}_j}}(\theta'_l)^v\psi^{\rho_2}\eta_V^{r_2} \wedge && (6)\\
&e(\varsigma'_l, \tilde{\delta}_{V-1})e(\psi, \tilde{\delta}_{V-1})^{-\rho_{\varsigma l}} = e(\varsigma'_{l+V-1}, \tilde{g})e(\psi, \tilde{g})^{-\rho_{\varsigma l+V-1}} \wedge && (7)\\
&e(\theta'_l, \tilde{\delta}_{V-1})e(\psi, \tilde{\delta}_{V-1})^{-\rho_{\theta_l}} = e(\theta'_{l+V-1}, \tilde{g})e(\psi, \tilde{g})^{-\rho_{\theta_{l+V-1}}} \wedge && (8)\\
&e(R'_{l+V-1}, Y)e(\psi, Y)^{-\rho_{R_{l+V-1}}}e(S'_{l+V-1}, \tilde{g})e(\psi, \tilde{g})^{-\rho_{S_{l+V-1}}} \cdot \\
&e(\varsigma'_{l+V-1}, W_1)e(\psi, W_1)^{-\rho_{\varsigma_{l+V-1}}}e(\theta'_{l+V-1}, W_2)e(\psi, W_2)^{-\rho_{\theta_{l+V-1}}} \cdot \\
&e(g, Z)^{-1} = 1 \wedge && (9)\\
&e(R'_{l+V-1}, T'_{l+V-1})e(R'_{l+V-1}, \tilde{\psi})^{-\rho_{T_{l+V-1}}} \cdot \\
&e(\psi, T'_{l+V-1})^{-\rho_{R_{l+V-1}}}e(\psi, \tilde{\psi})^{\rho_3}e(g, \tilde{g})^{-1} = 1 && (10)
\end{aligned}$$

Equation 4 is part of the proof of possession of the wallet signature that signs $(sk_{\mathcal{U}_j}, v)$. Equation 5 and 6 prove that $\phi_{V,l}$ and the security tag $\varphi_{V,l}$ are well-formed and are computed on input $(sk_{\mathcal{U}_j}, v)$. Equation 7, 8, 9 and 10 prove that the values ς_l and θ_l, which were used to compute $\phi_{V,l}$ and $\varphi_{V,l}$ respectively, are part of the public parameters and fulfill $l \leq L - V + 1$. This is accomplished by proving possession of a signature on ς_{l+V-1} and θ_{l+V-1} in equations 9 and 10, and proving that the indices of ς_{l+V-1} and θ_{l+V-1} and ς_l and θ_l are related by a difference of $V - 1$ is equation 7 and 8 respectively. This non-interactive argument signs the payment information *payinfo*.

- Output a payment $pay \leftarrow (\kappa, \sigma', \phi_{V,l}, \varphi_{V,l}, \varsigma'_l, \theta'_l, \varsigma'_{l+V-1}, \theta'_{l+V-1}, R'_{l+V-1}, S'_{l+V-1}, T'_{l+V-1}, R, \pi_v, V)$ and an updated wallet $W' \leftarrow (\sigma, v, l + V)$.

SpendVf($pk, pay, payinfo$). Execute the following steps:

- Parse σ' as (h', s') and output 0 if $h' = 1$ or if $e(h', \kappa) = e(s', \tilde{g})$ does not hold.
- Output 0 if $R \neq H(payinfo)$ or if $payinfo$ does not contain the identifier of the merchant.
- Verify π_v by using $payinfo$, pk, $\phi_{V,l}$, $\varphi_{V,l}$, ς'_l, θ'_l, ς'_{l+V-1}, θ'_{l+V-1}, R'_{l+V-1}, S'_{l+V-1}, T'_{l+V-1}, V, R and κ. Output 0 if the proof is not correct, else output V.

Identify($params, PK, pay_1, pay_2, payinfo_1, payinfo_2$). Execute:

- Parse pay_1 as

$$\begin{aligned}pay_1 =&(\kappa_1, \sigma'_1, \phi_{V_1,l_1,1}, \varphi_{V_1,l_1,1}, \varsigma'_{l_1,1}, \theta'_{l_1,1}, \varsigma'_{l_1+V_1-1,1}, \theta'_{l_1+V_1-1,1}, \\ &R'_{l_1+V_1-1,1}, S'_{l_1+V_1-1,1}, T'_{l_1+V_1-1,1}, R_1, \pi_{v,1}, V_1).\end{aligned}$$

- Parse pay_2 as

$$\begin{aligned}pay_2 =&(\kappa_2, \sigma'_2, \phi_{V_2,l_2,2}, \varphi_{V_2,l_2,2}, \varsigma'_{l_2,2}, \theta'_{l_2,2}, \varsigma'_{l_2+V_2-1,2}, \theta'_{l_2+V_2-1,2}, \\ &R'_{l_2+V_2-1,2}, S'_{l_2+V_2-1,2}, T'_{l_2+V_2-1,2}, R_2, \pi_{v,2}, V_2).\end{aligned}$$

- For $k \in [0, V_1 - 1]$, compute the serial numbers

$$\text{SN}_{k,1} \leftarrow e(\phi_{V_1,l_1,1}[2], \tilde{\delta}_k)e(\phi_{V_1,l_1,1}[1], \tilde{\eta}_{V_1,k}).$$

  For $k \in [0, V_2 - 1]$, compute the serial numbers

$$\text{SN}_{k,2} \leftarrow e(\phi_{V_2,l_2,2}[2], \tilde{\delta}_k)e(\phi_{V_2,l_2,2}[1], \tilde{\eta}_{V_2,k}).$$

- Output 1 if none of the serial numbers $\text{SN}_{k_1,1}$, for $k_1 \in [0, V_1 - 1]$, is equal to $\text{SN}_{k_2,2}$, for $k_2 \in [0, V_2 - 1]$.
- Else, output $payinfo_1$ if $payinfo_1 = payinfo_2$.
- Else, let $k_1 \in [0, V_1 - 1]$ and $k_2 \in [0, V_2 - 1]$ be two indices such that $\text{SN}_{k_1,1} = \text{SN}_{k_2,2}$. Compute

$$T_1 \leftarrow e(\varphi_{V_1,l_1,1}[2], \tilde{\delta}_{k_1})e(\varphi_{V_1,l_1,1}[1], \tilde{\eta}_{V_1,k_1}),$$

  and

$$T_2 \leftarrow e(\varphi_{V_2,l_2,2}[2], \tilde{\delta}_{k_2})e(\varphi_{V_2,l_2,2}[1], \tilde{\eta}_{V_2,k_2}).$$

  For each $pk_{\mathcal{U}_j} \in \text{PK}$, check whether $T_1T_2^{-1} = e(pk_{\mathcal{U}_j}, \tilde{\delta}_{k_1}^{R_1}\tilde{\delta}_{k_2}^{-R_2})$ and output $pk_{\mathcal{U}_j}$ if the equality holds. Output $\perp$ if the equality does not hold for any $pk_{\mathcal{U}_j} \in \text{PK}$.

H INTEGRATION

In our schemes, deposited payments are verified by the authorities. Double-spending is detected by storing previous payments on a bulletin board and checking that serial numbers in a deposited payment are not equal to any serial number of previously submitted payments. A blockchain can be used to implement the bulletin board. When a provider wishes to deposit a payment, the payment is broadcast to authorities, which could also be n validators in a proof-of-stake blockchain. The consensus mechanism of the blockchain is then used to agree on whether the payment is valid and, in that case, recorded in the blockchain. This allows decentralizing also the deposit phase of our protocols.

Also, note that authorities in this setting may leave and enter the system dynamically. However, we must take into account that an authority that leaves the system still possesses a valid share of the

secret key, and we assume that, after leaving the system, the authority becomes corrupt. Similarly to decentralized e-cash schemes, in our schemes we assume an honest majority, i.e. if $n = 2a + 1$, there are at most a corrupt authorities. For our schemes to be secure, if a is the number of corrupt authorities currently in the system, and b is the number of authorities that have left, we need that $a + b < t$, where t is the threshold. Therefore, algorithm KeyGenV needs to be run periodically to create a new public verification key pk and new key pairs $(sk_{\mathcal{V}_i}, pk_{\mathcal{V}_i})_{i\in[1,n]}$ so as to ensure that $a + b < t$. This implies that e-cash expires whenever a new key is created. A time interval in which users can convert old wallets to use the new secret key can be given [23]. Once this interval ends, e-cash expiration makes it possible to delete the payments stored for double-spending detection and reset the bulletin board, which avoids an ever-growing blockchain [25]. The blockchain can thus be used as a settlement layer for offline e-cash transactions.

For our deployment, we have implemented a non-interactive distributed key generation protocol [34] to replace the KeyGenV algorithm. This protocol enables key resharing, which we use when the number of authorities that have left is such that $a + b < t$ holds, and computation of a new key, which we use when $a + b < t$ does not hold anymore.

This scheme would get the best of both worlds, that of online blockchain and offline e-cash, but further research needs to be done to specify this model formally and parameterize a real-world implementation. Threshold-issuance offline e-cash may end up solving the pressing problems of privacy and scalability of payments in both blockchain and even CBDCs as e-cash moves from theory into practice.